Profiling and Timing Code

性能测算和计时

> In the process of developing code and creating data processing pipelines, there are often trade-offs you can make between various implementations. Early in developing your algorithm, it can be counterproductive to worry about such things. As Donald Knuth famously quipped, "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."

在开发阶段以及创建数据处理任务流时，经常都会出现多种可能的实现方案，每种都有各自优缺点，你需要在这之中进行权衡。在开发你的算法的早期阶段，过于关注性能很可能会影响你的实现效率。正如高德纳（译者注：Donald Knuth，《计算机程序设计艺术》作者，最年轻的ACM图灵奖获得者，计算机算法泰山北斗）的名言：“我们应该忘掉那些小的效率问题，在绝大部分情况下：过早的优化是所有罪恶之源。”

> But once you have your code working, it can be useful to dig into its efficiency a bit. Sometimes it's useful to check the execution time of a given command or set of commands; other times it's useful to dig into a multiline process and determine where the bottleneck lies in some complicated series of operations. IPython provides access to a wide array of functionality for this kind of timing and profiling of code. Here we'll discuss the following IPython magic commands:
>
> - `%time`: Time the execution of a single statement
> - `%timeit`: Time repeated execution of a single statement for more accuracy
> - `%prun`: Run code with the profiler
> - `%lprun`: Run code with the line-by-line profiler
> - `%memit`: Measure the memory use of a single statement
> - `%mprun`: Run code with the line-by-line memory profiler

但是，一旦你的代码已经开始工作了，那么你就应该开始深入的考虑一下性能问题了。有时你会需要检查一行代码或者一系列代码的执行时间；有时你又需要对多个线程进行研究，找到一系列复杂操作当中的瓶颈所在。IPython提供了这类计时或性能测算的丰富功能。本章节中我们会讨论下述的IPython魔术指令：

- `%time`: 测量单条语句的执行时间
- `%timeit`: 对单条语句进行多次重复执行，并测量平均执行时间，以获得更加准确的结果
- `%prun`: 执行代码，并使用性能测算工具进行测算
- `%lprun`: 执行代码，并使用单条语句性能测算工具进行测算
- `%memit`: 测量单条语句的内存占用情况
- `%mprun`: 执行代码，并使用单条语句内存测算工具进行测算

> The last four commands are not bundled with IPython–you'll need to get the `line_profiler` and `memory_profiler` extensions, which we will discuss in the following sections.

后面四个指令并不是随着IPython一起安装的，你需要去获取安装 `line_profiler` 和 `memory_profiler` 扩展，我们会在下面小节中介绍。

Timing Code Snippets: `%timeit` and `%time`

代码计时工具：`%timeit` 和 `%time`

> We saw the `%timeit` line-magic and `%%timeit` cell-magic in the introduction to magic functions in [IPython Magic Commands]; it can be used to time the repeated execution of snippets of code:

我们在[IPython魔术命令]中已经介绍过 `%timeit` 行魔术指令和 `%%timeit` 块魔术指令；它们用来对于代码（块）进行重复执行，并测量执行时间：

In [1]:
```
%timeit sum(range(100))
```
```
737 ns ± 26.5 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

> Note that because this operation is so fast, `%timeit` automatically does a large number of repetitions. For slower commands, `%timeit` will automatically adjust and perform fewer repetitions:

这里说明一下，因为这个操作是非常快速的，因此 `%timeit` 自动做了很多次的重复执行。如果换成一个执行慢的操作，`%timeit` 会自动调整（减少）重复次数。

In [2]:
```
%%timeit
total = 0
for i in range(1000):
    for j in range(1000):
        total += i * (-1) ** j
```
```
238 ms ± 1.76 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

> Sometimes repeating an operation is not the best option. For example, if we have a list that we'd like to sort, we might be misled by a repeated operation. Sorting a pre-sorted list is much faster than sorting an unsorted list, so the repetition will skew the result:

值得注意的是，有些情况下，重复多次执行反而会得出一个错误的测量数据。例如，我们有一个列表，希望对它进行排序，重复执行的结果会明显的误导我们。因为对一个已经排好序的列表执行排序是非常快的，因此在第一次执行完成之后，后面重复进行排序的测量数据都是错误的：

