A Comprehensive Guide to Profiling Python Programs

Understand what parts of your code are problematic

Eyal Trabelsi · Follow

Published in Better Programming
8 min read · Dec 1, 2020

Listen Share More

Photo by Jason Chen on Unsplash

Today, users have high expectations for the programs they use. Users expect programs to have amazing features, to be fast, and to consume a reasonable amount of memory.

As developers, we should thrive to give our users the best experience possible. We need to find and eliminate the bottlenecks before they affect our users. Unfortunately, as programs become more and more complex, it becomes harder to find those issues and bottlenecks.

Today, I am going to focus on a technique to find those bottlenecks, and this technique is profiling. My goal today is to show you that profiling is not rocket science. Everyone can find their bottlenecks without breaking a sweat.

The code for this article can be found on GitHub, and a more detailed recorded version can be found on YouTube.

Profiling Definition and Benefits

A profile is a set of statistics that describes how our program is executed. When we know how different parts of our code behave, we can use this information to optimize our code.

You must be asking yourself why should I even care about optimizing my code, my feature is already done. You should care about optimization because it affects our users and our company. These are the main reasons:

- **Fast programs are better than slow ones** — whether latency or throughput.
- **Memory efficiency is good** — Most of us are terrified of out-of-memory errors.
- **Saving money is awesome** — If out-of-memory errors do not terrify you, your AWS bills might.
- **Hardware has limits**— Even if we are willing to pay to improve performance, hardware will only take you so far.

Hopefully I have convinced you that profiling is a concept that every programmer needs in his toolbox. But before we jump into the water, I am going to cover some safety rules.

Safety Rules

The following rules were written with blood.

- **Make sure it’s actually needed** — Optimized code tends to be harder to write and read, which makes it less maintainable and buggier. We need to gather the requirements (SLA/SLOS) to understand our definition of “done.”
- **Make sure our code is well tested** — I know everyone says it, but it's really important. It will be sad if we fix the performance problem, but our program will not work.
- **Good work takes cycles** — We should focus on the problematic parts of the code (the bottleneck).

Now we’ve gone through the safety rules, we can finally jump into the water and get to know what types of profilers exist.

Types of Profilers

When we want to find our bottleneck, we need to use the right tool for the job, as there is no free lunch.

Picking the right tool for the job depends on the following characteristics:

- **The resource we want to measure** — whether it's CPU, RAM, I/O, or other exotic metrics.
- **Profiling strategy** — how the profiler collects the data: in a deterministic manner using hooks or in a statistical manner by collecting information at each time interval.
- **Profiling granularity** — at what level we get the information: program level, function level, or line level. Obviously, we can extract more insights if we use finer granularity levels, like line level, but that will add a lot of noise.

Our Example

In this article, I am going to use Peter Norvig's naive spelling corrector. Norvig's spelling corrector uses the Levenshtein distance to find spelling mistakes.

> *"The Levenshtein distance is a string metric for measuring the difference between the two words. The distance between two words is the minimum number of single-character edits (i.e., insertions, deletions, or substitutions) required to change one word into the other." — The Levenshtein Distance Algorithm*

Norvig's spelling corrector corrects spelling mistakes by finding the most probable words with a Levenshtein distance of two or less. This spelling corrector seems quite naive, but it actually achieves 80% or 90% accuracy.

```python
import re
from collections import Counter

def words(text):
    return re.findall(r'\w+', text.lower())

WORDS = Counter(words(open('big.txt').read()))

def P(word, N=sum(WORDS.values())):
    return WORDS[word] / N

def candidates(word):
    return (known([word]) or known(edits1(word)) or known(edits2(word)) or [word])

def known(words):
    return set(w for w in words if w in WORDS)

def edits1(word):
    letters    = 'abcdefghijklmnopqrstuvwxyz'
    splits     = [(word[:i], word[i:])    for i in range(len(word) + 1)]
    deletes    = [L + R[1:]               for L, R in splits if R]
    transposes = [L + R[1] + R[0] + R[2:] for L, R in splits if len(R)>1]
    replaces   = [L + c + R[1:]           for L, R in splits if R for c in letters]
    inserts    = [L + c + R               for L, R in splits for c in letters]
    return set(deletes + transposes + replaces + inserts)

def edits2(word):
    return (e2 for e1 in edits1(word) for e2 in edits1(e1))

def word_correction(word):
    return max(candidates(word), key=P)

def sentence_correction(sentence):
    return " ".join(word_correction(word) for word in sentence.split(" "))
```

spelling_corrector.py hosted with ❤ by GitHub view raw

We can use our spelling corrector to understand the following message “grofilingg is not rocet Sgience,” and we get “profiling is not rocket science.”

