Profiling Python

Famously, premature optimization has been called the source of all evil, at least when it comes to computer programming. The key part of this quote is the word premature. In order to optimize your code, you need to know what parts need optimization. This is where profiling comes in. With profiling, you can figure out just how much time is being used in various parts of your code, as well as how much memory is being used. With this information, you will be armed to figure what where to focus your energy to get the largest improvement. In this workshop, I will be assuming that you have some Python experience and ready to try and get the most out the code you have written.

Time Profiling

The first step in getting better, faster code is to do some profiling to see where your time is being spent. The three broad categories of profiling are

- course time measurements
- function profiling
- line profiling

Course Time Measurements

The most basic form of profiling is just seeing how long different chunks of code take to run. You can do this most simply by measuring the start and end times and finding the difference.

In [1]:
```
import time
start = time.clock()
for a in range(1000000):
    b = a ^ a
end = time.clock()
print(end-start)
```

```
0.37612984851227454
```

A more organized way would be to have a timer object that you can easily reuse rather than sticking a number of start-end time commands. An example class might look like the code below (which was blatantly borrowed from the website https://www.huyng.com/posts/python-performance-analysis (https://www.huyng.com/posts/python-performance-analysis)

In [2]:
```
class Timer(object):
    def __init__(self, verbose=False):
        self.verbose = verbose

    def __enter__(self):
        self.start = time.time()
        return self

    def __exit__(self, *args):
        self.end = time.time()
        self.secs = self.end - self.start
        self.msecs = self.secs * 1000  # millisecs
        if self.verbose:
            print('elapsed time: %f ms' % self.msecs)
```

You can then use it with the 'with' statement to time chunks of code.

In [3]:
```
with Timer() as t:
    for a in range(1000000):
        b = a^a
print("=> elasped 1 million: %s s" % t.secs)

with Timer() as t:
    for a in range(10000000):
        b = a^a
print("=> elasped 10 million: %s s" % t.secs)
```

```
=> elasped 1 million: 0.3885676860809326 s
=> elasped 10 million: 3.661386489868164 s
```

If you use the IPython shell, there are special functions called magics that let you interact with the Python interpreter. Two of these magics are %timeit and %%timeit. The first one times how long a single statement takes to run.

In [4]:
```
%timeit range(1000000)
```

```
The slowest run took 5.22 times longer than the fastest. This could mea
n that an intermediate result is being cached.
1000000 loops, best of 3: 1.09 µs per loop
```

In [5]:
```
%%timeit
a = range(1000000)
```

```
1000000 loops, best of 3: 1.1 µs per loop
```

Function Profiling

While this might be good enough for simple code, it quickly becomes too coarse to do any real analysis with. Luckily, Python includes two profiling modules as part of a standard installation. The first a pure Python module named 'profile'. Because it is pure Python, it introduces quite a lot overhead. It is ideal if you want to alter the behavior of the profiler, however. The second profiler is named 'cProfile', which is a C implementation of the same behavior as 'profile'. It is therefore much faster.

The easiest way to use it is to use the run method. We will start by creating our load function to make the code easier to read.

In [6]:
```
def myload(num):
    for a in range(num):
        b = a^a
```

Now we can profile it and see what happens.

In [7]:
```
import cProfile

cProfile.run('myload(1000000)')
```

```
         4 function calls in 0.242 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.242    0.242    0.242    0.242 <ipython-input-6-421e6c04
9a13>:1(myload)
        1    0.000    0.000    0.242    0.242 <string>:1(<module>)
        1    0.000    0.000    0.242    0.242 {built-in method builtin
s.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_ls
prof.Profiler' objects}
```

As you can see, cProfile gives a breakdown of each function call, how many times it gets called, and various measures of the amount of time used. This is a good starting point for larger programs. The major issue with the run method is that you can only use a string that can be run with the exec function. You can also use cProfile directly (like the timer class above) to collect stats for sections of code.

Line Profiler

If you need more detail, you can use another module named line_profiler. This one usually is not included in most Python installations. You will likely need to install it with

- pip install line_profiler

It comes with a command line utility called kernprof that you can use to profile your scripts. You need to add a decorator to any functions you want to profile like

In [8]:
```
@profile
myload(1000000)
```

```
  File "<ipython-input-8-cd3c5e2218d4>", line 2
    myload(1000000)
         ^
SyntaxError: invalid syntax
```

Then run it with

- kernprof -l myscript.py

This generates a binary output file with the profiling information. You can process the results with

- python -m line_profiler myscript.py.lprof

If you want to use line_profiler directly within your script, you can import the LineProfiler class and use it to profile statements in your script.

