Introduction

This document is designed to be a concise and easily referenceable guide for software engineers on how to benchmark Python code and how to optimize it. As a result, it should help Python developers write more efficient Python code in the future.

Profilers

A profiler is a piece of software that allows its users to identify the speed of different parts of a program. There are different levels to profiling. The highest level, or easiest to do is to time the execution of one process, or a deeper analysis would include benchmarking specific functions and so on. In this section the document goes through some profilers in the order of deeper profiling.

Use timeit to benchmark a function

The simplest way to benchmark a python application is with the builtin module `timeit`.

You can run timeit on a function easily:

```
>>> import my_func
>>> import timeit
>>> timeit.Timer(lambda: my_func()).timeit(number=100)
0.1962261199951172
```

It seems like `my_func` took almost 0.2 s to execute on average over 100 run. When using `timeit` you should note the following:

- `timeit` repeats the function 1,000,000 times by default (this number can be changed) and averages the result
- `timeit` turns garbage collection off to make the timing result more precise and repeatable, so make sure there's enough ram

Function profiling

A function profiler will observe how long each function runs. There are a few different python profilers. In the linked article you can read about different function profilers. Each one differs from the others in some aspects. For example `cProfile` is supposed to be a more efficient profiler as it is written in C. Take the following example:

```
import cProfile
import time
def a(life):
    for a in range(life):
            time.sleep(2)
            b()
def b():
    now = time.time()
    while(time.time() < now + 3):
            pass

cProfile.run("a(4)")
```

In this example `cProfile` is timing a call to the function `a`, as it evaluates the string given to it. We expect to spend some time in function `b`. Function `b` uses busy wait, while `a` uses sleep, busy waiting means that there are going to be a lot of calls to `time.time()`. The output of `cProfile` confirms our expectations:

```
         48464105 function calls in 20.015 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000   20.015   20.015 <stdin>:1(a)
        4    7.868    1.967   12.000    3.000 <stdin>:1(b)
        1    0.000    0.000   20.015   20.015 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.
            Profiler' objects}
        1    0.000    0.000    0.000    0.000 {range}
        4    8.015    2.004    8.015    2.004 {time.sleep}
 48464093    4.132    0.000    4.132    0.000 {time.time}
```

Line profiling

Further profiling is possible with a tool like line_profiler. This tool finds how long each line is executed for instead of only measuring function run times. This will help pinpointing efficiency issues to specific operations.

Run it from shell with `$ kernprof -lv script_to_profile.py`

Running this profiler on the previous example will produce an output like so (Note the added `profile` decorators, these don't need to be imported):

```
Wrote profile results to script_to_profile.py.lprof
Timer unit: 1e-06 s

Total time: 20.0099 s
File: script_to_profile.py
Function: a at line 4

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     4                                           @profile
     5                                           def a(life):
     6         5         18.0      3.6      0.0      for a in range(life):
     7         4    8009732.0 2002433.0     40.0          time.sleep(2)
     8         4   12000134.0 3000033.5     60.0          b()

Total time: 7.40769 s
File: script_to_profile.py
Function: b at line 10

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    10                                           @profile
    11                                           def b():
    12         4         18.0      4.5      0.0      now = time.time()
    13  10665687    4701745.0      0.4     63.5      while(time.time() < now +
        3):
    14  10665683    2705927.0      0.3     36.5          pass
```

Memory profiler

This profiler is a bit different as it profiles each lines' impact on memory usage instead of execution time. Using it is pretty straight forward, just add `@profile` decorator to the function we want to memory profile (note these do not need to be imported). Take the following example:

```
@profile
def my_func():
    a = [1] * (10 ** 6)
    b = [2] * (2 * 10 ** 7)
    del b
    return a

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

Memory profiler can be ran on this file like this:

```
$python -m memory_profiler file.py
Line #    Mem usage  Increment   Line Contents
================================================
     3                           @profile
     4      5.97 MB    0.00 MB   def my_func():
     5     13.61 MB    7.64 MB       a = [1] * (10 ** 6)
     6    166.20 MB  152.59 MB       b = [2] * (2 * 10 ** 7)
     7     13.61 MB -152.59 MB       del b
     8     13.61 MB    0.00 MB       return a
```

Optimizing Python code

In this section the document goes through how to make Python code execute faster. The methods are arranged in increasing order of effort to apply the optimization to an already existing Python app.

Take advantage of memoization

Writing algorithms with better run time is how people usually optimize Python code. Take the following naive way of implementing a function that returns the Nth Fibonacci number.

