Profiling in Python

Fabian Pedregosa

CHAIRE HAVAS-DAUPHINE

Profiling : measure information of a running program (timing, memory consumption, disk usage, etc.)

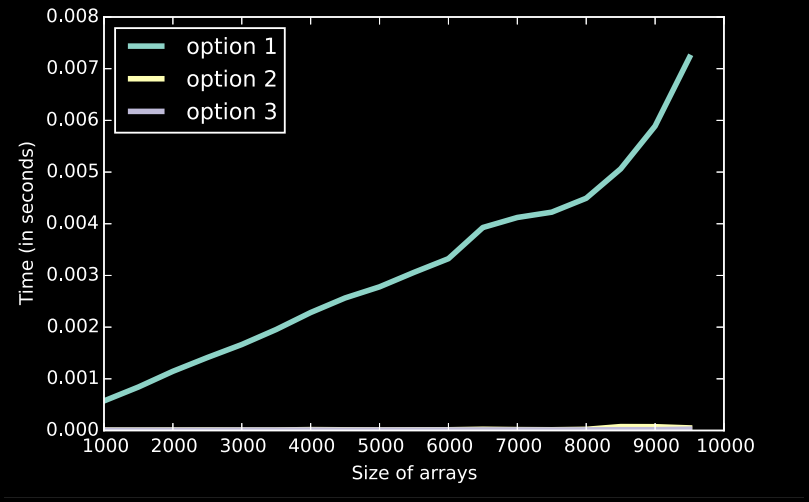
- Profiling is about avoiding unnecessary optimizations and focusing on important ones.
- It is not enough to known algorithmic to make optimal code:

```
# three equivalent
# ways to add two vectors

# option 1
for i in range(a.size):
    a[i] = a[i] + b[i]

# option 2
a = a + b

# option 3
a += b
```



- Profiling is necessary to make meaningful comparisons.

The main Python profilers

- **cProfile** (in the standard library)
- **line_profiler**: line-by-line timing.
- **memory_profiler**: to get memory usage information (☺)
- IPython magic commands.
- **yep** extension profiler: to get timing of compiled extensions (☺)

Example: Gradient-descent algorithm

```
import numpy as np
from scipy import optimize


def gd(f, grad, x0, rtol=1e-3):
    xk = x0
    fk = f(xk)
    tol = np.inf
    while tol >= rtol:
        # .. compute gradient ..
        gk = grad(xk)

        # .. find step size by line-search ..
        step_size = optimize.line_search(f, grad, xk, -gk, gfk=gk)[0]

        # .. perform update ..
        for i in range(xk.size):
            xk[i] = xk[i] - step_size * gk[i]
        fk_new = f(xk)
        tol = np.abs(fk - fk_new) / fk
        fk = fk_new
    return xk
```

Question: should I optimize the gradient evaluation, the line-search or the coefficient update ?

Profiling How-To

1. Create a realistic example
2. Run it with the appropriate profiler.
    - cProfile
    - line_profiler
    - memory_profiler
    - yep

(most of the time) cProfile is useless

```
$ python -m cProfile -s time gd.py

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        2    0.555    0.278    0.555    0.278 {method 'randn' of 'mtrand.RandomState' objects}
        1    0.465    0.465    1.075    1.075 samples_generator.py:38(make_classification)
   106/90    0.254    0.002    0.290    0.003 {built-in method load_dynamic}
       65    0.174    0.003    0.174    0.003 {built-in method dot}
      449    0.067    0.000    0.067    0.000 {built-in method loads}
        2    0.051    0.026    0.051    0.026 {method 'take' of 'numpy.ndarray' objects}
      261    0.039    0.000    0.094    0.000 doccer.py:12(docformat)
    628/1    0.039    0.000    2.453    2.453 {built-in method exec}
     3737    0.036    0.000    0.036    0.000 {built-in method stat}
        7    0.031    0.004    0.034    0.005 {method 'get_filename' of 'zipimport.zipimporter' o
     1985    0.031    0.000    0.122    0.000 <frozen importlib._bootstrap>:2016(find_spec)
1114/1108    0.030    0.000    0.086    0.000 {built-in method __build_class__}
        1    0.026    0.026    0.211    0.211 gd.py:5(gd)
      449    0.022    0.000    0.136    0.000 <frozen importlib._bootstrap>:1534(get_code)
      7/3    0.020    0.003    0.223    0.074 {method 'load_module' of 'zipimport'.
```

Enter line_profiler

- line-by-line time consumption
- decorate the function with `@profile`:

```
@profile
def gd(f, grad, x0, rtol=1e-3):
  xk = x0
  fk = f(xk)
  tol = np.inf
  while tol >= rtol:
      # .. compute gradient ..
      gk = grad(xk)

      # etc.
```

