Faster Python Programs - Measure, don't Guess

A Tutorial at PyCon 2016

May 29, 2016

Portland, OR, USA

author: Dr.-Ing. Mike Müller

email: mmueller@python-academy.de

twitter: @pyacademy

version: 3.4

Python Academy - The Python Training Specialist

- Dedicated to Python training since 2006
- Wide variety of Python topics
- Experienced Python trainers
- Trainers are specialists and often core developers of taught software
- All levels of participants from novice to experienced software developers
- Courses for system administrators
- Courses for scientists and engineers
- Python for data analysis
- Open courses around Europe
- Customized in-house courses
- Python programming
- Development of technical and scientific software
- Code reviews
- Coaching of individuals and teams migrating to Python
- Workshops on developed software with detailed explanations

More Information: www.python-academy.com

Current Training Modules - Python Academy

As of 2016

Module Topic	Length (days)	In-house	Open
Python for Programmers	3	yes	yes
Python for Non-Programmers	4	yes	yes
Python for Programmers in Italian	3	yes	yes
Python for Scientists and Engineers	3	yes	yes
Python for Data Analysis	3	yes	yes
HPC with Python	3	yes	yes
Cython in Depth	2	yes	yes
Advanced Python	3	yes	yes
Introduction to Django	4	yes	yes
Advanced Django	3	yes	yes
Professional Testing with Python	3	yes	yes
SQLAlchemy	1	yes	yes
High Performance XML with Python	1	yes	yes
Camelot	1	yes	yes
Optimizing Python Programs	1	yes	yes
Python Extensions with Other Languages	1	yes	no
Data Storage with Python	1	yes	no
Introduction to Software Engineering with Python	1	yes	no
Introduction to wxPython	1	yes	no
Introduction to PySide/PyQt	1	yes	no
Overview of the Python Standard Library	1	yes	no
Threads and Processes in Python	1	yes	no
Windows Programming with Python	1	yes	no
Network Programming with Python	1	yes	no
Introduction to IronPython	1	yes	no

We offer on-site and open course all over Europe. We always customize and extend training modules as needed. We also provide consulting services such as code review, custom programming and tailor-made workshops.

More information: www.python-academy.com

Contents

1 How Fast is Fast Enough?

1.1 Introduction

Since Python is an interpreted language, some types of computations are slower in Python than in compiled languages. Depending on the application, this may or may not be a problem. This tutorial introduces several methods to speed up Python. Before starting to optimize, however the cost involved should be considered. Optimized code may need more effort to develop and maintain, leading to prolonged development time. So there is always a balance between speed of development and speed of program execution.

1.2 Optimization Guidelines

> Premature optimization is the root of all evil.
>
> D. Knuth or C. A. R. Hoare or folklore (attributions seems not totally clear)

Before you start thinking about optimization make sure your program works correctly. Never optimize before the program produces the desired results.

Optimization often comes with a price: It tends to make your code less readable. Since most of the programming time for software is spent on maintenance rather than developing new code, readability and maintainability is of great importance for an effective life cycle of your program. Therefore, always think twice if it is really worth before you make your code less readable the speed gain. After all, we deliberately choose Python for its excellent readability and pay with somewhat slower programs for certain tasks.

A few general guidelines are formulated as follows:

1. Make sure your program is really too slow. Do you really need more performance? Are there any other slowdown factors such as network traffic or user input that have more impact on speed? Does it hurt if the program runs slowly?
2. Don't optimize as you go. Don't waste time before you are certain that you will need the additional speed.
3. Only realistic use cases and user experience should be considered.
4. Architecture can be essential for performance. Is it appropriate?
5. Are there any bugs that slow down the program?
6. If the program is really too slow, find the bottlenecks by profiling (use module `profile`).
7. Always check the result of optimization with all unit tests. Don't optimize with bugs.

Usually for complex programs, the most performance gain can be achieved by optimization of algorithms. Finding what big-O notation an algorithm has is very important to predict performance for large amounts of data.

The first thing you should check before making any changes in your program is external causes that may slow down your program. Likely candidates are:

- network connections
- database access
- calls to system functions

1 How Fast is Fast Enough?

In most cases hardware is cheaper than programmer time. Always check if there is enough memory for application. Swapping memory pages to disc may slow down execution by an order of magnitude. Make sure you have plenty of free disk space and a recent and fast processor. The Python Cookbook [MART2005], also available online [1], is a very good compilation of short and not so short solutions to specific problems. Some of the recipes, especially in the algorithm section are applicable to performance issues.

The Python in a Nutshell book ([MART2006]) contains a good summary on optimization, including profiling as well as large-scale and small-scale optimization (see pages 474 to 489). There are two chapters about optimization in [ZIAD2008]. A good resource for scientific applications in Python is [LANG2006] that also contains substantial material on optimization and extending of Python with other languages.

Some of them are exemplified in the following section.

From now on we assume you have done all the above-mentioned steps and still need more speed.

2 Strategy

2.1 Measuring in Stones

Programs will run at different speeds on different hardware. The use of benchmarks allows to measure how fast your hardware and, in the case of Python, how fast the used implementation is. Python has the module `test.pystone` that allows to benchmark hardware and implementation. We can use it as a standalone script.

With Python 2.7:

```
$ python2.7 pystone2.py
Pystone(1.1) time for 50000 passes = 0.617802
This machine benchmarks at 80932.1 pystones/second
```

Python 3.4 is a little bit slower:

```
$ python3.4 pystone3.py
Pystone(1.2) time for 50000 passes = 0.721088
This machine benchmarks at 69339.7 pystones/second
```

PyPy is **significantly** faster than CPython and IronPython for this benchmark:

```
$ pypy pystone2.py
Pystone(1.1) time for 50000 passes = 0.082999
This machine benchmarks at 602417 pystones/second
```

But Jython is slower than CPython:

```
jython2.7 pystone2.py
Pystone(1.1) time for 50000 passes = 1,58169
This machine benchmarks at 31611,7 pystones/second
```

We can also use `pystone` in our programs:

```
>>> from test import pystone
>>> pystone.pystones()
(1.2585885652668103, 39727.04136987008)
```

The first value is the benchmark time in seconds and the second the pystones. We can use the pystone value to convert measured run times in seconds into pystones:

```
# file: pystone_converter.py

"""Convert seconds to kilo pystones."""

from test import pystone
```

```python
BENCHMARK_TIME, PYSTONES = pystone.pystones()


def kpystone_from_seconds(seconds):
    """Convert seconds to kilo pystones."""
    return (seconds * PYSTONES) / 1e3

if __name__ == '__main__':

    def test():
        """Show how it works
        """
        print
        print '%10s %10s' % ('seconds', 'kpystones')
        print
        for seconds in [0.1, 0.5, 1.0, 2.0, 5.0]:
            print ('%10.5f %10.5f' % (seconds, kpystone_from_seconds(seconds)))

    test()
```

We will use this function to compare our results.

2.2 Profiling CPU Usage

There are three modules in the Python standard library that allow measuring the used CPU time:

- `profile`
- `hotshot` and
- `cProfile`

Because `profile` is a pure Python implementation and `hotshot` might be removed in a future version of Python, `cProfile` is the recommended tool. It is part of the standard library for version 2.5 onwards. All three profilers are deterministic and therefore actually run the code they are profiling and measure its execution time. This has some overhead but provides reliable results in most cases. `cProfile` tries to minimize this overhead. Since Python works with the interpreter, the overhead is rather small. The other type of profiling is called statistical and uses random sampling of the effective instruction pointer. This has less overhead but is also less precise. We won't look at those techniques.

Let's write a small program whose whole purpose is to use up CPU time:

```python
# file profile_me.py

"""Example to be profiled.
"""
import sys
import time

if sys.version_info.major < 3:
    range = xrange


def fast():
    """Wait 0.001 seconds.
    """
    time.sleep(1e-3)


def slow():
    """Wait 0.1 seconds.
    """
    time.sleep(0.1)


def use_fast():
    """Call `fast` 100 times.
    """
    for _ in range(100):
        fast()


def use_slow():
    """Call `slow` 100 times.
    """
    for _ in range(100):
        slow()


if __name__ == '__main__':
    use_fast()
    use_slow()
```

2.2.1 *Working with Standard Python*

Now we import our module as well as `cProfile`:

```
>>> import profile_me
>>> import cProfile
```

and make an instance of `Profile`:

```
>>> profiler = cProfile.Profile()
```

First we call our fast function:

```
>>> profiler.runcall(profile_me.use_fast)
```

and look at the statistics `cProfile` provides:

```
>>> profiler.print_stats()
         202 function calls in 0.195 CPU seconds
   Ordered by: standard name
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      100    0.000    0.000    0.195    0.002 profile_me.py:3(fast)
        1    0.000    0.000    0.195    0.195 profile_me.py:9(use_fast)
        1    0.000    0.000    0.000    0.000 ~:0(<method 'disable' of
             ' _lsprof.Profiler' objects>)
      100    0.194    0.002    0.194    0.002 ~:0(<time.sleep>)
```

The column headers have the following meaning:

- `ncalls` is the number of calls to this function
- `tottime` is the total time spent in this function, where calls to sub-functions are excluded from time measurement
- `percall` is `tottime` divided by `ncalls`
- `cumtime` is the cumulative time, that is the total time spent in this including the time spent in sub-functions
- `percall` is `cumtime` divided by `ncalls`
- `filename:lineno(function)` are the name of the module, the line number and the name of the function

We can see that the function `fast` is called 100 times and that it takes about 0.002 seconds per call. At first look it is surprising that `tottme` is zero. But if we look at the time the function `time.sleep` uses up, it becomes clear the `fast` spends only 0.001 seconds (0.195 - 0.194 seconds) and the rest of the time is burnt in `time.sleep()`.

We can do the same thing for our slow function:

```
>>> profiler = cProfile.Profile()
>>> profiler.runcall(profile_me.use_slow)
>>> profiler.print_stats()
         202 function calls in 10.058 CPU seconds
   Ordered by: standard name
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.001    0.001   10.058   10.058 profile_me.py:13(use_slow)
      100    0.001    0.000   10.058    0.101 profile_me.py:6(slow)
        1    0.000    0.000    0.000    0.000 ~:0(<method 'disable' of
             '_lsprof.Profiler' objects>)
      100   10.057    0.101   10.057    0.101 ~:0(<time.sleep>)
```

Not surprisingly, the run times are nearly two orders of magnitude greater, because we let `sleep` use up one hundred times more time.

Another method to invoke the profiler is to use the function `run`:

```
>>> cProfile.run('profile_me.use_fast()')
         203 function calls in 0.195 CPU seconds
   Ordered by: standard name
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.195    0.195 <string>:1(<module>)
      100    0.000    0.000    0.195    0.002 profile_me.py:3(fast)
        1    0.000    0.000    0.195    0.195 profile_me.py:9(use_fast)
        1    0.000    0.000    0.000    0.000 ~:0(<method 'disable' of
            '_lsprof.Profiler' objects>)
      100    0.195    0.002    0.195    0.002 ~:0(<time.sleep>)
```

Here we supply the function to be called as a string with parenthesis, i.e. a string that can be used in an `exec` statement as opposed to the function object we supplied to the `runcall` method of our `Profile` instance.