```
 def fib(num):
     if num < 2:
             return num
     else:
             return fib(num-1) + fib(num-2)

timeit.Timer(lambda: fib(40)).timeit(1)
```

The result of this script will tell us that `fib(40)` takes about 60 seconds to run.

This run time can be accelerated by using memoization, a form of dynamic programming. The idea is to not call the function fib with the same number twice, but instead reuse previously calculated numbers. In Python 3, doing this is trivial. Just add the decorator `@lru_cache(maxsize=32)` to fib and we'll see that this time `fib(40)` takes only about 3 microseconds.

Use NumPy's ndarray when possible instead of Python lists

Python lists are wonderful, because they can contain any mixture of object types. However, this feature also adds a lot of memory overhead. A memory optimized alternative is `numpy.ndarray`, which can only

hold objects of one type. These n-dimensional arrays are a lot like C arrays, so they are statically sized, but in return they work faster, consume far less memory and provide more sophisticated methods than Python lists.

Here is the output of `memory_profiler`, notice the size difference between the list and the `ndarray`:

```
$ python -m memory_profiler test.py
Filename: test.py

Line #    Mem usage    Increment   Line Contents
================================================
     3   19.652 MiB   19.652 MiB   @profile
     4                             def create_lists():
     5  330.469 MiB  310.816 MiB       big_list = range(10000000)
     6  406.773 MiB   76.305 MiB       smaller_list = numpy.arange(0, 10000000)
```

NumPy's `ndarray`s can be used just like Python lists for the most basic functions, like accessing elements and assigning values to them:

```
>>> arr = np.array([1, 2, 3])
>>> arr[0] = arr[2]
>>> arr
array([3, 2, 3])
```

However; evidently it is not designed to be a drop-in replacement for Python's lists:

```
>>> a = np.array([1,2,3])
>>> b = np.array([4,5,6])
>>> a.extend(b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'numpy.ndarray' object has no attribute 'extend'
>>> a.append(b)
Traceback (most ///recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'numpy.ndarray' object has no attribute 'append'
>>> c = a + b
>>> c
array([5, 7, 9])
```

Ndarrays are missing `append`, `extend`, and their + behavior is different from normal lists', etc. They can hold basic data types and with some extra work they can hold class instances. Ndarray is designed to work with large data sets of basic data types. It works especially well for big data and machine learning purposes.

Use optimized libraries

When using third party libraries, you should keep performance in mind and determine which possible library might perform best for your use cases. For example, take NumExpr, which is a package that aims to make NumPy faster and use less memory. They claim that "the main reason why NumExpr achieves better performance than NumPy is that it avoids allocating memory for intermediate results. This results in better cache utilization and reduces memory access in general. Due to this, NumExpr works best with large arrays". Its performance is further aided by having a highly parallel workflow by having its virtual machine chunk the array's data and distribute it among the available CPU cores.

However, NumExpr is not the only package that does this. There are tons of packages that increase performance of other widely used packages. Finding them can be tricky, there are some sites like Intel's list of Python packages optimized by them for their processors.

Use optimized python interpreter

Many people call Python an interpreted language, but it is actually compiled, just not in the same way that for example C is. It does way less optimization and so it can get away with not doing it ahead of time, but as a project gets bigger and bigger the compilation time will show itself.

A simple solution to this is to use pypy, which is an alternative implementation of Python which uses a JIT compiler to increase performance. It is highly compatible with existing Python code and it also supports stackless mode and fully integrates with some other popular libraries, like Twisted and Django.

Compile python

C and C++ are known for their speed in part due to them being compiled, while Python is known for its user friendliness and speed of development. There are ways to combine the best of both worlds. We can write Python code and cross compile it into another more optimizable language like C, or C++.

One of the projects that does this is called Cython. By adding static types to regular Python code Cython can optimize it to have better performance. It also allows any Python application to easily interface with C code, or to easily build a Python wrapper around C code. The CPython ecosystem is also mature and widely used.

Numba is another very interesting project that uses LLVM to compile Python code just-in-time (JIT), or ahead of time with pycc. Its performance is comparable to C and C++. It is different from Cython, because it does not need static types, it can infer the type of each variables.

Wrap compiled code in python

Although we discussed many ways to speed up Python code, there are times when using another language to do some of the computation is recommended. Sometimes using another program that is written in another programming language is more efficient than putting in the effort to develop a Python program with the same functionality and then optimize it. For example, if there is a library written in C that does exactly what we need.

In this case we can wrap these libraries in a Python wrapper and reuse them.

SWIG is a project that allows C and C++ code to be used in other languages like Python. The only extra step a developer has to take is to create an interface file, which looks similar to a C header file. For example:

```
/* example.i */
%module example
%{
/* Put header files here or function declarations like below */
extern double My_variable;
extern int fact(int n);
extern int my_mod(int x, int y);
extern char *get_time();
%}

extern double My_variable;
extern int fact(int n);
extern int my_mod(int x, int y);
extern char *get_time();
```

SWIG does not support as many features as some alternate projects. For example, Boost.Python is a C++ library that allows Python to interface with its code. It supports a few neat features, like overloading functions, having default arguments, document strings and manipulating Python objects.

While Cython can be used to compile Python code into low level C code. It can also be used to interface with C code easily.