- run the code with `kernprof -l -v my_script.py`
- written by Robert Kern

```
$ kernprof -l -v gd.py

Total time: 0.561124 s
File: gd.py
Function: gd at line 4

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     4                                           @profile
     5                                           def gd(f, grad, x0, rtol=1e-12):
     6         1            3      3.0      0.0      xk = x0
     7         1          998    998.0      0.2      fk = f(xk)
     8         1            2      2.0      0.0      tol = np.inf
     9        18           28      1.6      0.0      while tol >= rtol:
    10                                                   # .. compute gradient ..
    11        17        29337   1725.7      5.2          gk = grad(xk)
    12
    13                                                   # .. find step size by line-search ..
    14        17       101399   5964.6     18.1          step_size = optimize.line_search(f, grad
    15
    16                                                   # .. perform update ..
    17    170017       152034      0.9     27.1          for i in range(xk.size):
    18    170000       260621      1.5     46.4              xk[i] = xk[i] - step_size * gk[i]
    19        17        16550    973.5      2.9          fk_new = f(xk)
    20        17          135      7.9      0.0          tol = np.abs(fk - fk_new) / fk
    21        17           17      1.0      0.0          fk = fk_new
```

- Last column contains % of time spent on that line.
- More than 70% of time is spent in performing the parameter vector update!

vectorizing the parameter update

```
$ kernprof -l -v gd.py


Total time: 0.138711 s
File: gd.py
Function: gd at line 4

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     4                                           @profile
     5                                           def gd(f, grad, x0, rtol=1e-12):
     6         1            3      3.0      0.0      xk = x0
     7         1         1007   1007.0      0.7      fk = f(xk)
     8         1            2      2.0      0.0      tol = np.inf
     9        17           34      2.0      0.0      while tol >= rtol:
    10                                                   # .. compute gradient ..
    11        16        29335   1833.4     21.1          gk = grad(xk)
    12
    13                                                   # .. find step size by line-search ..
    14        16        91213   5700.8     65.8          step_size = optimize.line_search(f, grad
    15
    16                                                   # .. perform update ..
    17        16          589     36.8      0.4          xk -= step_size * gk
    18        16        16363   1022.7     11.8          fk_new = f(xk)
    19        16          145      9.1      0.1          tol = np.abs(fk - fk_new) / fk
    20        16           20      1.2      0.0          fk = fk_new
```

- from 48.5% to 0.4% !
- total time divided by 4, without leaving Python.

memory_profiler

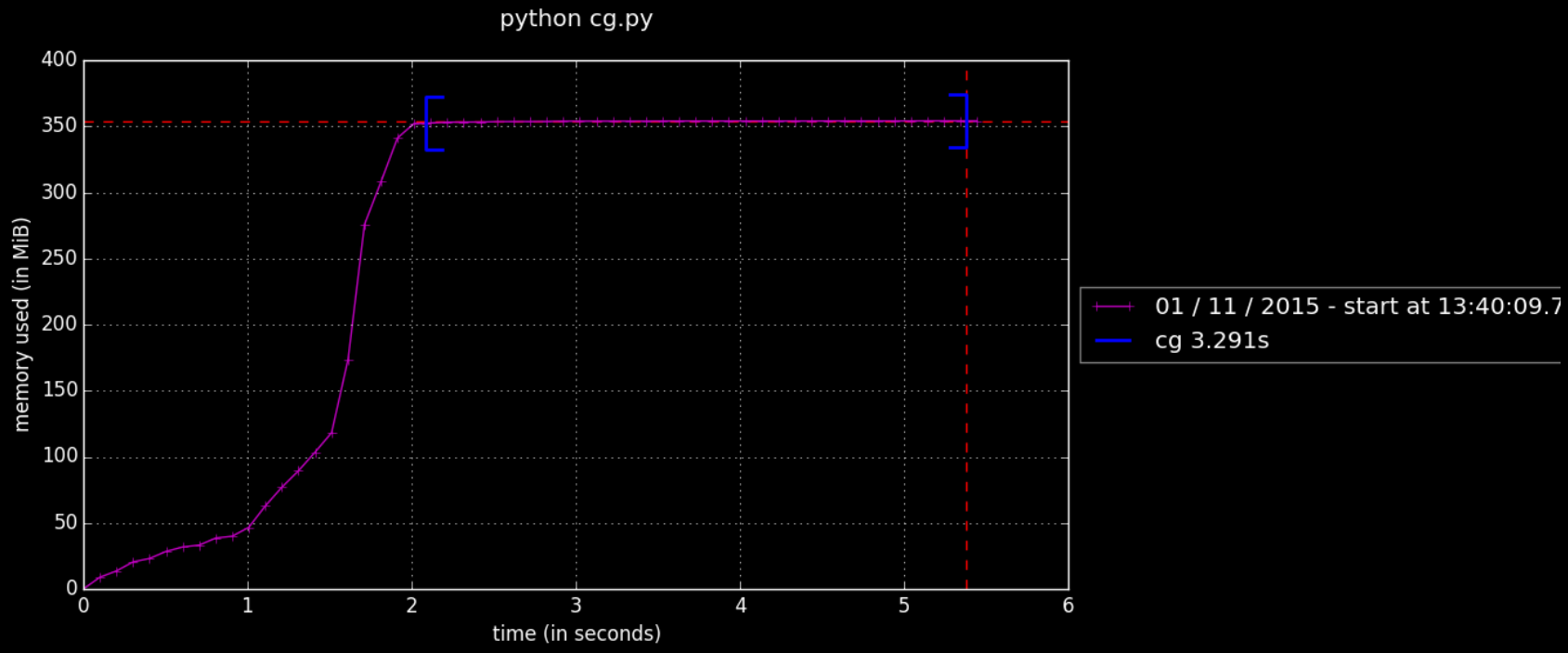
- Get *memory consumption* of a Python program.
- Two modes: line-by-line and time-based.
- line-by-line similar interface as line_profiler:

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

Line #    Mem usage    Increment   Line Contents
================================================
     4  352.117 MiB    0.000 MiB   @profile
     5                              def gd(f, grad, x0, rtol=1e-12):
     6  352.117 MiB    0.000 MiB       xk = x0
     7  352.145 MiB    0.027 MiB       fk = f(xk)
     8  352.145 MiB    0.000 MiB       tol = np.inf
     9  354.133 MiB    1.988 MiB       while tol >= rtol:
    10                                      # .. compute gradient ..
    11  354.133 MiB    0.000 MiB           gk = grad(xk)
    12
    13                                      # .. find step size by line-search ..
    14  354.133 MiB    0.000 MiB           step_size = optimize.line_search(f, grad, xk, -gk,
    15
    16                                      # .. perform update ..
    17  354.133 MiB    0.000 MiB           xk -= step_size * gk
    18  354.133 MiB    0.000 MiB           fk_new = f(xk)
    19  354.133 MiB    0.000 MiB           tol = np.abs(fk - fk_new) / fk
    20  354.133 MiB    0.000 MiB           fk = fk_new
```

memory_profiler

- time-based memory consumption.
- run script with `mprof run my_script.py`
- plot with `mprof plot`



memory_profiler, example 2

- Two functions, process_data, create_data

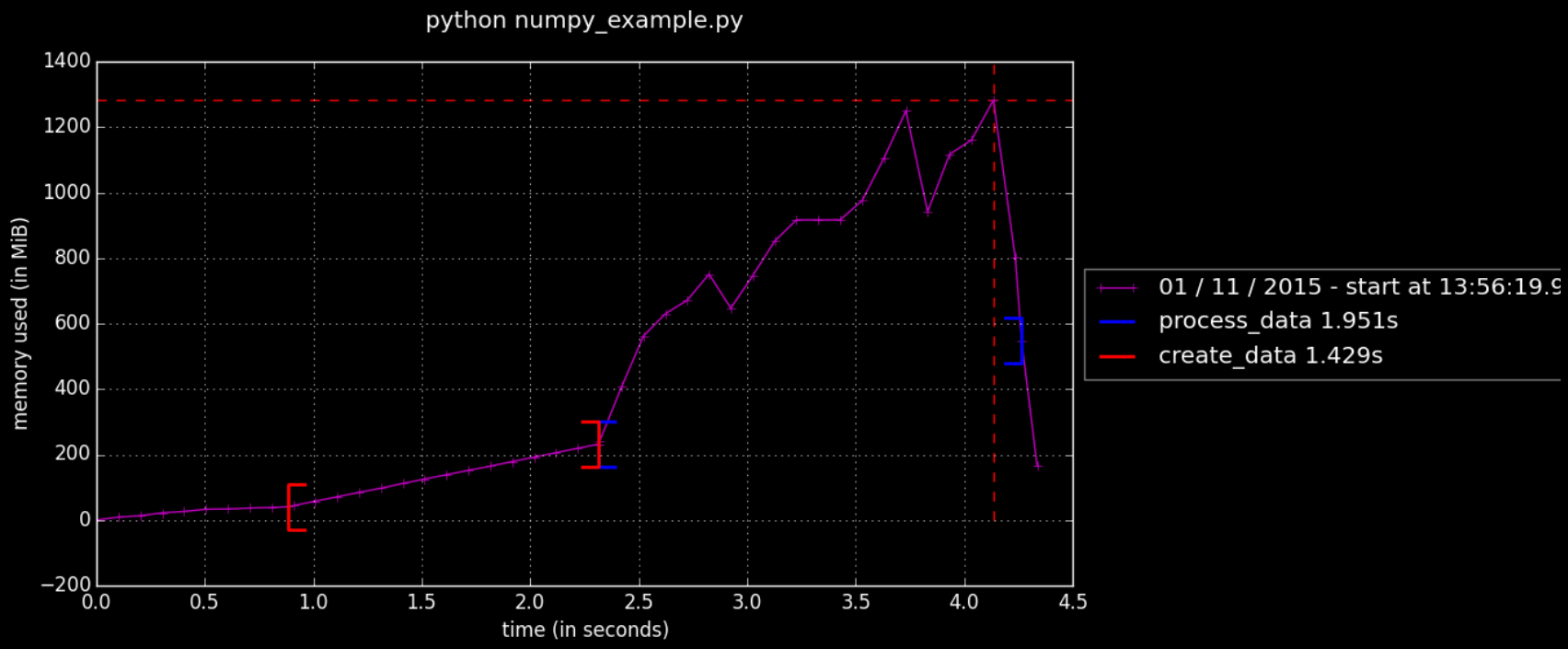
```
import numpy as np
import scipy.signal