In [3]:
```
import random
L = [random.random() for i in range(100000)]
%timeit L.sort()
```
```
766 μs ± 182 μs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

> For this, the `%time` magic function may be a better choice. It also is a good choice for longer-running commands, when short, system-related delays are unlikely to affect the result. Let's time the sorting of an unsorted and a presorted list:

在这种情况下，`%time` 魔术指令可能会是一个更好的选择。对于一个执行时间较长的操作来说，它也更加适用，因为与系统相关的那些持续时间很短的延迟将不会对结果产生什么影响。让我们对一个未排序和一个已排序的列表进行排序，并观察执行时间：

In [4]:
```
import random
L = [random.random() for i in range(100000)]
print("sorting an unsorted list:")
%time L.sort()
```
```
sorting an unsorted list:
CPU times: user 29.7 ms, sys: 9 μs, total: 29.7 ms
Wall time: 29.5 ms
```

In [5]:
```
print("sorting an already sorted list:")
%time L.sort()
```
```
sorting an already sorted list:
CPU times: user 4 ms, sys: 0 ns, total: 4 ms
Wall time: 4.01 ms
```

> Notice how much faster the presorted list is to sort, but notice also how much longer the timing takes with `%time` versus `%timeit`, even for the presorted list! This is a result of the fact that `%timeit` does some clever things under the hood to prevent system calls from interfering with the timing. For example, it prevents cleanup of unused Python objects (known as *garbage collection*) which might otherwise affect the timing. For this reason, `%timeit` results are usually noticeably faster than `%time` results.

你应该首先注意到的是对于未排序的列表和对于已排序的列表进行排序的执行时间差别（译者注：在我的笔记本上，接近5倍的时间）。而且你还需要了解 `%time` 和 `%timeit` 执行的区别，即使都是使用已经排好序的列表的情况下。这是因为 `%timeit` 会使用一种额外的机制来防止系统调用影响计时的结果。例如，它会阻止Python解析器清理不再使用的对象（也被称为*垃圾收集*），否则垃圾收集会影响计时的结果。因此，我们认为通常情况下 `%timeit` 的结果都会比 `%time` 的结果要快。

> For `%time` as with `%timeit`, using the double-percent-sign cell magic syntax allows timing of multiline scripts:

对于 `%time` 和 `%timeit` 指令，使用两个百分号可以对一段代码进行计时：

In [6]:
```
%%time
total = 0
for i in range(1000):
    for j in range(1000):
        total += i * (-1) ** j
```
```
CPU times: user 334 ms, sys: 0 ns, total: 334 ms
Wall time: 333 ms
```

> For more information on `%time` and `%timeit`, as well as their available options, use the IPython help functionality (i.e., type `%time?` at the IPython prompt).

更多关于 `%time` 和 `%timeit` 的资料，包括它们的选项，可以使用IPython的帮助功能（如在IPython提示符下键入 `%time?`）进行查看。

Profiling Full Scripts: `%prun`

脚本代码块性能测算：`%prun`

> A program is made of many single statements, and sometimes timing these statements in context is more important than timing them on their own. Python contains a built-in code profiler (which you can read about in the Python documentation), but IPython offers a much more convenient way to use this profiler, in the form of the magic function `%prun`.

一个程序都是有很多条代码组成的，有的时候对整段代码块性能进行测算比对每条代码进行计时要更加重要。Python自带一个内建的代码性能测算工具（你可以在Python文档中找到它），而IPython提供了一个更加简便的方式来使用这个测算工具，使用 `%prun` 魔术指令。

> By way of example, we'll define a simple function that does some calculations:

我们定义一个简单的函数作为例子：

In [7]:
```
def sum_of_lists(N):
    total = 0
    for i in range(5):
        L = [j ^ (j >> i) for j in range(N)]
        total += sum(L)
    return total
```

> Now we can call `%prun` with a function call to see the profiled results:

然后我们就可以使用 `%prun` 来调用这个函数，并查看测算的结果：

In [8]:
```
%prun sum_of_lists(1000000)
```

> In the notebook, the output is printed to the pager, and looks something like this:

```
14 function calls in 0.714 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        5    0.599    0.120    0.599    0.120 <ipython-input-19>:4(<listcomp>)
        5    0.064    0.013    0.064    0.013 {built-in method sum}
        1    0.036    0.036    0.699    0.699 <ipython-input-19>:1(sum_of_lists)
        1    0.014    0.014    0.714    0.714 <string>:1(<module>)
        1    0.000    0.000    0.714    0.714 {built-in method exec}
```

在译者的笔记本上，这个指令的结果输出如下：

```
14 function calls in 0.500 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        5    0.440    0.088    0.440    0.088 <ipython-input-8-f105717832a2>:4(<listcomp>)
        5    0.027    0.005    0.027    0.005 {built-in method builtins.sum}
        1    0.025    0.025    0.492    0.492 <ipython-input-8-f105717832a2>:1(sum_of_lists)
        1    0.008    0.008    0.500    0.500 <string>:1(<module>)
        1    0.000    0.000    0.500    0.500 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

> The result is a table that indicates, in order of total time on each function call, where the execution is spending the most time. In this case, the bulk of execution time is in the list comprehension inside `sum_of_lists`. From here, we could start thinking about what changes we might make to improve the performance in the algorithm.