We can also supply a file where the measured runtime data will be stored:

```
>>> cProfile.run('profile_me.use_fast()', 'fast.stats')
```

Now we can use the `pstats` module to analyze these data:

```
>>> cProfile.run('profile_me.use_fast()', 'fast.stats')
>>> import pstats
>>> stats = pstats.Stats('fast.stats')
```

We can just print out the data in the same format we saw before:

```
>>> stats.print_stats()
Wed Mar 11 16:11:39 2009    fast.stats
         203 function calls in 0.195 CPU seconds
   Random listing order was used
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      100    0.194    0.002    0.194    0.002 ~:0(<time.sleep>)
        1    0.000    0.000    0.000    0.000 ~:0(<method 'disable' of '_lsprof.Profiler'
objects>)
      100    0.000    0.000    0.195    0.002 profile_me.py:3(fast)
        1    0.000    0.000    0.195    0.195 <string>:1(<module>)
        1    0.000    0.000    0.195    0.195 profile_me.py:9(use_fast)
```

We can also sort by different columns and restrict the number of lines printed out. Here we sort by the number of calls and want to see only the first three columns:

```
>>> stats.sort_stats('calls').print_stats(3)
Wed Mar 11 16:11:39 2009    fast.stats
         203 function calls in 0.195 CPU seconds
   Ordered by: call count
   List reduced from 5 to 3 due to restriction <3>
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
```

```
       100    0.194    0.002    0.194    0.002 ~:0(<time.sleep>)
       100    0.000    0.000    0.195    0.002 profile_me.py:3(fast)
         1    0.000    0.000    0.000    0.000 ~:0(<method 'disable' of
              '_lsprof.Profiler' objects>)
```

Or we sort by time used and show all lines:

```
>>> stats.sort_stats('time').print_stats()
Wed Mar 11 16:11:39 2009    fast.stats
         203 function calls in 0.195 CPU seconds
   Ordered by: internal time
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      100    0.194    0.002    0.194    0.002 ~:0(<time.sleep>)
      100    0.000    0.000    0.195    0.002 profile_me.py:3(fast)
        1    0.000    0.000    0.195    0.195 profile_me.py:9(use_fast)
        1    0.000    0.000    0.195    0.195 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 ~:0(<method 'disable' of
             '_lsprof.Profiler' objects>)
```

We can also get information about which function is called by a certain function:

```
>>> stats.print_callers('fast')
   Ordered by: internal time
   List reduced from 5 to 2 due to restriction <'fast'>
Function                   was called by...
profile_me.py:3(fast)       profile_me.py:9(use_fast)
    ((100, 100, 0.00040628818660897912, 0 .19478914258667296))    0.195
profile_me.py:9(use_fast)   <string>:1(<module>)
    ((1, 1, 0.00026121123840443721, 0.1950503538250774))    0.195
```

We can also find out what functions are called:

```
>>> stats.print_callees('use_fast')
   Ordered by: internal time
   List reduced from 5 to 1 due to restriction <'use_fast'>
Function                   called...
profile_me.py:9(use_fast)   profile_me.py:3(fast)
    ((100, 100, 0.00040628818660897912, 0.19478914258667296))    0.195
```

There are more interesting attributes such as the number of calls:

```
>>> stats.total_calls
203
```

2.2.2   Working with IPython

In IPython you can get similar results as with `profile.run()` with the magic `%prun`:

```
%prun profile_me.use_fast()
```

```
         204 function calls in 0.114 seconds

Ordered by: internal time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   100    0.113    0.001    0.113    0.001 {built-in method sleep}
   100    0.001    0.000    0.114    0.001 profile_me.py:12(fast)
     1    0.000    0.000    0.114    0.114 profile_me.py:24(use_fast)
     1    0.000    0.000    0.114    0.114 {built-in method exec}
     1    0.000    0.000    0.114    0.114 <string>:1(<module>)
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects
```

You can limit the number of output lines:

```
%prun -l 2 profile_me.use_fast()

         204 function calls in 0.114 seconds
```

```
Ordered by: internal time
List reduced from 6 to 2 due to restriction <2>

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   100    0.113    0.001    0.113    0.001 {built-in method sleep}
   100    0.001    0.000    0.113    0.001 profile_me.py:12(fast)
```

Using a string as a filter. our output shows only functions with this string in their names. In our case word `fast`:

```
%prun -l fast profile_me.use_fast()
```

```
         204 function calls in 0.113 seconds

Ordered by: internal time
List reduced from 6 to 2 due to restriction <'fast'>

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   100    0.001    0.000    0.114    0.001 profile_me.py:12(fast)
     1    0.001    0.001    0.115    0.115 profile_me.py:24(use_fast)
```

We can create an `pstats.Stats` object:

```
stats = %prun -r profile_me.use_fast()
```

That behaves as the one generated with standard Python:

```
stats.sort_stats('calls').print_stats(3)
```

```
          204 function calls in 0.113 seconds

Ordered by: call count
List reduced from 6 to 3 due to restriction <3>

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   100    0.001    0.000    0.113    0.001 profile_me.py:12(fast)
   100    0.112    0.001    0.112    0.001 {built-in method sleep}
     1    0.000    0.000    0.113    0.113 {built-in method exec}
```

The option `-s` sorts and works multiple times:

```
%prun -s calls -s time profile_me.use_fast()
```

```
      204 function calls in 0.113 seconds

Ordered by: call count, internal time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   100    0.113    0.001    0.113    0.001 {built-in method sleep}
   100    0.000    0.000    0.113    0.001 profile_me.py:12(fast)
     1    0.000    0.000    0.113    0.113 profile_me.py:24(use_fast)
     1    0.000    0.000    0.113    0.113 {built-in method exec}
     1    0.000    0.000    0.113    0.113 <string>:1(<module>)
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects
```

You can also save your results to a file. Either the text as seen on the screen:

```
%prun -T stats.txt profile_me.use_fast()
```

```
*** Profile printout saved to text file 'stats.txt'.
```

now `stats.txt` contains the screen output.

Or, you dump it into a stats file for later use with other tools:

```
%prun -D ipython.stats profile_me.use_fast()
```

```
*** Profile stats marshalled to file 'ipython.stats'.
```

2.3 Wall Clock vs. CPU Time

Per default `cProfile` measures wall clock time, i.e. the time elapsed between start and end of the function. Since typically computers do more than one thing at a time, this times usually does not correspond with the usage time of the CPU. Also, often computers often wait for IO. During this time the CPU is more or less idle but the time elapses nevertheless.

Unfortunately, there are differences between operating systems in how they measure CPU time. Let's look at a simple test function:

```
"""Checking different timing functions.
"""

from __future__ import print_function

import os
import sys
import time
import timeit


if sys.version_info.major < 3:
    range = xrange


def clock_check(duration=1):
    """Check the measured time with different methods.
    """
    start_os_time0 = os.times()[0]
    start_time_clock = time.clock()
    start_default_timer = timeit.default_timer()
    for _ in range(int(1e6)):
        1 + 1
    time.sleep(duration)
    durtation_os_time0 = os.times()[0] - start_os_time0
    durtation_time_clock = time.clock() - start_time_clock
    durtation_default_timer = timeit.default_timer() - start_default_timer
    print('durtation_os_time0:     ', durtation_os_time0)
    print('durtation_time_clock:   ', durtation_time_clock)
    print('durtation_default_timer:', durtation_default_timer)


if __name__ == '__main__':
    clock_check()
```

We use three different methods to get time stamps:

1. `os.times()[0]` provides the CPU time on all operating systems. While it has six decimals, i.e. microseconds accuracy on Windows, it is only two significant decimals on Unix-like systems.
2. `start_time_clock = time.clock()` is the CPU time on Unix but the wall clock time on Windows.

3. `timeit.default_timer()` chooses the right timing function for wall clock, i.e. `time.time()` on Unix and `time.clock()` on windows.

This is our output on Unix/Linux/MacOSX:

```
durtation_os_time0:      0.05
durtation_time_clock:    0.041949
durtation_default_timer: 1.04296183586
```

and on Windows:

```
durtation_os_time0:      0.03125
durtation_time_clock:    1.02673477293
durtation_default_timer: 1.02673567261
```

We write a script to look at how `cProfile` can be used to measure both, wall clock and CPU time:

```python
# file: cpu_time.py

"""Measuring CPU time instead of wall clock time.
"""

import cProfile
import os
import sys
import time

# Make it work with Python 2 and Python 3.
if sys.version_info.major < 3:
    range = xrange
```

After some imports and a Python 2/3 compatibility helper, we define a function to measure CPU time that is aware of the differences between operating systems:

```python
def cpu_time():
    """Function for cpu time. Os dependent.
    """
    if sys.platform == 'win32':
        return os.times()[0]
    else:
        return time.clock()
```

We use two functions:

```python
def sleep():
    """Wait 2 seconds.
    """
    time.sleep(2)
```

```python


def count():
    """100 million loops.
    """
    for _ in range(int(1e8)):
        1 + 1


def test():
    """Run functions
    """
    sleep()
    count()
```

One that sleeps and one that just loops many times to consume CPU time. We put them into a test function and use `cProfile` with different timing methods:

```python
def profile():
    """Profile with wall clock and cpu time.
    """
    profiler = cProfile.Profile()
    profiler.run('test()')
    profiler.print_stats()

    profiler = cProfile.Profile(cpu_time)
    profiler.run('test()')
    profiler.print_stats()

if __name__ == '__main__':
    profile()
```

Without providing a time measurement function: `profiler = cProfile.Profile()`, we get the default wall clock timing.

Providing our timer function: `cProfile.Profile(cpu_time)`, we get the CPU time.

The output on Windows:

```
$ cpu_time.py
         6 function calls in 5.233 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.010    0.010    5.233    5.233 <string>:1(<module>)
        1    0.000    0.000    2.000    2.000 cpu_time.py:25(sleep)
        1    3.222    3.222    3.222    3.222 cpu_time.py:31(count)
        1    0.000    0.000    5.222    5.222 cpu_time.py:38(test)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objec
        1    2.000    2.000    2.000    2.000 {time.sleep}


         6 function calls in 3.141 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    3.141    3.141 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 cpu_time.py:25(sleep)
        1    3.141    3.141    3.141    3.141 cpu_time.py:31(count)
        1    0.000    0.000    3.141    3.141 cpu_time.py:38(test)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objec
        1    0.000    0.000    0.000    0.000 {time.sleep}
```

And the output on Unix/Linux/MacOSX:

```
$ python cpu_time.py
         6 function calls in 7.171 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    7.171    7.171 <string>:1(<module>)
        1    0.000    0.000    2.001    2.001 cpu_time.py:25(sleep)
        1    5.169    5.169    5.169    5.169 cpu_time.py:31(count)
        1    0.000    0.000    7.171    7.171 cpu_time.py:38(test)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objec
        1    2.001    2.001    2.001    2.001 {time.sleep}


         6 function calls in 4.360 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    4.360    4.360 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 cpu_time.py:25(sleep)
        1    4.360    4.360    4.360    4.360 cpu_time.py:31(count)
        1    0.000    0.000    4.360    4.360 cpu_time.py:38(test)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objec
        1    0.000    0.000    0.000    0.000 {time.sleep}
```

Both seem to correspond.

We can also time with Python 3:

```
$ python3 cpu_time.py
         7 function calls in 6.223 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    6.223    6.223 <string>:1(<module>)
        1    0.000    0.000    2.000    2.000 cpu_time.py:25(sleep)
        1    4.223    4.223    4.223    4.223 cpu_time.py:31(count)
        1    0.000    0.000    6.223    6.223 cpu_time.py:38(test)
        1    0.000    0.000    6.223    6.223 {built-in method exec}
        1    2.000    2.000    2.000    2.000 {built-in method sleep}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objec


         7 function calls in 4.231 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    4.231    4.231 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 cpu_time.py:25(sleep)
        1    4.230    4.230    4.230    4.230 cpu_time.py:31(count)
        1    0.000    0.000    4.231    4.231 cpu_time.py:38(test)
        1    0.000    0.000    4.231    4.231 {built-in method exec}
        1    0.000    0.000    0.000    0.000 {built-in method sleep}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objec
```

Conclusion: Always be aware what you are actual measuring. Don't assume to be on particular operating system, try to make your program run cross platform.

2.4   A More Complex Function

We want explore a bit more fancy function. This is one to calculate `pi` with the Monte Carlo method:

```python
# file: simple_pi.py

"""Calculating pi with Monte Carlo.
"""

from __future__ import print_function

import math
import random
import sys


if sys.version_info[0] < 3:
    range = xrange
```

```
def pi_plain(total):
    """Calculate pi with `total` hits.
    """
    count_inside = 0
    for _ in range(total):
        x = random.random()
        y = random.random()
        dist = math.sqrt(x * x + y * y)
        if dist < 1:
            count_inside += 1
    return 4.0 * count_inside / total

if __name__ == '__main__':

    def test():
        """Check if it works.
        """
        n = int(1e6)
        print('pi:', pi_plain(n))

    test()
```

We can also use NumPy for this:

```
# file: numpy_pi.py
"""Calculating pi with Monte Carlo Method and NumPy.
"""

from __future__ import print_function

import numpy                                                  #1


def pi_numpy(total):                                          #2
    """Compute pi.
    """
    x = numpy.random.rand(total)                              #3
    y = numpy.random.rand(total)                              #4
    dist = numpy.sqrt(x * x + y * y)                          #5
    count_inside = len(dist[dist < 1])                        #6
    return 4.0 * count_inside / total

if __name__ == '__main__':

    def test():
        """Time the execution.
        """
        import timeit
        start = timeit.default_timer()
        pi_numpy(int(1e6))
        print('run time', timeit.default_timer() - start)
    test()
```

2.5 A Picture is Worth a Thousand Words

Doing the statistics with tables is worthwhile and interesting. But there is another way to look at the profiling results: making graphs.

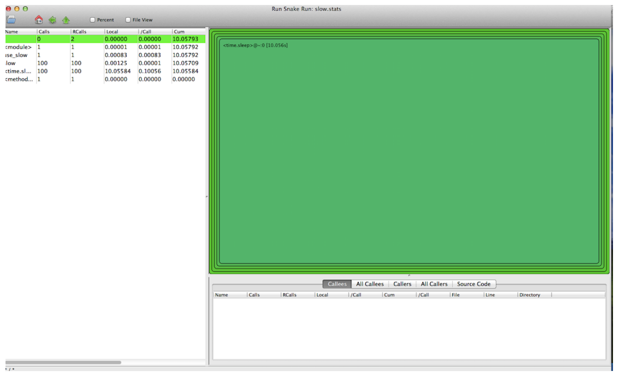
2.5.1 *RunSnakeRun*

A very nice tool for this is RunSnakeRun [2]. It is written in Python itself and uses wxPython and SquareMap. Unfortunately, it does not support Python 3 yet. Also, there are problems installing it in conda environments.