@profile
def create_data():
    ret = []
    for n in range(70):
        ret.append(np.random.randn(1, 70, 71, 72))
    return ret


@profile
def process_data(data):
    data = np.concatenate(data)
    detrended = scipy.signal.detrend(data, axis=0)
    return detrended
```

memory_profiler, example 2

- Two functions, process_data, create_data



IPython shortcuts

- `line_profiler` and `memory_profiler` have IPython shortcuts (AKA magic commands).
  - `line_profiler` shortcut: %lprun
  - `memory_profiler` shortcut: %memit, %mprun
  - IPython %timeit:

YEP

recursive acronym for YEP extension profiler

- Existing Python profilers only record Python functions.
- Many codebases contain significant parts written in Cython/C/C++.
- YEP Allows to get timing information for compiled extensions (Cython/C/C++)
- Only one that does it without special compiling/linking.
- Integrates within Python the `google-perftools` profiler.
- Does not work on Windows.

YEP

Simple example using scikit-learn

- The svm module in scikit-learn uses libsvm.

```
import numpy as np
from sklearn import svm, datasets

X, y = datasets.make_classification(n_samples=1000)

clf = svm.SVC()
clf.fit(X, y)
```

- run with `$ python -m yep -v simple_svm.py`

YEP output

```
$ python -m yep -v simple_svm.py
       2   0.7%   0.7%      268  97.5% _PyEval_EvalFrameEx
       0   0.0%   0.7%      267  97.1% _PyClassMethod_New
       0   0.0%   0.7%      267  97.1% _PyEval_EvalCode
       0   0.0%   0.7%      267  97.1% _PyEval_EvalCodeEx
       0   0.0%   0.7%      267  97.1% _PyObject_Call
       0   0.0%   0.7%      267  97.1% __PyBuiltin_Init
       0   0.0%   0.7%      206  74.9% __mh_execute_header
       0   0.0%   0.7%      193  70.2% _Py_GetArgcArgv
       0   0.0%   0.7%      193  70.2% _Py_Main
       0   0.0%   0.7%      193  70.2% _main
       0   0.0%   0.7%      193  70.2% start
       0   0.0%   0.7%      188  68.4% _PyInit_libsvm
       0   0.0%   0.7%      187  68.0% _svm_train
       0   0.0%   0.7%      186  67.6% svm::Solver::Solve
      48  17.5%  18.2%      184  66.9% svm_csr::Kernel::kernel_precomputed
     135  49.1%  67.3%      135  49.1% svm::Kernel::kernel_rbf
       0   0.0%  67.3%       74  26.9% _PyEval_CallObjectWithKeywords
       0   0.0%  67.3%       74  26.9% _PyImport_ImportModuleLevelObject
       0   0.0%  67.3%       74  26.9% __PyObject_CallMethodIdObjArgs
       0   0.0%  67.3%       71  25.8% svm::Solver::select_working_set
       0   0.0%  67.3%       27   9.8% _PyType_GenericAlloc
      25   9.1%  76.4%       25   9.1% _stat
       0   0.0%  76.4%       22   8.0% _PyInit_posix
```

- YEP is still a work in progress
- many rough edges.

Conclusion

- Python has many tools for code profiling.
- Some of them are mature and easy to use.
- Integration with IPython.
- Happy hacking!