这个结果的表格，使用的是每个函数调用执行总时间进行排序（从大到小）。从上面的结果可以看出，绝大部分的执行时间都发生在函数 `sum_of_lists` 中的列表解析之上。然后，我们就可以知道如果需要优化这段代码的性能，可以从哪个方面开始着手了。

> For more information on `%prun`, as well as its available options, use the IPython help functionality (i.e., type `%prun?` at the IPython prompt).

更多关于 `%prun` 的资料，包括它的选项，可以使用IPython的帮助功能（在IPython提示符下键入 `%prun?`）进行查看。

Line-By-Line Profiling with `%lprun`

使用`%lprun`对单条代码执行性能进行测算

> The function-by-function profiling of `%prun` is useful, but sometimes it's more convenient to have a line-by-line profile report. This is not built into Python or IPython, but there is a `line_profiler` package available for installation that can do this. Start by using Python's packaging tool, `pip`, to install the `line_profiler` package:

刚才介绍的对于整个函数进行测算的 `%prun` 很有用，但是有时能对单条代码进行性能测算会更加方便我们调优。这个功能不是内置在Python或者IPython里的，你需要安装一个第三方包 `line_profiler` 来完成这项任务。使用Python包管理工具 `pip` 可以很容易地安装 `line_profiler` 包：

```
$ pip install line_profiler
```

> Next, you can use IPython to load the `line_profiler` IPython extension, offered as part of this package:

然后，你可以使用IPython来载入 `line_profiler` 扩展模块：

In [9]:
```
%load_ext line_profiler
```

> Now the `%lprun` command will do a line-by-line profiling of any function–in this case, we need to tell it explicitly which functions we're interested in profiling:

然后 `%lprun` 魔术指令就可以对任何函数进行单行的性能测算了，我们需要明确指出要对哪个函数进行性能测算：

In [10]:
```
%lprun -f sum_of_lists sum_of_lists(5000)
```

> As before, the notebook sends the result to the pager, but it looks something like this:

```
Timer unit: 1e-06 s

Total time: 0.009382 s
File: <ipython-input-19-fa2be176cc3e>
Function: sum_of_lists at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def sum_of_lists(N):
     2         1            2      2.0      0.0      total = 0
     3         6            8      1.3      0.1      for i in range(5):
     4         5         9001   1800.2     95.9          L = [j ^ (j >> i) for j in range(N)]
     5         5          371     74.2      4.0          total += sum(L)
     6         1            0      0.0      0.0      return total
```

像刚才一样，notebook会在一个弹出页面中展示结果，在译者的笔记本上执行效果如下：

```
Timer unit: 1e-06 s

Total time: 0.007372 s
File: <ipython-input-7-f105717832a2>
Function: sum_of_lists at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def sum_of_lists(N):
     2         1          2.0      2.0      0.0      total = 0
     3         6          9.0      1.5      0.1      for i in range(5):
     4         5       7114.0   1422.8     96.5          L = [j ^ (j >> i) for j in range(N)]
     5         5        246.0     49.2      3.3          total += sum(L)
     6         1          1.0      1.0      0.0      return total
```

> The information at the top gives us the key to reading the results: the time is reported in microseconds and we can see where the program is spending the most time. At this point, we may be able to use this information to modify aspects of the script and make it perform better for our desired use case.

结果第一行给我们提供了下面表中的时间单位：微秒，我们可以从中看到函数中哪一行执行花了最多时间。然后，我们就可以根据这些信息对我们的代码进行调优，以达到我们需要的性能指标。

> For more information on `%lprun`, as well as its available options, use the IPython help functionality (i.e., type `%lprun?` at the IPython prompt).