The usage is very simple. After installing RunSnakeRun just type:

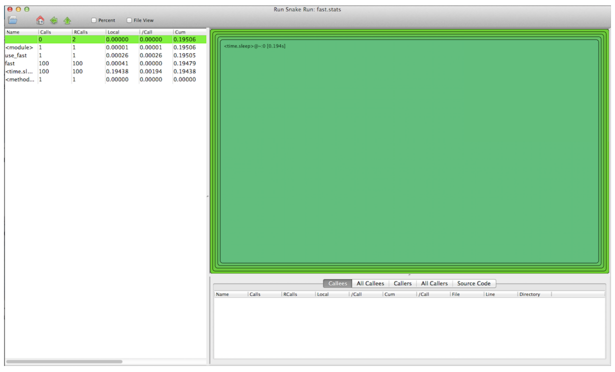
```
runsnake slow.stats
```

at the command line and you will get nice interactive graphs that should look like this for our slow example:



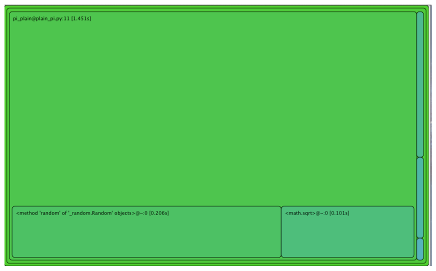
Name	Calls	RCalls	Local	/Call	Cum
	0	2	0.00000	0.00000	10.05793
module>	1	1	0.00001	0.00001	10.05792
se_slow	1	1	0.00083	0.00083	10.05792
low	100	100	0.00125	0.00001	10.05709
time.sl...	100	100	10.05584	0.10056	10.05584
method...	1	1	0.00000	0.00000	0.00000

Our fast example is not really fast and the graphical view shows a very similar picture

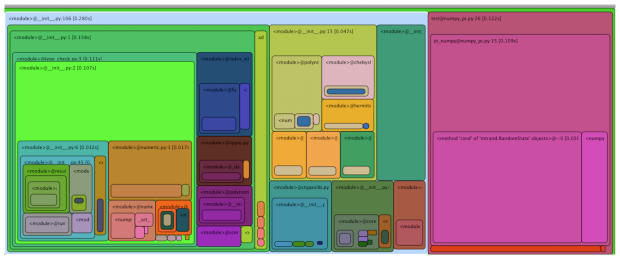


Name	Calls	RCalls	Local	/Call	Cum
	0	2	0.00000	0.00000	0.19506
<module>	1	1	0.00001	0.00001	0.19506
use_fast	1	1	0.00026	0.00026	0.19505
fast	100	100	0.00041	0.00000	0.19479
<time.sl...	100	100	0.19438	0.00194	0.19438
<method...	1	1	0.00000	0.00000	0.00000

Our calculation of pi gives a more interesting picture:



Even though our code NumPy version has about the same number of lines, the graph becomes much more complex because we use NumPy functions:



Note that a large part is spent during initialization. If you do the profiling interactively and repeat it several times, you might a different result, because the initialization has to happen only ones.

2.5.2 SnakeVis

Another visualization tool is SnakeVis [3]. It is inspired by RunSnakeRun but uses a different visualization technique. It runs on Python 3. The installation via `pip` is standard. There is a command line version `snakeviz`. Calling it with a name of a file that contains profiling information from the command line will open a browser with the visualization.

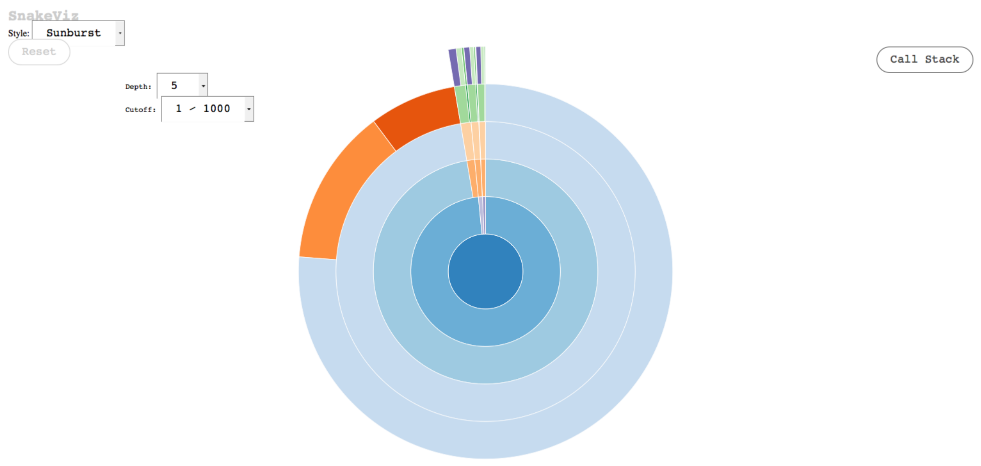
We create our profiling statistics at the command line:

```
python -m cProfile -o pi.stat simple_pi.py
```

Now, we start SnakeViz:

```
snakeviz pi.stat
```

This graph will show up in the browser:



The logic of the results is somewhat reversed to that of RunSnakeRun. The functions are color-coded. One color represents one function. The main function is the closed circle in the middle. The functions it calls are represented by the arcs outside of it. Hovering over an arc with the mouse, will highlight it and show the portion o the time that is used up in the function itself. The rest will be used by functions it calls. In our example the function `pi()` is the second outer arc. It also takes up the majority of the outermost arc because a lot the run time is spent in this function itself. The second biggest arc is `random()` followed by `sqrt()`.

Clicking on an arc will reset the display in such a way that the selected function becomes the full circle in the middle and the other, more outer, functions are rearranged accordingly. This works like a zoom.

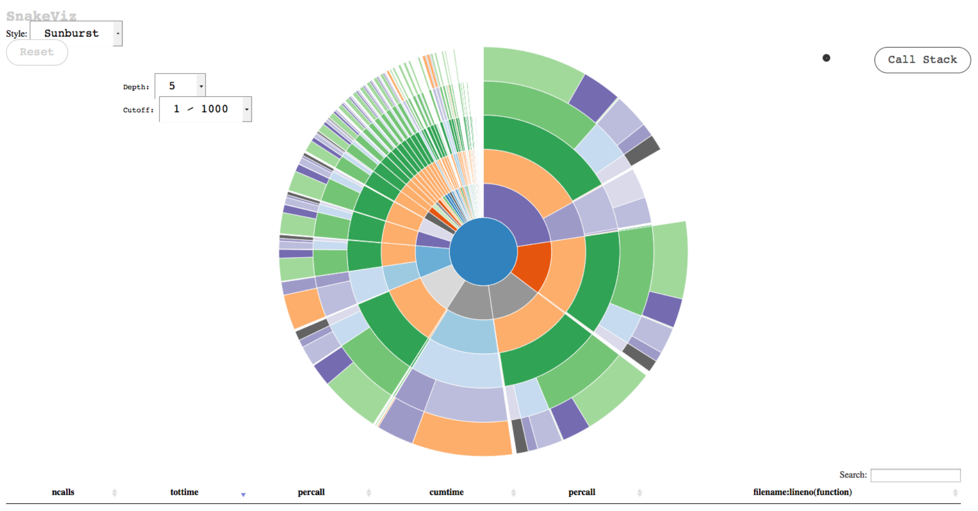
We can use the NumPy version. Creating the profiling statistics:

```
python -m cProfile -o numpy_pi.stats  numpy_pi.py
```

Showing the visualization:

```
pi mike$ snakeviz numpy_pi.stats
```

yields a much more complex graph:



Many functions are called by more than one function. This results in a color appearing at many isolated places.

There is also an extension for IPython. We can install it with

```
%load_ext snakeviz
```

and visualize one line of code:

```
%snakeviz func()
```

or a whole cell with multiple lines of code:

```
%%snakeviz
# code for profiling and visualization
# here
```

2.6 Going Line-by-Line

With `cProfile` the finest resolution we get is the function call. But there is `line_profiler` by Robert Kern that allows line-by-line profiling. `line_profiler` comes bundled with `kernprof` that adds some features to `cProfile`. The installation is simple:

```
pip install line_profiler
```

We can use `kernprof` from the command line, which just uses `cProfile`. The option `-v` shows the statistics right away:

```
$ kernprof -v profile_me.py
Wrote profile results to profile_me.py.prof
         406 function calls in 10.204 seconds
```

```
Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000   10.204   10.204 <string>:1(<module>)
   100    0.001    0.000   10.081    0.101 profile_me.py:15(slow)
     1    0.001    0.001    0.121    0.121 profile_me.py:21(use_fast)
     1    0.001    0.001   10.082   10.082 profile_me.py:28(use_slow)
     1    0.001    0.001   10.204   10.204 profile_me.py:4(<module>)
   100    0.001    0.000    0.120    0.001 profile_me.py:9(fast)
     1    0.000    0.000   10.204   10.204 {execfile}
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects
   200   10.199    0.051   10.199    0.051 {time.sleep}
```

We add the decorator `profile` to the function we would like to profile:

```python
# file profile_me_use_line_profiler.py

"""Example to be profiled.
"""

import time
import sys

if sys.version_info.major < 3:
    range = xrange


def fast():
    """Wait 0.001 seconds.
    """
    time.sleep(1e-3)


def slow():
    """Wait 0.1 seconds.
    """
    time.sleep(0.1)

@profile
def use_fast():
    """Call `fast` 100 times.
    """
    for _ in range(100):
        fast()

@profile
def use_slow():
    """Call `slow` 100 times.
    """
    for _ in range(100):
        slow()


if __name__ == '__main__':
    use_fast()
    use_slow()
```

Now we can use the option `-l` to turn on `line_profiler`:

```
$ kernprof -l -v profile_me_use_line_profiler.py
Wrote profile results to profile_me_use_line_profiler.py.lprof
Timer unit: 1e-06 s

File: profile_me_use_line_profiler.py
Function: use_fast at line 20
Total time: 0.120634 s

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    20                                           @profile
    21                                           def use_fast():
    22                                               """Call `fast` 100 times.
    23                                               """
    24       101          732      7.2      0.6      for _ in range(100):
    25       100       119902   1199.0     99.4          fast()

File: profile_me_use_line_profiler.py
Function: use_slow at line 27
Total time: 10.086 s

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    27                                           @profile
    28                                           def use_slow():
    29                                               """Call `slow` 100 times.
    30                                               """
    31       101         1147     11.4      0.0      for _ in range(100):
    32       100     10084845 100848.4    100.0          slow()
```

This shows us how much time each line used. Our test functions are very short. Let's create a small function that accumulates the sums of all elements in a list:

```
# file accumulate.py

"""Simple test function for line_profiler.
"""
```

```
@profile
def accumulate(iterable):
    """Accumulate the intermediate steps in summing all elements.

    The result is a list with the length of `iterable`.
    The last element is the sum of all elements of `iterable`
    >>>accumulate(range(5))
    [0, 1, 3, 6, 10]
    accumulate(range(10))
    [0, 1, 3, 6, 10, 15, 21, 28, 36, 45]
    """
    acm = [iterable[0]]
    for elem in iterable[1:]:
        old_value = acm[-1]
        new_value = old_value + elem
        acm.append(new_value)
    return acm


if __name__ == '__main__':
    accumulate(range(10))
    accumulate(range(100))
```

Let's look at the output:

```
$ kernprof -l -v accumulate.py
Wrote profile results to accumulate.py.lprof
Timer unit: 1e-06 s

File: accumulate.py
Function: accumulate at line 3
Total time: 0.000425 s

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     3                                           @profile
     4                                           def accumulate(iterable):
     5                                               """Accumulate the intermediate steps i
     6
     7                                               The result is a list with the lenght o
     8                                               The last elments is the sum of all ele
     9                                               >>>accumulate(range(5))
    10                                               [0, 1, 3, 6, 10]
    11                                               accumulate(range(10))
    12                                               [0, 1, 3, 6, 10, 15, 21, 28, 36, 45]
    13                                               """
    14         2            5      2.5      1.2      acm = [iterable[0]]
    15       110           99      0.9     23.3      for elem in iterable[1:]:
    16       108           94      0.9     22.1          old_value = acm[-1]
    17         108             98       0.9      23.1          new_value = old_value + elem
    18         108            127       1.2      29.9          acm.append(new_value)
    19           2              2       1.0       0.5      return acm
```

The algorithm could be written more concisely. In fact, the three lines inside the loop could be one. But we would like to see how much each operation takes and therefore spread things over several lines.