In [8]:
```
import line_profiler

profiler = line_profiler.LineProfiler()
profiler.add_function(myload(1000000))
profiler.print_stats()
```

```
Timer unit: 4.72616e-07 s


C:\Users\berna_000\Anaconda3\lib\site-packages\ipykernel\__main__.py:4:
 UserWarning: Could not extract a code object for the object None
```

The above warning is because we are running this through Jupyter. If you were to run this as a regular script in IPython, it won't occur. It will also give you the line-by-line profiling information that looks like

File: test.py

Function: get_number at line 43

Total time: 4.44195 s

Line # Hits Time Per Hit % Time Line Contents

```
43                                           def get_number():
44   5000001      2223313      0.4     50.1      for x in xrange(5000000):
45   5000000      2218638      0.4     49.9          yield x
```

You can also use it in a fashion similar to cProfile, using the run() method.

```
In [9]: profiler.run('myload(1000000)')
        profiler.print_stats()
```

```
Timer unit: 4.72616e-07 s
```

Memory Profiling

The other resource that you may need to optimize for is memory. Similarly to line_profilr is a module named memory_profiler. You can import is with

- pip install memory_profiler

The most basic way to use it is again similar to line_profiler. It includes a decorator named @profile which you can add to your script to profile the functions in question. You can then run it with

- python -m memory_profiler myscript.py

Similarly to line_profiler, memory_profiler includes a command line utility called mprof. You can run it with

- mprof run myscript.py
- mprof list

It can even generate a plot of memory usage with

- mprof plot

In an interactive way, you can use memory_profiler directly in a few different ways. The easiest is similar to the coarse timing examples above.

In [10]:
```
import memory_profiler

usage = memory_profiler.memory_usage()
print(usage)
b = list()
for a in range(10000000):
    b.append(a)
usage = memory_profiler.memory_usage()
print(usage)
```

```
[34.484375]
[420.90234375]
```

You can analyze the memory usage for a given function using the same function.

In [11]:
```
def myload2(a):
    b = list()
    for c in range(a):
        b.append(c)

usage = memory_profiler.memory_usage((myload2,(1000000,)))
print(usage)
```

```
[420.9453125, 421.01953125, 436.859375, 447.46875, 459.2578125, 422.132
8125]
```

While the above gives you information about raw memory usage, the issue in your code might be an inefficient use of memory by having more objects around than are strictly needed. You can use objgraph to get access to what is taking up chunks of memory. You can install it with

- pip install objgraph

Then you can start with a report of the most common objects in memory.

In [12]:
```
import objgraph

objgraph.show_most_common_types()
print('Show growth of objects')
objgraph.show_growth()
myload2(1000000)
objgraph.show_growth()
```

```
function                   7609
dict                       4984
tuple                      3620
weakref                    1891
list                       1362
wrapper_descriptor         1195
getset_descriptor          1060
builtin_function_or_method 961
method_descriptor          910
type                       890
Show growth of objects
function                      7607     +7607
dict                          4813     +4813
tuple                         3224     +3224
weakref                       1892     +1892
list                          1325     +1325
wrapper_descriptor            1195     +1195
getset_descriptor             1060     +1060
builtin_function_or_method     960      +960
method_descriptor              910      +910
type                           890      +890
Interactive        1        +1
```

Optimizing Python

Now that you have some idea of what parts of your code are most problematic, you can now start to look at what you could do to get better performance out of it. One of the major efficiency problems with Python is also one of its greatest powers, the combination of untyped variables and object oriented programming. Because variables are untyped, Python ends up having to check what object it refers to each time it is ever accessed. Many of the optimization ideas are based on minimizing this issue. The following is simply a list of possible ideas in no particular order.

Use keys and sort

Sorting is something that can easily be done inefficiently. Luckily, Python has an extremely efficient sort method in lists.

In [13]:
```
a = [1,74,5,2,8,3,9,10,2]
%timeit a.sort()
a
```

```
The slowest run took 6.94 times longer than the fastest. This could mea
n that an intermediate result is being cached.
1000000 loops, best of 3: 817 ns per loop
```

Out[13]:
```
[1, 2, 2, 3, 5, 8, 9, 10, 74]
```

As you can see, the sort is done in place. You can sort more complicated lists using keys. Say you had a list of tuples; you can sort based any of the values contained.