更多关于 `%lprun` 的资料，包括它的选项，可以使用IPython的帮助功能（在IPython提示符下键入 `%lprun?`）进行查看。

Profiling Memory Use: `%memit` and `%mprun`

测算内存使用：`%memit` 和 `%mprun`

> Another aspect of profiling is the amount of memory an operation uses. This can be evaluated with another IPython extension, the `memory_profiler`. As with the `line_profiler`, we start by `pip`-installing the extension:

对于性能测算来说，还有一个方面需要我们注意的是操作使用的内存大小。这需要用到另外一个IPython的扩展模块 `memory_profiler`。就像 `line_profiler` 那样，我们可以使用 `pip` 安装这个扩展模块：

```
$ pip install memory_profiler
```

> Then we can use IPython to load the extension:

然后将扩展模块加载到IPython中：

In [11]:
```
%load_ext memory_profiler
```

> The memory profiler extension contains two useful magic functions: the `%memit` magic (which offers a memory-measuring equivalent of `%timeit`) and the `%mprun` function (which offers a memory-measuring equivalent of `%lprun`). The `%memit` function can be used rather simply:

内存性能测算工具 `memory_profiler` 包括两个有用的魔术指令：`%memit`（提供了与 `%timeit` 等同的内存测算功能）和 `%mprun`（提供了与 `%lprun` 等同的内存测算功能）。`%memit` 的用法非常简单：

In [12]:
```
%memit sum_of_lists(1000000)
```
```
peak memory: 125.05 MiB, increment: 72.98 MiB
```

> We see that this function uses about 100 MB of memory.

我们可以看到这个函数使用了约100MB的内存。

> For a line-by-line description of memory use, we can use the `%mprun` magic. Unfortunately, this magic works only for functions defined in separate modules rather than the notebook itself, so we'll start by using the `%%file` magic to create a simple module called `mprun_demo.py`, which contains our `sum_of_lists` function, with one addition that will make our memory profiling results more clear:

对于单行代码的内存使用测算，我们可以使用 `%mprun` 魔术指令。不幸的是，这个魔术指令只能应用在独立模块里面的函数上，而不能应用在notebook本身。因此我们需要使用 `%%file` 魔术指令来创建一个简单的模块，模块的名称为 `mprun_demo.py`，该模块定义了前面的 `sum_of_lists` 函数，在这个例子中，我们加了一行代码，来让我们的内存测算结果更加的明显：

In [13]:
```
%%file mprun_demo.py
def sum_of_lists(N):
    total = 0
    for i in range(5):
        L = [j ^ (j >> i) for j in range(N)]
        total += sum(L)
        del L # 将列表L的引用删除
    return total
```
```
Overwriting mprun_demo.py
```

> We can now import the new version of this function and run the memory line profiler:

下面我们可以载入这个模块，然后使用内存测算工具对改写后的函数进行单条代码的内存性能测算：

In [14]:
```
from mprun_demo import sum_of_lists
%mprun -f sum_of_lists sum_of_lists(1000000)
```

> The result, printed to the pager, gives us a summary of the memory use of the function, and looks something like this:

在弹出页面中展示的结果给我们大概描述了函数中每行代码内存的使用情况，在译者笔记本上结果如下：

```
Filename: ./mprun_demo.py

Line #    Mem usage    Increment   Line Contents
================================================
     4     71.9 MiB      0.0 MiB           L = [j ^ (j >> i) for j in range(N)]


Filename: ./mprun_demo.py

Line #    Mem usage    Increment   Line Contents
================================================
     1     39.0 MiB      0.0 MiB   def sum_of_lists(N):
     2     39.0 MiB      0.0 MiB       total = 0
     3     46.5 MiB      7.5 MiB       for i in range(5):
     4     71.9 MiB     25.4 MiB           L = [j ^ (j >> i) for j in range(N)]
     5     71.9 MiB      0.0 MiB           total += sum(L)
     6     46.5 MiB    -25.4 MiB           del L # remove reference to L
     7     39.1 MiB     -7.4 MiB       return total
```

> Here the `Increment` column tells us how much each line affects the total memory budget: observe that when we create and delete the list `L`, we are adding about 25 MB of memory usage. This is on top of the background memory usage from the Python interpreter itself.

这里的 `Increment` 列告诉我们函数的每一行怎样影响到了总内存的使用量：观察一下当我们使用列表解析创建 `L` 和使用 `del` 删除 `L` 时发生的情况，这里会有大约25MB内存的使用变化。这是在Python解析器本身占用的基本内存基础上我们函数使用到的内存用量。

> For more information on `%memit` and `%mprun`, as well as their available options, use the IPython help functionality (i.e., type `%memit?` at the IPython prompt).

更多关于 `%memit` 和 `mprun` 的资料，包括它们的选项，可以使用IPython的帮助功能（在IPython提示符下键入 `%memit?` 或 `%mprun?`）进行查看。