Another example looks at some simple mathematical calculations:

```
#calc.py

"""Simple test function for line_profiler doing some math.
"""

import math
import sys

if sys.version_info.major < 3:
    range = xrange


@profile
def calc(number, loops=1000):
    """Do some math calculations.
    """
    sqrt = math.sqrt
    for x in range(loops):
        x = number + 10
        x = number * 10
        x = number ** 10
        x = pow(number, 10)
        x = math.sqrt(number)
        x = sqrt(number)
        math.sqrt
        sqrt

if __name__ == '__main__':
    calc(100, int(1e5))
```

The output shows which operation takes the most time:

```
$ kernprof -l -v calc.py
Wrote profile results to calc.py.lprof
Timer unit: 1e-06 s

File: calc.py
Function: calc at line 7
Total time: 1.33158 s
```

```
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     7                                           @profile
     8                                           def calc(number, loops=1000):
     9                                               """Do some math calculations.
    10                                               """
    11         1            4      4.0      0.0      sqrt = math.sqrt
    12    100001        77315      0.8      5.8      for x in xrange(loops):
    13    100000        87124      0.9      6.5          x = number + 10
    14    100000        84518      0.8      6.3          x = number * 10
    15    100000       330587      3.3     24.8          x = number ** 10
    16    100000       378584      3.8     28.4          x = pow(x, 10)
    17    100000       109849      1.1      8.2          x = math.sqrt(number)
    18    100000        93211      0.9      7.0          x = sqrt(number)
    19    100000        88768      0.9      6.7          math.sqrt
    20    100000        81624      0.8      6.1          sqrt
```

The function `pow` takes by far the most time, whereas `sqrt` from the math module is fast. Note that there seems to be no difference between `math.sqrt` and `sqrt`, which is just a local reference. Let's look at this in a further example:

```python
# local_ref.py

"""Testing access to local name and name referenced in another module.
"""

import math
import sys

if sys.version_info.major < 3:
    range = xrange

# If there is no decorator `profile`, make one that just calls the function,
# i.e. does nothing.
# This allows to call `kernprof` with and without the option `-l` without
# commenting or un-commentimg `@profile' all the time.
# You can add this to the builtins to make it available in the whole program.
try:
    @profile
    def dummy():
        """Needs to be here to avoid a syntax error.
        """
        pass
except NameError:
    def profile(func):
        """Will act as the decorator `profile` if it was already found.
        """
        return func
```

```
@profile
def local_ref(counter):
    """Access local name.
    """
    # make it local
    sqrt = math.sqrt
    for _ in range(counter):
        sqrt

@profile
def module_ref(counter):
    """Access name as attribute of another module.
    """
    for _ in range(counter):
        math.sqrt


@profile
def test(counter):
    """Call both functions.
    """
    local_ref(counter)
    module_ref(counter)

if __name__ == '__main__':
    test(int(1e7))
```

There are two functions to be line-traced. `local_ref` gets a local reference to `math.sqrt` and `module_ref` calls `math.sqrt` as it is.

We run this with the option `-v`, and we get:

```
$ kernprof  -v local_ref.py
Wrote profile results to local_ref.py.prof
         9 function calls in 14.847 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000   14.847   14.847 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 local_ref.py:18(profile)
        1    0.001    0.001   14.846   14.846 local_ref.py:2(<module>)
        1    0.000    0.000   14.845   14.845 local_ref.py:21(mock)
        1    4.752    4.752    4.752    4.752 local_ref.py:28(local_ref)
        1   10.093   10.093   10.093   10.093 local_ref.py:37(module_ref)
        1    0.000    0.000   14.845   14.845 local_ref.py:44(test)
        1    0.001    0.001   14.847   14.847 {execfile}
        1    0.000    0.000    0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}
```

This shows that `local_ref` is more than twice as fast as `module_ref` because it avoids many lookups on the module `math`.

Now we run it with the options `-v -l`:

```
$ kernprof  -v -l local_ref.py
Wrote profile results to local_ref.py.lprof
Timer unit: 1e-06 s

File: local_ref.py
Function: dummy at line 12
Total time: 0 s

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    12                                           @profile
    13                                           def dummy():
    14                                               """Needs to be here to
avoid a syntax error.
    15                                               """
    16                                               pass

File: local_ref.py
Function: test at line 44
Total time: 125.934 s

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    44                                           @profile
    45                                           def test(counter):
    46                                               """Call both functions.
    47                                               """
    48         1     58162627 58162627.0     46.2      local_ref(counter)
    49         1     67771433 67771433.0     53.8      module_ref(counter)
```

This takes much longer. The differences in run times are largely gone. After correspondence with Robert Kern, the author of `line_profiler`, it turns out that the substantial overhead the line tracing adds causes a distortion of measuring results. Conclusion: Use `line_profiler` for expensive atomic calls such as to a function in an extension module like NumPy.

2.6.1 Exercise

Profile these two functions with `cProfile` and with `%prun` in an IPython Notebook as well as with the line profiler. Save the results and visualize them with SnakeVis.

```
# file: create_list.py

import sys

if sys.version_info.major < 3:
    range = xrange


def insert_zero(n=int(1e4)):
    """Assemble list with `insert`. Inefficient.
    """
    L = []
    for x in range(n):
        L.insert(0, x)
    return L


def append_reverse(n=int(1e4)):
    """Assemble list with `append` and `reverse`.
    """
    L = []
    for x in range(n):
        L.append(x)
    L.reverse()
    return L
```

2.7 Profiling Memory Usage

Current computers have lots of RAM, still it can be a problem if an application uses more RAM than is physically available, leading to swapping and a large performance penalty. In particular, long running applications tend to use up more RAM over time. Although Python does automatic memory management, there are cases where memory is not released because there are still references to objects that are no longer needed. We can call the garbage collector manually, but this does not always produce the desired effects.

2.7.1 Heapy

The Guppy_PE framework [4] provides the tool `heapy` that is very useful for inspecting Python memory usage. It is not the easiest tool to work with but still provides valuable insights in how memory is used by Python objects. Currently, it is only available for Python 2.

Installing for Python 2.x is easy.:

```
pip install guppy
```

We look at some features at the interactive prompt. First we import `hpy` and call it:

```
>>> from guppy import hpy
>>> hp = hpy()
```

Now we can look at the heap:

```
>>> hp.heap()
Partition of a set of 55690 objects. Total size = 3848216 bytes.
```

```
 Index  Count   %     Size   % Cumulative  % Kind (class / dict of class)
     0  27680  50  1522412  40    1522412  40 str
     1    150   0   666120  17    2188532  57 dict of module
     2  10459  19   474800  12    2663332  69 tuple
     3   2913   5   186432   5    2849764  74 types.CodeType
     4   2814   5   168840   4    3018604  78 function
     5    368   1   167488   4    3186092  83 dict (no owner)
     6    345   1   151596   4    3337688  87 dict of class
     7    145   0    90956   2    3428644  89 dict of type
     8    192   0    82156   2    3510800  91 type
     9   6310  11    75720   2    3586520  93 int
<140 more rows. Type e.g. '_.more' to view.>
```

There are 150 types of objects in our fresh interactive session. 40 % of the memory is taken up by strings and 17 % by module name space dictionaries.

We create a new object, a list with one million integers:

```
>>> big_list = list(range(int(1e6)))
```

and look at our heap again:

```
>>> hp.heap()
Partition of a set of 1055602 objects. Total size = 19924092 bytes.
 Index  Count   %     Size   % Cumulative  % Kind (class / dict of class)
     0 1006210  95 12074520  61   12074520  61 int
     1    208   0  4080320  20  16154840  81 list
     2  27685   3  1522628   8  17677468  89 str
     3    150   0   666120   3  18343588  92 dict of module
     4  10458   1   474768   2  18818356  94 tuple
     5   2913   0   186432   1  19004788  95 types.CodeType
     6   2813   0   168780   1  19173568  96 function
     7    374   0   168328   1  19341896  97 dict (no owner)
     8    345   0   151596   1  19493492  98 dict of class
     9    145   0    90956   0  19584448  98 dict of type
<140 more rows. Type e.g. '_.more' to view.>
```

Now integers, of which we have one million in our list, take up 61 % of the memory followed by lists that use up 20 %. Strings are down to 8%. We delete our list:

```
>>> del big_list
```

and we are (nearly) back to our initial state:

```
>>> hp.heap()
Partition of a set of 55700 objects. Total size = 3861984 bytes.
 Index  Count   %     Size   % Cumulative  % Kind (class / dict of class)
     0  27685  50  1522632  39    1522632  39 str
     1    150   0   666120  17   2188752  57 dict of module
     2  10458  19   474768  12   2663520  69 tuple
     3   2913   5   186432   5   2849952  74 types.CodeType
     4   2813   5   168780   4   3018732  78 function
     5    374   1   168328   4   3187060  83 dict (no owner)
     6    345   1   151596   4   3338656  86 dict of class
     7    145   0    90956   2   3429612  89 dict of type
     8    192   0    82156   2   3511768  91 type
     9   6309  11    75708   2   3587476  93 int
<140 more rows. Type e.g. '_.more' to view.>
```

We can tell hp to count only newly added objects with:

```
>>> hp.setref()
```

There are still a few objects, but much fewer than before:

```
>>> hp.heap()
Partition of a set of 93 objects. Total size = 8768 bytes.
 Index  Count   %     Size   % Cumulative  % Kind (class / dict of class)
     0      8   9     4024  46       4024  46 types.FrameType
     1      7   8      980  11       5004  57 dict of type
     2     16  17      704   8       5708  65 __builtin__.weakref
     3     20  22      700   8       6408  73 tuple
     4      4   4      560   6       6968  79 dict (no owner)
     5      4   4      560   6       7528  86 dict of guppy.etc.Glue.Interface
     6      9  10      320   4       7848  90 str
     7      7   8      280   3       8128  93 __builtin__.wrapper_descriptor
     8      1   1      140   2       8268  94 dict of guppy.etc.Glue.Owner
     9      4   4      128   1       8396  96 guppy.etc.Glue.Interface
<5 more rows. Type e.g. '_.more' to view.>
```

Now we can create our big list again:

```
>>> big_list = list(range(int(1e6)))
```

The list and the integers in it take up 99 % (74 + 25) of the memory now:

```
>>> hp.heap()
Partition of a set of 1000742 objects. Total size = 16120680 bytes.
 Index  Count   %     Size   % Cumulative  % Kind (class / dict of class)
     0 999908 100 11998896  74   11998896  74 int
     1      3   0  4066700  25   16065596 100 list
     2    750   0    46532   0   16112128 100 str
     3      8   0     4012   0   16116140 100 types.FrameType
     4      7   0      980   0   16117120 100 dict of type
     5     22   0      776   0   16117896 100 tuple
     6     16   0      704   0  16118600 100 __builtin__.weakref
     7      4   0      560   0  16119160 100 dict (no owner)
     8      4   0      560   0  16119720 100 dict of guppy.etc.Glue.Interface
     9      7   0      280   0  16120000 100 __builtin__.wrapper_descriptor
<8 more rows. Type e.g. '_.more' to view.>
```

Even we have an error of 1 % in our example, it is good enough to find out how memory changes when we do certain things.

If we use `setref` several times in a row, we get slightly different results:

```
>>> h = hp.heap()
>>> hp.setref()
>>> h.size
16120804
>>> hp.heap().size
5620
>>> big_list = list(range(int(1e6)))
>>> hp.heap().size
16067724
>>> hp.setref()
>>> hp.heap().size
4824
>>> big_list = list(range(int(1e6)))
>>> hp.heap().size
16066788
>>> hp.setref()
>>> hp.heap().size
4768
```

There is much more information in the heap. Let's have a look:

```
>>> h = hp.heap()
```

We can use the index to extract single lines:

```
>>> h[0]
Partition of a set of 999910 objects. Total size = 11998920 bytes.
 Index  Count   %     Size   % Cumulative  % Kind (class / dict of class)
     0 999910 100 11998920 100  11998920 100 int
```

We can order everything by type:

```
>>> h.bytype
Partition of a set of 1000746 objects. Total size = 16120804 bytes.
 Index  Count   %     Size   % Cumulative  % Type
     0 999910 100 11998920  74  11998920  74 int
     1      3   0  4066700  25  16065620 100 list
     2    750   0    46536   0  16112156 100 str
     3      8   0     4028   0  16116184 100 types.FrameType
     4     17   0     2380   0  16118564 100 dict
     5     24   0      856   0  16119420 100 tuple
     6     16   0      704   0  16120124 100 __builtin__.weakref
     7      7   0      280   0  16120404 100 __builtin__.wrapper_descriptor
     8      4   0      128   0  16120532 100 guppy.etc.Glue.Interface
     9      3   0      120   0  16120652 100 types.MethodType
<3 more rows. Type e.g. '_.more' to view.>
```

Since there are only three more lines to display, we use the method `more` to see all of h content:

```
>>> _.more
 Index  Count   %     Size   % Cumulative  % Type
    10      2   0       72   0  16120724 100 types.InstanceType
    11      1   0       64   0  16120788 100 types.CodeType
    12      1   0       16   0  16120804 100 long
```