In [14]:
```
import operator
b = [(1,2,3),(9,1,7),(3,4,5)]
%timeit b.sort(key=operator.itemgetter(0))
print(b)
%timeit b.sort(key=operator.itemgetter(1))
print(b)
```

```
100000 loops, best of 3: 4.1 µs per loop
[(1, 2, 3), (3, 4, 5), (9, 1, 7)]
100000 loops, best of 3: 4.04 µs per loop
[(9, 1, 7), (1, 2, 3), (3, 4, 5)]
```

Avoiding dots

Whenever you use object methods or module functions and give the full path to them, Python needs to resolve and check every step along the way. This happens everytime it is referenced. For example, making a list of words upper case involves:

In [15]:
```
%%timeit
lowercase = ['this', 'is', 'a', 'lower', 'case', 'string']
uppercase = []
for word in lowercase:
    uppercase.append(str.upper(word))
```

```
100000 loops, best of 3: 7.12 µs per loop
```

You can speed up the references to str.upper and uppercase.append, as below:

In [16]:
```
upper = str.upper
uppercase = []
append = uppercase.append
```

In [17]:
```
%%timeit
lowercase = ['this', 'is', 'a', 'lower', 'case', 'string']
for word in lowercase:
    append(upper(word))
```

```
100000 loops, best of 3: 5.11 µs per loop
```

Using different coding techniques

Sometimes, using a different technique to get the same result will alter the amount of time used. Say you were creating a dictionary of characters and how many times they were used. As an example, we'll use the string 'abcd' and simply cycle over them. The first way to create the dictionary would be to create an empty one, and add new entries if the character isn't already being counted.

In [18]:
```
%%timeit
n = 16
myDict = {}
for i in range(0, n):
    char = 'abcd'[i%4]
    if char not in myDict:
        myDict[char] = 0
    myDict[char] += 1
```

```
100000 loops, best of 3: 15 µs per loop
```

A different technique would be to use a try/except block instead.

In [19]:
```
%%timeit
n = 16
myDict = {}
for i in range(0, n):
    char = 'abcd'[i%4]
    try:
        myDict[char] += 1
    except KeyError:
        myDict[char] = 1
```

```
10000 loops, best of 3: 19.5 µs per loop
```

For small data, the first is slightly faster. For larger data, the reverse is true.

In [20]:
```
%%timeit
n = 1600
myDict = {}
for i in range(0, n):
    char = 'abcde'[i%5]
    if char not in myDict:
        myDict[char] = 0
    myDict[char] += 1
```

```
1000 loops, best of 3: 1.25 ms per loop
```

In [21]:
```
%%timeit
n = 1600
myDict = {}
for i in range(0, n):
    char = 'abcde'[i%5]
    try:
        myDict[char] += 1
    except KeyError:
        myDict[char] = 1
```

```
1000 loops, best of 3: 1.16 ms per loop
```

List Comprehensions

When you need to create lists of values, you can do the normal thing of using a for loop.

In [22]:
```
%%timeit
a = []
for b in range(1000):
    a.append(b**2)
```

```
1000 loops, best of 3: 1.22 ms per loop
```

You can use comprehensions, instead, where you define the creation rule within the list definition. This may be faster in some cases, as below.

In [23]:
```
%%timeit
a = [x**2 for x in range(1000)]
```

```
1000 loops, best of 3: 1 ms per loop
```

You can make very complex creation rules, as long as they can be defined with if and for statements.

Use Numpy

In scientific computations, numerical work is the bulk of what your code needs to handle. You can speed these types of operations up by having external libraries written in C do all of the heavy lifting. The main library is numpy. As an example, let's say we want to scale a matrix.

In [24]:
```
%%timeit
w = 100
h = 100
matrix1 = [[1 for x in range(w)] for y in range(h)]
for x in range(h):
    for y in range(w):
        matrix1[x][y] = 5 * matrix1[x][y]
```

```
100 loops, best of 3: 5.73 ms per loop
```

In [25]:
```
import numpy
```

In [26]:
```
%%timeit
w = 100
h = 100
matrix1 = numpy.ones((w, h))
matrix1 = 5 * matrix1
```

```
The slowest run took 8.17 times longer than the fastest. This could mea
n that an intermediate result is being cached.
10000 loops, best of 3: 54.1 µs per loop
```

Replace for with map

The map function takes the explicit looping of a for loop and turns it into an implicit looping. This moves the bulk of the loop to the interpreter level.