We can also order by referrers:

```
>>> h.byrcs
Partition of a set of 1000746 objects. Total size = 16120804 bytes.
 Index  Count   %     Size   % Cumulative  % Referrers by Kind (class / dict of class)
     0 1000648 100 12045316  75  12045316  75 list
     1      3   0  4063336  25  16108652 100 dict of module
     2     27   0     4708   0  16113360 100 <Nothing>
     3      6   0     3472   0  16116832 100 tuple
     4     21   0     1456   0  16118288 100 type
     5      4   0      560   0  16118848 100 guppy.etc.Glue.Interface
     6      3   0      420   0  16119268 100 dict of guppy.etc.Glue.Owner
     7      8   0      352   0  16119620 100 guppy.heapy.heapyc.HeapView
     8      7   0      280   0  16119900 100 dict of type
     9      7   0      256   0  16120156 100 dict (no owner), dict of guppy.etc.Glue.Interf
<9 more rows. Type e.g. '_.more' to view.>
```

Let's look at some examples for how we can use `hpy`. First we write a decorator that tells us how much memory the result of a function uses:

```
# file: memory_size_hpy.py

"""Measure the size of used memory with a decorator.
"""

from __future__ import print_function

import functools                                                  #1

from guppy import hpy                                             #2
```

```
memory = {}                                                        #3


def measure_memory(function):                                      #4
    """Decorator to measure memory size.
    """

    @functools.wraps(function)                                     #5
    def _measure_memory(*args, **kwargs):                          #6
        """This replaces the function that is to be measured.
        """
        measurer = hpy()                                           #7
        measurer.setref()                                          #8
        inital_memory = measurer.heap().size                       #9
        try:
            res = function(*args, **kwargs)                        #10
            return res
        finally:                                                   #11
            memory[function.__name__] = (measurer.heap().size -
                                         inital_memory)
    return _measure_memory                                         #12


if __name__ == '__main__':

    @measure_memory                                                #13
    def make_big(number):
        """Example function that makes a large list.
        """
        return list(range(number))                                 #14

    make_big(int(1e6))                                             #15
    print('used memory', memory)                                   #16
```

First we import `functools` (#1) that will help us to write a nice decorator. Then we import `hpy` (#2) and define a global dictionary (#3) that will hold all values for memory. We define a function that takes a function as argument (#4) and another function inside it that takes a variable number of positional and keyword arguments (#6). This is a typical setup of a decorator that takes no arguments (with arguments we would need a third level). We also decorate this function with `@functools.wraps` (#5) to preserve the docstring and the name of the original function after it is decorated.

Now we call `hpy` (#7) and set the measured memory back (#8). We measure our initially used memory (#9) and call the function with the supplied arguments (#10). We always want to have the size of memory after the call (#11). Finally, we return our internally defined function. Note that we store the result of the called function in `res`. This is necessary to get the memory that is used by the object the function returns. We return our newly created function (#12)

We decorate our function (#13) that just returns a list of size `number` (#14). After we call the function (#15), we can print the used memory (#16).

When we suspect that a function leaks memory, we can use `guppy` to measure the memory growth after a function returned:

```
# file memory._growth_hpy.py

"""Measure the memory growth during a function call.
"""

from __future__ import print_function

from guppy import hpy                                               #1

if sys.version_info.major < 3:
    range = xrange

def check_memory_growth(function, *args, **kwargs):                 #2
    """Measure the memory usage of `function`.
    """
    measurer = hpy()                                                #3
    measurer.setref()                                               #4
    inital_memory = measurer.heap().size                            #5
    function(*args, **kwargs)                                       #6
    return measurer.heap().size - inital_memory                     #7

if __name__ == '__main__':

    def test():
        """Do some tests with different memory usage patterns.
        """

        def make_big(number):                                       #8
            """Function without side effects.

            It cleans up all used memory after it returns.
            """
            return range(number)

        data = []                                                   #9

        def grow(number):
            """Function with side effects on global list.
            """
            for x in range(number):
                data.append(x)                                      #10
        size = int(1e6)
        print('memory make_big:', check_memory_growth(make_big,
                                                      size))    #11
        print('memory grow:', check_memory_growth(grow, size))  #12
    test()
```

After importing `hpy` (#1) we define a helper function that takes the function to be measured, and positional and keyword arguments that will be handed to this function (#2). Now we call `hpy` (3) and set the measured memory back (#4). We measure our initially used memory (#5) and call the function with the supplied arguments (#6). Finally, we return difference in memory size before and after the function call (#7).

We define a function that just returns a list (#8) and thus does not increase memory size after it is finished. The size of the returned list is not measured.

We use a global list as data storage (#9) and define a second function that appends elements to this list (`#10`). Finally, we call our helper function with both functions as arguments (`#11` and #12).

2.7.2 Pympler

Pympler [5] it is a merge of the formerly independent projects asizeof, heapmonitor, and muppy. It works with Python 2 and 3. We can use it very similarly to heapy.

Let's start a new interpreter and make an instance of `pympler.tracker.SummaryTracker`:

```
>>> from pympler import tracker
>>> mem_tracker = tracker.SummaryTracker()
```

We need to call `print_diff()` several times to get to the baseline:

```
>>> mem_tracker.print_diff()
                   types |   # objects |   total size
======================= | =========== | ============
                    list |        1353 |    138.02 KB
                     str |        1345 |     75.99 KB
                     int |         149 |      3.49 KB
                    dict |           2 |      2.05 KB
      wrapper_descriptor |           8 |    640     B
                 weakref |           3 |    264     B
       member_descriptor |           2 |    144     B
       getset_descriptor |           2 |    144     B
   function (store_info) |           1 |    120     B
                    cell |           2 |    112     B
          instancemethod |          -1 |    -80     B
                   tuple |          -1 |   -216     B
>>> mem_tracker.print_diff()
  types |   # objects |   total size
======= | =========== | ============
    str |           2 |     97     B
   list |           1 |     96     B
>>> mem_tracker.print_diff()
  types |   # objects |   total size
======= | =========== | ============
```

Now we create our big list and look at the memory again:

```
>>> big_list = list(range(int(1e6)))
>>> mem_tracker.print_diff()
    types |   # objects |   total size
======= | =========== | ============
    int |      999861 |     22.89 MB
   list |           1 |      7.63 MB
```

Let's look at some examples for how we can use `pympler`. First we write a decorator that tells us how much memory the result of a function uses:

```
# file: memory_size_pympler.py

"""Measure the size of used memory with a decorator.
"""

from __future__ import print_function

import functools                                                 #1
import sys

if sys.version_info.major < 3:
    range = xrange

from pympler import tracker                                      #2

memory = {}                                                      #3


def measure_memory(function):                                    #4
    """Decorator to measure memory size.
    """

    @functools.wraps(function)                                   #5
    def _measure_memory(*args, **kwargs):                        #6
        """This replaces the function that is to be measured.
        """
        measurer = tracker.SummaryTracker()                      #7
        for _ in range(5):                                       #8
            measurer.diff()                                      #9
        try:
            res = function(*args, **kwargs)                      #10
            return res
        finally:                                                 #11
            memory[function.__name__] = (measurer.diff())
    return _measure_memory                                       #12
```

```
if __name__ == '__main__':

    @measure_memory                                                #13
    def make_big(number):
        """Example function that makes a large list.
        """
        return list(range(number))                                 #14

    make_big(int(1e6))                                             #15
    print('used memory', memory)                                   #16
```

First we import `functools` (#1) that will help us to write a nice decorator. Then we import `pympler.tracker` (#2) and define a global dictionary (#3) that will hold all values for memory. We define a function that takes a function as argument (#4) and another function inside it that takes a variable number of positional and keyword arguments (#6). This is a typical setup of a decorator that takes no arguments (with arguments we would need a third level). We also decorate this function with `@functools.wraps` (#5) to preserve the docstring and the name of the original function after it is decorated.

Now we make an instance of our tracker (#7). We use a loop (#8) and call `tracker.diff()` several times (#9). Then we call the function with the supplied arguments (#10). We always want to have the size of memory after the call (#11). Finally, we return our internally defined function. Note that we store the result of the called function in `res`. This is necessary to get the memory that is used by the object the function returns. We return our newly created function (#12)

We decorate our function (#13) that just returns a list of size `number` (#14). After we call the function (#15), we can print the used memory (#16).

When we suspect that a function leaks memory, we can use `pympler` to measure the memory growth after a function returned:

```
# file memory_growth_pympler.py

"""Measure the memory growth during a function call.
"""
from __future__ import print_function

import sys

if sys.version_info.major < 3:
    range = xrange

from pympler import tracker                                       #1


def check_memory_growth(function, *args, **kwargs):               #2
    """Measure the memory usage of `function`.
    """
    measurer = tracker.SummaryTracker()                           #3
    for _ in range(5):                                            #4
        measurer.diff()                                           #5
    function(*args, **kwargs)                                  #6
    return measurer.diff()                                     #7

if __name__ == '__main__':

    def test():
        """Do some tests with different memory usage patterns.
        """

        def make_big(number):                                  #8
            """Function without side effects.

            It cleans up all used memory after it returns.
            """
            return range(number)

        data = []                                              #9

        def grow(number):
            """Function with side effects on global list.
            """
            for x in range(number):
                data.append(x)                                 #10
        size = int(1e6)
        print('memory make_big:', check_memory_growth(make_big,
                                                      size))     #11
        print('memory grow:', check_memory_growth(grow, size))   #12

    test()
```

After importing `pympler.tracker` (#1) we define a helper function that takes the function to be measured, and positional and keyword arguments that will be handed to this function (#2). We make an instance of `tracker.SummaryTracker` (3) and use a loop (#4) to call `measurer.diff()` several times. This way, we set the baseline of memory usage (#5). We call the function with the supplied arguments (#6). Finally, we return the difference in memory size before and after the function call (#7).

We define a function that just returns a list (#8) and thus does not increase memory size after it is finished. The size of the returned list is not measured.

We use a global list as data storage (#9) and define a second function that appends elements to this list (`#10`). Finally, we call our helper function with both functions as arguments (`#11` and #12).

Pympler offers more tools. Let's look at the possibilities to measure the memory size of a given object. We would like to measure the memory size of a list as we append elements. We write a function that takes the length of the list and a function that is to be used to measure the memory size of an object:

```
# file: pympler_list_growth.py

"""Measure the size of a list as it grows.
"""
```

```python
from __future__ import print_function

import sys

if sys.version_info.major < 3:
    range = xrange

from pympler.asizeof import asizeof, flatsize


def list_mem(length, size_func=flatsize):
  """Measure incremental memory increase of a growing list.
  """
  my_list= []
  mem = [size_func(my_list)]
  for elem in range(length):
    my_list.append(elem)
    mem.append(size_func(my_list))
  return mem
```

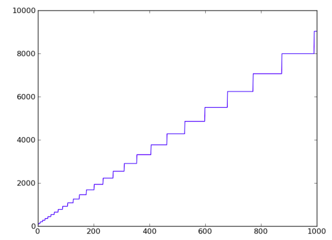
Now we use this function with three different functions: `pympler.asizeof.flatsize`, `pympler.asizeof.asizeof` and `sys.getsizeof`:

```python
if __name__ == '__main__':
  SIZE = 1000
  SHOW = 20

  for func in [flatsize, asizeof, sys.getsizeof]:
    mem = list_mem(SIZE, size_func=func)
    try:
      from matplotlib import pylab
      pylab.plot(mem)
      pylab.show()
    except ImportError:
      print('matplotlib seems not be installed. Skipping the plot.')
      if SIZE > SHOW:
        limit = SHOW // 2
        print(mem[:limit], '... skipping %d elements ...' % (SIZE - SHOW),
              end='')
        print(mem[-limit:])
      else:
        print(mem)
```

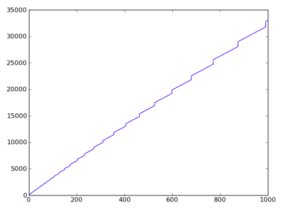
The code just calls our function and supplies one of the functions to measure memory size as an argument. If matloptlib is installed, it draws a graph for each call. Let's look at the resulting graphs.

Using `pympler.asizeof.flatsize` we get this kind of step diagram:



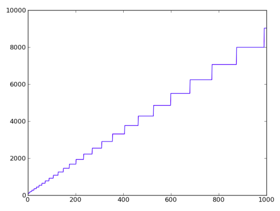
We can see nicely how the list grows discontinuously. Python allocates more memory than it actually needs to append the next element. This way it can append several elements before it needs to increase its size again. These steps get bigger the bigger the list is.

Using `pympler.asizeof.asizeof` we get a different looking graph:



This function also measures the size of all referenced objects. In our case all the integer that are stored in the list. Therefore, there is an continuous increase in memory size between the steps case by the list allocation.

For this simple case `sys.getsizeof` produces the same result as `pympler.asizeof.flatsize`:



For more complex cases `pympler.asizeof.flatsize` might give different results.