In [27]:
```
%%timeit
uppercase = []
lowercase = ['this', 'is', 'a', 'lower', 'case', 'string','this', 'is',
 'a', 'lower', 'case', 'string','this', 'is', 'a', 'lower', 'case', 'str
ing','this', 'is', 'a', 'lower', 'case', 'string']
for word in lowercase:
    uppercase.append(str.upper(word))
```

```
10000 loops, best of 3: 23 µs per loop
```

In [28]:
```
%%timeit
lowercase = ['this', 'is', 'a', 'lower', 'case', 'string','this', 'is',
 'a', 'lower', 'case', 'string','this', 'is', 'a', 'lower', 'case', 'str
ing','this', 'is', 'a', 'lower', 'case', 'string']
uppercase = map(str.upper, lowercase)
```

```
1000000 loops, best of 3: 1.59 µs per loop
```

Moving to local variables

In cases where you can't use map, you can move your for loop to a function so that all of the intermediate variables are local. Local variables are accessed much more efficiently than global variables.

In [29]:
```
def myupper(lowercase):
    upper = str.upper
    newlist = []
    append = newlist.append
    for word in lowercase:
        append(upper(word))
    return newlist
```

In [30]:
```
%%timeit
uppercase = []
lowercase = ['this', 'is', 'a', 'lower', 'case', 'string','this', 'is',
 'a', 'lower', 'case', 'string','this', 'is', 'a', 'lower', 'case', 'str
ing','this', 'is', 'a', 'lower', 'case', 'string']
for word in lowercase:
    uppercase.append(str.upper(word))
```

```
10000 loops, best of 3: 23 µs per loop
```

In [31]:
```
%%timeit
lowercase = ['this', 'is', 'a', 'lower', 'case', 'string','this', 'is',
 'a', 'lower', 'case', 'string','this', 'is', 'a', 'lower', 'case', 'str
ing','this', 'is', 'a', 'lower', 'case', 'string']
uppercase = myupper(lowercase)
```

```
10000 loops, best of 3: 16.5 µs per loop
```

Minimizing function calls

One of the more expensive operations in Python is the function call. The example below is a function that is too general.

In [32]:
```
x = 0
def doit1(i):
    global x
    x = x + i

list = range(100000)
```

In [33]:
```
%%timeit
for i in list:
    doit1(i)
```

```
10 loops, best of 3: 53 ms per loop
```

Moving the for loop into the function, and aggregating your data into a list, gives you:

In [34]:
```
x = 0
def doit2(list):
    global x
    for i in list:
        x = x + i

list = range(100000)
```

In [35]:
```
%%timeit
doit2(list)
```

```
10 loops, best of 3: 25.6 ms per loop
```

Don't import ideas from other languages

There are tricks that work well in other languages that do not port to Python. An example is doubling a value, which can be done with

- a * 2
- a<<1
- a + a

In C, the middle one works fastest. What about Python?

In [36]:
```
%%timeit
x=42
y = 2*x
```

```
The slowest run took 9.63 times longer than the fastest. This could mea
n that an intermediate result is being cached.
10000000 loops, best of 3: 147 ns per loop
```

In [37]:
```
%%timeit
x=42
y = x<<1
```

```
The slowest run took 8.67 times longer than the fastest. This could mea
n that an intermediate result is being cached.
1000000 loops, best of 3: 218 ns per loop
```

In [38]:
```
%%timeit
x=42
y = x + x
```

```
The slowest run took 10.02 times longer than the fastest. This could me
an that an intermediate result is being cached.
10000000 loops, best of 3: 141 ns per loop
```

The shift method actually is the worst version in Python.

Cython

You can move particularly problematic code into its own external C module of code. Cython lets you import the compiled C code back into your Python program. Scientific distributions, such as Anaconda, have it included.

You can use it with IPython notebooks with

In [21]:
```
%load_ext Cython
```

The easiest way to take advantage of it is to use statically typed variables. This way, Python knows what the variables refer to and doesn't need to check.

In [22]:
```
def sum1():
    a = 0
    for i in range(1000000):
        a += i
%timeit sum1
```

```
10000000 loops, best of 3: 88.9 ns per loop
```

In [23]:
```
%%cython
def sum2():
    cdef int a = 0
    for i in range(10):
        a += i
```

In [24]:
```
%timeit sum2
```

```
10000000 loops, best of 3: 64.5 ns per loop
```

For more complex problems, you can write your Cython code in source files with '.pyx' as a filename ending. You then go through a compile step. It is then usable from your Python code. There is a large amount of information available at

- http://docs.cython.org (http://docs.cython.org)

Setting it up correctly on Windows is painful, since you need to have a C compiler that Cython can find and use.

Moving from CPython to PyPy

There are many different implementations of Python, written in different languages. CPython is the standard implementation. Jython is one that has been implemented in Java.

Pypy is a highly optimized implementation that optimizes your Python code for a JIT (Just-In-Time) compiler. This way, many of the performance issues of checking objects before using them is optimized away. You only get speed-ups for code that runs for long periods of time.