We can also measure the number of allocation steps it takes when a list grows one element at a time:

```python
# file: list_alloc_steps.py

"""Measure the number of memory allocation steps for a list.
"""
from __future__ import print_function

import sys

if sys.version_info.major < 3:
    range = xrange
```

```
from pympler.asizeof import flatsize


def list_steps(lenght, size_func=sys.getsizeof):
    """Measure the number of memory alloaction steps for a list.
    """
    my_list = []
    steps = 0
    int_size = size_func(int())
    old_size = size_func(my_list)
    for elem in range(lenght):
        my_list.append(elem)
        new_size = sys.getsizeof(my_list)
        if new_size - old_size > int_size:
            steps += 1
        old_size = new_size
    return steps


if __name__ == '__main__':
    steps = [10, 100, 1000, 10000, int(1e5), int(1e6), int(1e7)]
    print('Using sys.getsizeof:')
    for size in steps:
        print('%10d: %3d' % (size, list_steps(size)))
    print('Using pympler.asizeof.flatsize:')
    for size in steps:
        print('%10d: %3d' % (size, list_steps(size, flatsize)))
```

The results are the same for `sys.getsizeof` and `pympler.asizeof.flatsize`:

```
Using sys.getsizeof:
        10:   3
       100:  10
      1000:  27
     10000:  46
    100000:  65
   1000000:  85
  10000000: 104
Using pympler.asizeof.flatsize:
        10:   3
       100:  10
      1000:  27
     10000:  46
    100000:  65
   1000000:  85
  10000000: 104
```

2.7.3 *Memory Usage Line-by-Line with* `memory_profiler`

Similarly to `line_profiler` that profiles CPU usage line-by-line, `memory_profiler` measures the memory line-by-line. We use a small sample code with one function and decorate it with `@profile`:

```python
# file: use_mem.py

import random
import sys

# Make it work with Python 2 and Python 3.
if sys.version_info.major < 3:
    range = xrange


@profile
def use_mem(numbers):
    """Different ways to use up memory.
    """
    a = sum([x * x for x in numbers])
    b = sum(x * x for x in numbers)
    c = sum(x * x for x in numbers)
    squares = [x * x for x in numbers]
    d = sum(squares)
    del squares
    x = 'a' * int(1e6)
    del x
    return 42


if __name__ == '__main__':

    numbers = [random.random() for x in range(int(1e6))]
    use_mem(numbers)
```

Running it from the command line:

```
$ python -m memory_profiler use_mem.py
```

for a list one million random numbers:

```
Line #    Mem usage    Increment   Line Contents
================================================
     8                             @profile
     9    33.430 MB     0.000 MB   def use_mem(numbers):
    10    94.797 MB    61.367 MB       a = sum([x * x for x in numbers])
    11    94.797 MB     0.000 MB       b = sum(x * x for x in numbers)
    12    94.797 MB     0.000 MB       c = sum(x * x for x in numbers)
    13   114.730 MB    19.934 MB       squares = [x * x for x in numbers]
    14   121.281 MB     6.551 MB       d = sum(squares)
    15   121.281 MB     0.000 MB       del squares
    16   312.020 MB   190.738 MB       x = 'a' * int(2e8)
    17   121.281 MB  -190.738 MB       del x
    18   121.281 MB     0.000 MB       return 42
```

and then for a list ten million random numbers:

```
Line #    Mem usage    Increment   Line Contents
================================================
     8                             @profile
     9   265.121 MB     0.000 MB   def use_mem(numbers):
    10   709.500 MB   444.379 MB       a = sum([x * x for x in numbers])
    11   799.570 MB    90.070 MB       b = sum(x * x for x in numbers)
    12   798.965 MB    -0.605 MB       c = sum(x * x for x in numbers)
    13   806.707 MB     7.742 MB       squares = [x * x for x in numbers]
    14   972.270 MB   165.562 MB       d = sum(squares)
    15   976.984 MB     4.715 MB       del squares
    16   943.906 MB   -33.078 MB       x = 'a' * int(2e8)
    17   871.207 MB   -72.699 MB       del x
    18   871.203 MB    -0.004 MB       return 42
```

The result is not as clear as expected. One reason might bet that it takes time to free memory. Therefore, the effects come later.

In addition to running from the command line you can import the decorator `from memory_profile import profile`. You can also track the memory usage over time. For example, his measures the usage of the interactive Python interpreter:

```
>>> from memory_profiler import memory_usage
>>> mem_over_time = memory_usage(-1, interval=0.5, timeout=3)
>>> mem_over_time
[7.453125, 7.4609375, 7.4609375, 7.4609375, 7.4609375, 7.4609375]
```

You can also supply a PID of another process. `memory_profiler` also comes with a IPython plug-in to be used with the magic function `%memit` analogous to `%timeit`.

2.7.4 Exercise

Measure the memory consumption of these functions `makelist()` and `make_gen()` with Pympler. Measure the memory consumption of `test()` line-by-line with `memory_profiler`. Use increasing numbers for n from 1e4 to 1e8 (or larger if you have enough memory). Explain your findings.

```
#file: make_list_gen.py

from __future__ import print_function

import sys
import time
```

```
if sys.version_info.major < 3:
    range = xrange


def make_list(n):
    return [x * 10 for x in range(n)]


def make_gen(n):
    return (x * 10 for x in range(n))


def test():
    n = int(1e4)  # 1e5, 1e6, 1e7, 1e8
    list_ = make_list(n)
    del list_
    gen = make_gen(n)
    del gen
    time.sleep(1)

test()
```

3 Algorithms and Anti-patterns

3.1 String Concatenation

Strings in Python are immutable. So if you want to modify a string, you have to actually create a new one and use parts of the old one:

```
>>> s = 'old text'
>>> 'new' + s[-5:]
'new text'
```

This means that new memory has to be allocated for the string. This is no problem for a few hundred or thousand strings, but if you have to deal with millions of strings, memory allocation time may be considerably longer. The solution in Python is to use a list to hold the sub strings and join them with `''.join()` string method.

3.1.1 Exercise

Write a test program that constructs a very long string (containing up to one million characters). Use the idiom `s += 'text'` and the idiom `text_list.append('text')` plus `''.join(text_list)` in a function for each. Compare the two approaches in terms of execution speed.

Hint: You can use `timeit.default_timer()` to get the time since the last call to this function. Alternatively, you can use the module `timeit` or the function `measure_run_time` from the module `measure_time` in the directory `measuring`. If you have PyPy install, run the timings with it.

3.2 List and Generator Comprehensions

Python offers list comprehension as a short and very readable way to construct a list.

```
>>> L = [x * x for x in range(10)]
>>> L
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

is a short form for:

```
>>> L = []
>>> for x in range(10):
...     L.append(x * x)
...
>>> L
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

If you are not interested in the list itself but rather some values computed from the whole list, you can use generator comprehension and avoid the list all together.

```
>>> sum(x * x for x in xrange(10))
285
```

3.2.1 Exercise

Write a test program that calculates the sum of all squares of the numbers form zero to one million. Use the idiom `l.append` and list comprehension as well as generator comprehension. Try it with `range` and `xrange`. Use different numbers, e.g. smaller and bigger than one million.

Hint: You can use `timeit.default_timer()` to get the time since the last call to this function. Alternatively, you can use the module `timeit` or the function `measureRunTime` which you can find in the `examples` directory in the subdirectory `modules`.

3.3 Think Global buy Local

A greta deal of things in Python are dynamic. This includes the lookup of variables. It follows the famous LGB local-global-built-in rule. If a variable name is not found in the local scope, Python looks for it in global and then in the built-in name space before raising an `NameError` when nothing was found.

Since every name space is a dictionary, it involves more look ups the more name spaces have to be searched. Therefore, local variables are faster than global variables. Let's look at an example:

```python
# file: local_global.py

"""Local vs. built-in.
"""

import sys

if sys.version_info.major < 3:
    range = xrange

GLOBAL = 1


def repeat(counter):
    """Using the GLOBAL value directly.
    """
    for count in range(counter):
        GLOBAL


def repeat_local(counter):
    """Making GLOBAL a local variable.
    """
    local = GLOBAL
    for count in range(counter):
        local


def test(counter):
    """Call both functions.
    """
    repeat(counter)
    repeat_local(counter)


if __name__ == '__main__':

    def do_profile():
        """Check the run times.
        """
        import cProfile
        profiler = cProfile.Profile()
        profiler.run('test(int(1e8))')
        profiler.print_stats()

    do_profile()
```

By running this code, we will see that the version that accesses the `GLOBAL` directly is about 25% slower than the version with the local variable.

The difference becomes larger when we move more outward and make a built-in name a local one:

```
"""Local vs. built-in.
"""

import sys

if sys.version_info.major < 3:
    range = xrange


def repeat(counter):
    """Using the built-in `True` in a loop.
    """
    for count in range(counter):
        True


def repeat_local(counter):
    """Making `True` a local variable.
    """
    true = True
    for count in range(counter):
        true


def test(counter):
    """Call both functions.
    """
    repeat(counter)
    repeat_local(counter)
```

```
if __name__ == '__main__':

    def do_profile():
        """Check the run times.
        """
        import cProfile
        profiler = cProfile.Profile()
        profiler.run('test(int(1e8))')
        profiler.print_stats()

    do_profile()
```

In this example it saves about 40% of the run. So, if you have large loops and you access globals or built-ins frequently, making them local might be quite useful.

4 The Right Data Structure

4.1 Use built-in Data Types

It is always a good idea to use Python built-in data structures. They are not only most often more elegant and robust than self-made data structures, but also faster in nearly all cases. They are well tested, often partially implemented in C and optimized through long time usage by many of talented programmers.

There are essential differences among built-in data types in terms of performance depending on the task.

4.2 `list` vs. `set`

If you need to search in items, dictionaries and sets are mostly preferable to lists.

```
>>> 9 in range(10)
True
>>> 9 in set(range(10))
True
```

Let's make a performance test. We define a function that searches in a list:

```
>>> import timeit
>>> def search_list(n):
...     my_list = range(n)
...     start = timeit.default_timer()
...     n in my_list
...     return timeit.default_timer() - start
...
```

and one that searches in a set:

```
>>> def search_set(n):
...     my_set = set(range(n))
...     start = timeit.default_timer()
...     n in my_set
...     return timeit.default_timer() - start
...
```

We define a function that compares both run time:

```
>>> def compare(n):
...     print 'ratio:', search_list(n) / search_set(n)
...
```

The set is considerably faster, especially for larger collections:

```
>>> compare(10)
 ratio: 1.83441560587
>>> compare(100)
ratio: 4.4749036373
>>> compare(1000)
ratio: 21.4793493288
>>> compare(10000)
ratio: 203.487480019
>>> compare(100000)
ratio: 1048.8407761
```

We did not measure the time it takes to convert the list into a set. So, let's define a modified function for the set that includes the creation of the set into the runtime measurement:

```
>>> def search_set_convert(n):
...     my_list = range(n)
...     start = timeit.default_timer()
...     my_set = set(my_list)
...     n in my_set
...     return timeit.default_timer() - start
...
```

we need a corresponding compare function:

```
>>> def compare_convert(n):
...     print 'ratio:', search_list(n) / search_set_convert(n)
...
```

Now the set is not faster anymore:

```
>>> compare_convert(10)
ratio: 0.456790136742
>>> compare_convert(100)
ratio: 0.316335542345
>>> compare_convert(1000)
ratio: 0.624656834843
>>> compare_convert(10000)
ratio: 0.405443366236
>>> compare_convert(100000)
ratio: 0.308628738218
>>> compare_convert(1000000)
ratio: 0.295318162219
```

If we need to search more than once, the overhead for creating the set gets relatively smaller. We write function that searches in our list several times:

```
>>> def search_list_multiple(n, m):
...     my_list = range(n)
...     start = timeit.default_timer()
...     for x in xrange(m):
...         n in my_list
...     return timeit.default_timer() - start
```

and do the same for our set:

```
>>> def search_set_multiple_convert(n, m):
...     my_list = range(n)
...     start = timeit.default_timer()
...     my_set = set(my_list)
...     for x in xrange(m):
...         n in my_set
...     return timeit.default_timer() - start
```

We also need a new compare function:

```
>>> def compare_convert_multiple(n, m):
...     print 'ratio:', (search_list_multiple(n, m) /
...     search_set_multiple_convert(n, m))
```

The set gets relatively faster with increasing collection size and number of searches.

```
>>> compare_convert_multiple(10, 1)
ratio: 0.774266745907
>>> compare_convert_multiple(10, 10)
ratio: 1.17802196759
>>> compare_convert_multiple(100, 10)
ratio: 2.99640026716
>>> compare_convert_multiple(100, 100)
ratio: 12.1363117596
>>> compare_convert_multiple(1000, 1000)
ratio: 39.478349851
>>> compare_convert_multiple(10, 1000)
ratio: 180.783828766
>>> compare_convert_multiple(10, 1000)
ratio: 3.81331204005
```

Let's assume we have two lists:

```
>>> list_a = list('abcdefg')
>>> list_a
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> list_b = list('fghijklmnopq')
>>> list_b
['f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q']
```

and we would like to find out which letters are in both lists. A simple implementation would look like this:

```
>>> in_both = []
>>> for a in list_a:
...     if a in list_b:
...         in_both.append(a)
```

```
>>> in_both
['f', 'g']
```

This can be achieved in fewer lines and in most cases faster with sets:

```
>>> set_a = set(list_a)
>>> set_b = set(list_b)
>>> set_a.intersection(set_b)
set(['g', 'f'])
```

Following the same method, we write a short performance test. First we write the function that uses lists:

```
>>> def intersect_list(n):
...     list_a = range(n)
...     list_b = range(n-3, 2 * n)
...     start = timeit.default_timer()
...     in_both = []
...     for a in list_a:
...         if a in list_b:
...             in_both.append(a)
...     run_time = timeit.default_timer() - start
...     return run_time, in_both
...
```

and check if the results is what we expected:

```
>>> intersect_list(10)
(1.0189864042331465e-005, [7, 8, 9])
```

Now, we write a function for sets:

```
>>> def intersect_set(n):
...     set_a = set(range(n))
...     set_b = set(range(n-3, 2 * n))
...     start = timeit.default_timer()
...     in_both = set_a.intersection(set_b)
...     run_time = timeit.default_timer() - start
...     return run_time, in_both
...
```

We are faster but the result of the intersection is the same:

```
>>> intersect_set(10)
(4.0926115616457537e-006, set([8, 9, 7]))
```

Finally, we write a comparison function in which we assert that both results are the same, and calculate the run time ratios.

```
>>> def compare_intersect(n):
...     list_time, list_result = intersect_list(n)
...     set_time, set_result = intersect_set(n)
...     assert set_result == set(list_result)
...     print 'ratio:', list_time / set_time
...
```

Now we can compare both versions with lists and sets:

```
>>> compare_intersect(10)
ratio: 2.75475854866
>>> compare_intersect(100)
ratio: 49.3294012578
>>> compare_intersect(1000)
ratio: 581.103479374
>>> compare_intersect(10000)
ratio: 7447.07128383
```

Note that the problem with the time for constructing the sets is not included here.

4.3 `list` vs. `deque`

For certain tasks we can use a deque instead of a list. A deque is a doubly linked list. This data structure allows faster insertion into the middle part. On the other hand, access of elements by index is slow.

So far we have instrumented our functions we want to test manually with timing code. It is far more elegant to move this timing code into its own, reusable module. In analogy to the decorator we wrote for profiling memory usage, we write one for speed in seconds and kilo stones:

```
# file: profile_speed.py

"""Profile the run time of a function with a decorator.
"""
import functools

import timeit                                                   #1

import pystone_converter                                           #2

speed = {}                                                         #3

def profile_speed(function):                                       #4
    """The decorator.
    """
    @functools.wraps(function)
    def _profile_speed(*args, **kwargs):                           #5
        """This replaces the original function.
        """
        start = timeit.default_timer()                             #6
        try:
            return function(*args, **kwargs)                       #7
        finally:
            # Will be executed *before* the return.
            run_time = timeit.default_timer() - start              #8
                                                                   #9
            kstones = pystone_converter.kpystone_from_seconds(run_time)
            speed[function.__name__] = {'time': run_time,
                                        'kstones': kstones}        #10
    return _profile_speed                                          #11
```

We need the time module (#1) to measure the elapsed time. We also import our converter from seconds to pystones (#2). Again, we use a global dictionary to store our speed profiling results (#3). The decorator function takes function to be speed tested as argument (#4). The nested function takes positional and keyword arguments (#5) that will be supplied to the measured function. We record a time stamp for the start (#6) and call our function with arguments (#7). After this, we calculate the run time (#8) and convert it into kilo pystones (#9). Finally, we store the measured values in the global dictionary (#10) and return our nested function (#11).

Now we can use our module at the interactive prompt:

```
>>> import profile_speed
```

We decorate a function that takes a list and deletes several elements somewhere in the list by assigning an empty list to the range to be deleted:

```
>>> @profile_speed.profile_speed
... def remove_from_list(my_list, start, end):
...     my_list[start:end] = []
...
```

Now we use a deque to do the same:

```
>>> @profile_speed.profile_speed
... def remove_from_deque(my_deque, start, end):
```

```
...     my_deque.rotate(-end)
...     for counter in range(end - start):
...         my_deque.pop()
...     my_deque.rotate(start)
...
```

We rotate by `-end` to move the elements that need to be deleted to the end, call `pop` as many times as needed and rotate back by `start`.

Let's look at this rotating with a small example: We would like to achieve this:

```
>>> L = range(10)
>>> L[2:4] = []
>>> L
[0, 1, 4, 5, 6, 7, 8, 9]
```

We import `deque` from the `collections` module:

```
>>> from collections import deque
```

make make a deque:

```
>>> d = deque(range(10))
>>> d
deque([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Now we rotate by the negative end index:

```
>>> d.rotate(-4)
>>> d
deque([4, 5, 6, 7, 8, 9, 0, 1, 2, 3])
```

We remove the last two elements:

```
>>> d.pop()
3
>>> d.pop()
2
```

and rotate back in the desired order:

```
>>> d.rotate(2)
>>> d
deque([0, 1, 4, 5, 6, 7, 8, 9])
```

Now, let's test the speed of our implementations. We make a large list:

```
>>> my_list = range(int(1e6))
```

We make a decque from our list:

```
>>> my_deque = deque(my_list)
```

Now we call both of our decorated functions:

```
>>> remove_from_list(my_list, 100, 105)
>>> remove_from_deque(my_deque, 100, 105)
```

The speed measuring results are in the global dictionary `speed` in `profile_speed`:

```
>>> profile_speed.speed['remove_from_list']
{'kstones': 0.05940467108987868, 'time': 0.0015446220713783987}
>>> profile_speed.speed['remove_from_deque']
{'kstones': 0.00090945420190496104, 'time': 2.3647349735256284e-005}
```

To be able to compare the results better, we calculate the ratio of both speeds:

```
>>> (profile_speed.speed['remove_from_list']['kstones'] /
...  profile_speed.speed['remove_from_deque']['kstones'])
71.706250305342934
```

Our deque is considerably faster than our list. But now we increase the range that is to be deleted:

```
>>> remove_from_list(my_list, 100, 1000)
>>> remove_from_deque(my_deque, 100, 1000)
```

And get a much smaller gain by using a deque:

```
>>> (profile_speed.speed['remove_from_list']['kstones'] /
...  profile_speed.speed['remove_from_deque']['kstones'])
4.925948467147018
```

We make the range even larger:

```
>>> remove_from_list(my_list, 100, 10000)
>>> remove_from_deque(my_deque, 100, 10000)
```

Our list eventually becomes faster than the deque:

```
>>> (profile_speed.speed['remove_from_list']['kstones'] /
...  profile_speed.speed['remove_from_deque']['kstones'])
0.5219062068409327
```

4.4 `dict` vs. `defaultdict`

Since Python 2.5 there is new `defaultdict` in the module `collections`. This works similarly to the the `defaultdict` method of dictionaries.

Let's assume we want to count how many of each letter are in the following sentence:

```
>>> s = 'Some letters appear several times in this text.'
```

We can do this in the standard way:

```
>>> d = {}
>>> for key in s:
...     d.setdefault(key, 0)
...     d[key] += 1
...
>>> d
{'a': 3, ' ': 7, 'e': 8, 'i': 3, 's': 4, 'm': 2,
 'l': 2, 'o': 1, 'n': 1, 'p': 2, 'S': 1, 'r': 3,
 't': 6, 'v': 1, 'x': 1, 'h': 1, '.': 1}
```

Or we can use the new `defaultdict`:

```
>>> dd = collections.defaultdict(int)
>>> for key in s:
...     dd[key] += 1
...
>>> dd
defaultdict(<type 'int'>, {'a': 3, ' ': 7, 'e': 8, 'i': 3, 's': 4, 'm': 2,
'l': 2, 'o': 1,  'n': 1, 'p': 2, 'S': 1, 'r': 3, 't': 6, 'v': 1, 'x': 1,
'h': 1, '.': 1})
>>>
```

Let's profile the speed differences. First, a function with our standard dictionary:

```
>>> @profile_speed.profile_speed
... def standard_dict(text):
...     d = {}
...     for key in text:
...         d.setdefault(key, 0)
...         d[key] += 1
...
```

And now one for the `defaultdict`:

```
>>> import collections
>>> @profile_speed.profile_speed
... def default_dict(text):
```

```
...     dd = collections.defaultdict(int)
...     for key in text:
...         dd[key] += 1
...
```

We call them both with the same data:

```
>>> standard_dict(s)
>>> default_dict(s)
```

and compare the results:

```
>>> (profile_speed.speed['standard_dict']['kstones'] /
     profile_speed.speed['default_dict']['kstones'])
1.0524903876080238
```

There is not much difference between them: Therefore, we increase the size of our data:

```
>>> s = 'a' * int(1e6)
>>> standard_dict(s)
>>> default_dict(s)
```

and get a more than twofold speedup:

```
>>> (profile_speed.speed['standard_dict']['kstones'] /
     profile_speed.speed['default_dict']['kstones'])
2.3854284818433915
```

Let's look at different example from the Python documentation. We have this data structure:

```
>>> data = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
```

Our goal is to produce a dictionary that groups all second tuple entries into a list:

```
>>> d.items()
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

Again, we define a decorated function for the two dictionary versions:

```
>>> @profile_speed.profile_speed
... def default_dict_group(data):
...     dd = collections.defaultdict(list)
...     for key, value in data:
...         dd[key].append(value)
...
```

```
>>> @profile_speed.profile_speed
... def standard_dict_group(data):
...     d = {}
...     for key, value in data:
...         d.setdefault(key, []).append(value)
...
```

Call them:

```
>>> default_dict_group(data)
>>> standard_dict_group(data)
```

and look at the results:

```
>>> (profile_speed.speed['standard_dict_group']['kstones'] /
     profile_speed.speed['default_dict_group']['kstones'])
0.69018090107868191
```

The `defaultdict` seems to be slower. So let's increase th data size:

```
>>> data = data * 10000
>>> standard_dict_group(data)
>>> default_dict_group(data)
```

Now we are nearly twice as fast:

```
>>> (profile_speed.speed['standard_dict_group']['kstones'] /
     profile_speed.speed['default_dict_group']['kstones'])
1.9115965603608458
```

Making the data even larger makes things only slightly faster:

```
>>> data = data * 10
>>> standard_dict_group(data)
>>> default_dict_group(data)
```

```
>>> (profile_speed.speed['standard_dict_group']['kstones'] /
     profile_speed.speed['default_dict_group']['kstones'])
1.9823501285360818
```

Another increase by a factor of ten actually produces a less favorable ratio for the `defaultdict`:

```
>>> data = data * 10
>>> standard_dict_group(data)
>>> default_dict_group(data)
>>> (profile_speed.speed['standard_dict_group']['kstones'] /
     profile_speed.speed['default_dict_group']['kstones'])
1.8241023044794571
```

4.5 Big-O notation and Data Structures

Normally, you would want to reduce complexity of your program to make it faster. One frequently used measure for complexity is the so called big-O [6] notation. The following table gives an overview of some notations along with a short description and some examples from Python.

Notation	Description	Python Examples
O(1)	constant time does not increase with size of data	len(my_list), len(my_dict), my_list[i], del my_dict[i], x in dict, x in set, my_list.append(i)
O(n)	linear time increase linearly with size of data	Loops on list, strings, dicts, sets, string methods, x in my_list
O(n log n)	quasi linear time increases a little faster than linearly	my_list.sort()
O(n^2)	quadratic time increases four times for each doubling of data	nested loops
O(n^3)	cubic time increases four times for each doubling of data	nested nested loops
O(n^c)	factorial	traveling sales man problem (not Python specific)

In general, using big-O notation we look only at the order of magnitude. Constant factors are neglected. So O(3*n) and O(20*n) are called O(n). Therefore, O(20*n) might be slower than O(n^2) for very small n. But for large n the constant factor has very little influence.

Actually we have already compared several of these notations in our examples above. Let's look at some more comparisons of notations.

4.6 O(1) vs. O(n) vs. O(n^2)

We use our decorator from the module `profile_speed`:

```
>>> import profile_speed
```

We write a function that takes an iterable and reverses it into a list. Our first implementation uses the method `insert` to insert every item at the first position:

```
>>> @profile_speed.profile_speed
... def use_on(iterable):
...     result = []
...     for item in iterable:
...         result.insert(0, item)
...     return result
...
```

Our second implementation uses `append` and reverse the list after all items are appended:

```
>>> @profile_speed.profile_speed
... def use_o1(iterable):
...     result = []
...     for item in iterable:
...         result.append(item)
...     result.reverse()
...     return result
...
```

Now we compare both functions in terms for runtime:

```
>>> def compare_on_o1(n):
...     r1 = use_on(range(n))
...     r2 = use_o1(range(n))
...     assert r1 == r2
...     print (profile_speed.speed['use_on']['kstones'] /
...            profile_speed.speed['use_o1']['kstones'])
...
>>> compare_on_o1(10)
1.6353049525
>>> compare_on_o1(100)
2.01816718953
>>> compare_on_o1(1000)
4.04768995537
>>> compare_on_o1(10000)
27.2673621812
>>> compare_on_o1(100000)
156.635364154
>>> compare_on_o1(int(1e6)) # this might take a while
2355.86619878
```

The speed differences are growing rapidly with increasing data sizes. The method `append` is O(1) and `reverse` is O(n). Even though `insert` is also O(n) it is called n times whereas `reverse` is called only once. Because we loop over all items of our iterable, the first function is O(n + n) but the second is O(n ^{2}). Putting this in numbers, we get:

```
>>> for x in [10, 100, 1000, 10000, 100000, 1000000]:
...     print x * x / (x + x)
...
5
50
500
5000
```

```
50000
500000
```

Of course instead of appending to a new list we can just convert the iterable into a list and reverse it:

```
>>> @profile_speed.profile_speed
... def use_list(iterable):
...     result = list(iterable)
...     result.reverse()
...     return result
...
```

Now we can compare both implementations that have the same big-O notation:

```
>>> def compare_o1_list(n):
...     r1 = use_list(range(n))
...     r2 = use_o1(range(n))
...     assert r1 == r2
...     speed = profile_speed.speed
...     print (speed['use_o1']['kstones'] /
...            speed['use_list']['kstones'])
...
```

```
>>> compare_o1_list(10)
1.24255753768
>>> compare_o1_list(100)
4.39513352799
>>> compare_o1_list(1000)
23.1481811661
>>> compare_o1_list(10000)
54.2245839131
>>> compare_o1_list(100000)
53.132471733
>>> compare_o1_list(1000000)
29.8124806601
```

Even though the big-O notation is the same, the `list` version is up to 50 times faster.

4.7 Exercises

1. Write a test program that searches the last number in a long list. Use `item in long_list` and `item in set(long_list)`. Perform this search 10 and more times. Compare the run times. Hint: You can use `timeit.default_timer()` to get the time since the last call to this function. Alternatively, you can use the module `timeit` or the function `measureRunTime` which you can find in the `examples` directory in the subdirectory `modules`.

5 Caching

5.1 Reuse before You Recalculate

If you find yourself calling the same function with the same arguments many time then caching might help to improve the performance of your program. Instead of doing an expensive calculation, database query, or rendering again and over again, caching just reuses the results of former function calls. Depending on whether the results will be the same for every call to the same function with the same arguments or if the result might change over time, we talk about deterministic or non-deterministic caching. An example for deterministic caching would be numerical calculations that should always produce the same result for the same input. Caching of database queries is non-deterministic because the database content might change. So after some timeout period the query has to be done anew.

All of the following examples are based on [ZIAD2008].

5.2 Deterministic caching

The first thing we need to do, if we want to cache function results, is to uniquely identify the function we want to call:

```python
# file: get_key.py
# based on Ziade 2008

"""Generate a unique key for a function and its arguments.
"""


def get_key(function, *args, **kw):                            #1
    """Make key from module and function names as well as arguments.
    """
    key = '%s.%s:' % (function.__module__,
                      function.__name__)                        #2
    hash_args = [str(arg) for arg in args]                      #3
    hash_kw = ['%s:%s' % (k, str(v))
                  for k, v in kw.items()]                       #4
    return '%s::%s::%s' % (key, hash_args, hash_kw)             #5
```

The function `get_key` takes a function and its positional and keyword arguments (#1). We extract the module name and function name from the function (#2). Now we convert all positional arguments into a list of strings (#3). We convert the keyword arguments into a list of strings using the keys and the string representation of the values (#4). Finally, we return a string that consists of the three strings we have assembled so far (#5).

Now we use our function for a decorator to memoize (a term for the kind of caching we perform) previously calculated results:

```python
# file: cache_deterministic.py
# form Ziade 2008

```

```
"""Example for a deterministic cache
"""

import functools


from get_key import get_key                                         #1

cache = {}                                                          #2


def memoize_deterministic(get_key=get_key, cache=cache):             #3
    """Parameterized decorator for memoizing.
    """

    def _memoize(function):                                         #4
        """This takes the function.
        """

        @functools.wraps(function)
        def __memoize(*args, **kw):                                 #5
            """This replaces the original function.
            """
            key = get_key(function, *args, **kw)                    #6
            try:
                return cache[key]                                   #7
            except KeyError:
                value = function(*args, **kw)                       #8
                cache[key] = value                                  #9
                return value                                        #10
        return __memoize
    return _memoize
```

We use our function `get_key` (#1) and define a global dictionary that will be used to store pre-calculated data (#2). Our decorator takes the function and the dictionary as arguments (#3). This allows us to use other functions to retrieve a key and other caches possibly data dictionary-like data stores such as shelve. The second level function takes the function that is to be called as argument (#4). The third level function takes the arguments (#5). Now we retrieve our key (#6) and try to access the result from our cache (#7). If the key is not in the cache, we call our function /#8), store the result in the cache (#9) and return the result (`#10`).

Let's try how it works. We import the time modul and our module with the decorator:

```
>>> import time
>>> import cache_deterministic
```

We define a new function that adds to numbers and is decorated:

```
>>> @cache_deterministic.memoize_deterministic()
... def add(a, b):
```

```
...     time.sleep(2)
...     return a + b
...
```

We simulate some heavy calculations by delaying everything for two seconds with `sleep`. Let's call function:

```
>>> add(2, 2)
4
```

This took about two seconds. Do it again:

```
>>> add(2, 2)
4
```

Now the return is immediate.

Again:

```
>>> add(3, 3)
6
```

Two seconds delay. But now:

```
>>> add(3, 3)
```

Instantaneous response.

5.3 Non-deterministic caching

For non-deterministic caching, we use an age that the computed value should not exceed:

```
# file: cache_non_deterministic.py
# form Ziade 2008

"""Example for a cache that expires.
"""

import functools
import time

from get_key import get_key

cache = {}

def memoize_non_deterministic(get_key=get_key, storage=cache,
                              age=0):                            #1
    """Parameterized decorator that takes an expiration age.
    """

    def _memoize(function):
        """This takes the function.
        """

        @functools.wraps(function)
        def __memoize(*args, **kw):
            """This replaces the original function.
            """
            key = get_key(function, *args, **kw)
            try:
                value_age, value = storage[key]                        #2
                deprecated = (age != 0 and
                             (value_age + age) < time.time())   #3
            except KeyError:
                deprecated = True                                      #4
            if not deprecated:
                return value                                           #5
            storage[key] = time.time(), function(*args, **kw)   #6
            return storage[key][1]                                     #7
        return __memoize
    return _memoize
```

This decorator is a variation of the deterministic one above. We can supply an age (#1). The value will be recalculated if this age is exceeded. We retrieve an age and a value from our cache (#2). The value will be deprecated, i.e. recalculated if we provide a non-zero age and the old age plus the specified age are smaller than the current time (#3). Note: This means, if you provide no age or an age of zero, the cache will never expire. The value will also be calculated if the key was not found (#4). We return the value if it is still valid (#5). Otherwise, we recalculate it and store it together with current time in the cache (#6) and return the freshly calculated value (#7).

Let's see how this works. We import our non-deterministic cache:

```
>>> import cache_non_deterministic
```

and define a new function with a maximum cache age of 5 seconds:

```
>>> @cache_non_deterministic.memoize_non_deterministic(age=5)
... def add2(a, b):
...   time.sleep(2)
...   return a + b
...
```

The first call takes about two seconds:

```
>>> add2(2, 2)
4
```

Immediately after this we do it again and get the response without delay:

```
>>> add2(2, 2)
4
```

Now we wait for at least 5 seconds ... and do it again:

```
>>> add2(2, 2)
4
```

This took again two seconds because the cache was not used and the value was recalculated due to the time since the last call being greater than five seconds.

5.4 Least Recently Used Cache

Caching is a common task. Therefore, Python starting from 3.2, provides a least recently used cache implementation in the `functools` module. This implementation is more elaborate than our attempts here. For example, it is thread-safe. There is a backport for Python 2.6 and higher. You can install it with:

```
pip install backports.functools_lru_cache
```

The size of the cache is determined by `maxsize`. There are three distinct case:

1. If `maxsize` is `0`, there is no caching and the function result will calculated every time the function is called.
2. If `maxsize` is `None`, it works similarly to our deterministic cache. That is, the LRU functionality is disabled and the cache can grow without limits.
3. If `maxize` is a positive integer, the most recent `masxize` function results are cached. This is the most interesting case because we actually use the LRU features.

Let's try it with a very small cache size of `2` to quickly see the effect of a filled cache:

```
import time
@lru_cache(maxsize=2)
def add(a, b):
    time.sleep(2) # wait two seconds
    return a + b
```

Now we use it:

```
>>> add(2, 2)
# takes two seconds
```

The second time around it returns without the two-second delay:

```
>>> add(2, 2)
# returns immediately
```

Now we call the function with two other combinations of arguments:

```
>>> add(10, 5)
# takes two seconds
>>> add(10, 20)
# takes two seconds
```

Now our small cache of size 2 is filled with two other results and a call with our original combination will delay again for the first call:

```
>>> add2(2, 2)
# takes two seconds
>>> add2(2, 2)
# returns immediately
```

We can look at the original function:

```
 >>> add.__wrapped__
<function __main__.add>
```

as well as at the cache:

```
>>> add.cache_info()
CacheInfo(hits=6, misses=3, maxsize=2, currsize=2)
```

Clearing the cache sets everything back to zero:

```
>>> add.cache_clear()
```

```
>>> add.cache_info()
CacheInfo(hits=0, misses=0, maxsize=2, currsize=0)
```

It is recommended to use `functools.lru_cache` over own solutions wherever possible. It is likely to be more stable. For example, coming up with your own solution that is thread-safe might be not as simple as it seems at first glance.

5.5 Memcached

Memcached [7] is a caching server that is primarily used to speed up database based dynamic web pages. It is very fast, trades RAM for speed, and is very powerful. We don't have time to look at it here. There are several ways to use Memcached from Python. Also the probably most popular web framework Django uses Memcached (Djangos cache [8]).

6 Compilation of Tools for Speedup and Extending

There are many more ways to extend Python. Therefore, a short compilation of methods and tools for this purpose is given here. The compilation is by no means exhaustive.

Method/Tool	Remarks	Link
algorithmic improvements	try this first	http://www.python.org
NumPy	matlab like array processing	http://numpy.scipy.org
PyPy	fast Python implementation	http://pypy.org/
Cython	C with Python syntax	http://cython.org/
ctypes	call DLLs directly	Pythons's standard library
cffi	new interface to C for CPython and PyPy	https://cffi.readthedocs.org
Numba	compile to LLVM code	http://numba.pydata.org/
f2py	stable Fortran extensions	http://cens.ioc.ee/projects/f2py2e
C extensions by hand	lots of work	http://docs.python.org/ext/ext.html
SWIG	mature, stable, widely used	http://www.swig.org
Boost.Python	C ++ template based, elegant	http://www.boost.org/libs/python/doc
SIP	developed for Qt, fast	http://www.riverbankcomputing.co.uk/sip
PyInline	inline other languages (alpha)	http://pyinline.sourceforge.net
Theano	mathematical expressions (GPU)	http://deeplearning.net/software/theano/
PyCUDA	Python on GPGPUs	http://mathema.tician.de/software/pycuda
PyOpenCL	Python on GPGPUs	http://mathema.tician.de/software/pyopencl
Copperhead	Python on GPGPUs	http://code.google.com/p/copperhead/
COM/DCOM, CORBA, XML-RPC, ILU	middleware	various
Weave	inline C++ in Python	http://scipy.org/Weave
Babel	unite C/C++, F77/90, Py, Java	http://www.llnl.gov/CASC/components
pyufora	compiled, automatically parallel Python for data-science	http://docs.pyfora.com/en/stable/

7 End

7.1 Colophon

This material is written in `reStructuredText` and has been converted into PDF using `rst2pdf`. All Python files are dynamically included from the development directory.

7.2 Links

LANG2006	Hans Petter Lantangen, Python Scripting for Computational Science, Second Edition, Springer Berlin, Heidelberg, 2006.
MART2005	Alex Martelli et al., Python Cookbook, O'Reilly,2nd Edition, 2005.
MART2006	Alex Martelli, Python in a Nutshell, O'Reilly, 2nd Edition, 2006.
ZIAD2008	Tarek Ziadè, Expert Python Programming: Best practices for designing, coding, and distributing your Python software, Packt 2008.
1	http://aspn.activestate.com/ASPN/Python/Cookbook
2	http://www.vrplumber.com/programming/runsnakerun/
3	http://jiffyclub.github.io/snakeviz/
4	http://guppy-pe.sourceforge.net
5	http://packages.python.org/Pympler/
6	http://en.wikipedia.org/wiki/Big_O_notation
7	http://www.danga.com/memcached/
8	http://docs.djangoproject.com/en/dev/topics/cache/