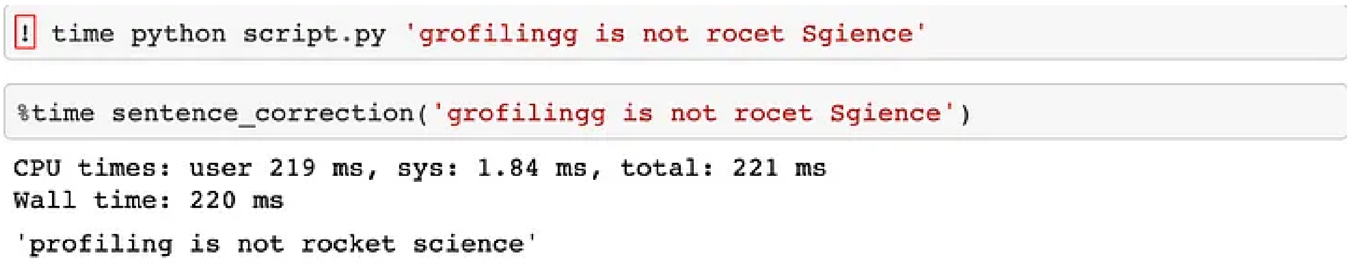
```
sentence_correction('grofilingg is not rocet Sgience')
'profiling is not rocket science'
```

We will start profiling our example with the first type of profiler, casual profilers.

Casual Profilers

Casual profilers give us a sense of how the program runs as a whole and allow us to understand whether a problem exists.

The first tool we going to cover is `time`, which helps us measure the user and system time for a single run. It is built-in in Python and comes with IPython magic, no additional installation is required.



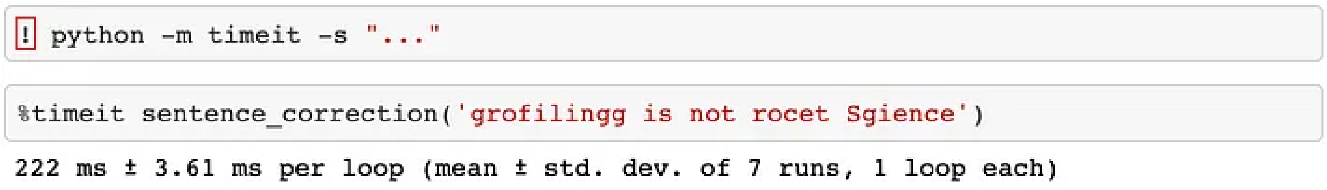
```
! time python script.py 'grofilingg is not rocet Sgience'
```

```
%time sentence_correction('grofilingg is not rocet Sgience')
```

```
CPU times: user 219 ms, sys: 1.84 ms, total: 221 ms
Wall time: 220 ms
'profiling is not rocket science'
```

The Wall time is the overall execution time, and the CPU time is the time that the program uses the CPU. In our example, we can see the Wall time is 220 milliseconds and 219 of it is spent on CPU, which means we are CPU bound.

The next tool I am going to cover is the `timeit` module, which helps us measure the execution time of multiple runs. It is built-in in Python, and comes with IPython, no additional installation required. It uses some neat tricks, like disabling the garbage collection, to make the result more consistent.



```
! python -m timeit -s "..."
```

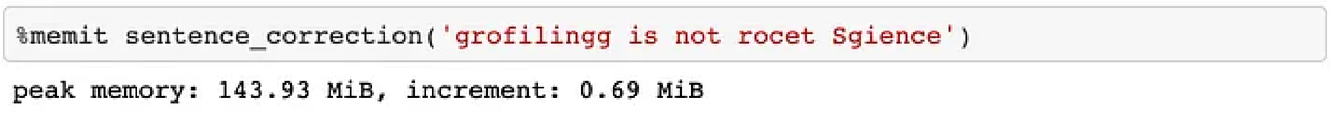
```
%timeit sentence_correction('grofilingg is not rocet Sgience')
```

```
222 ms ± 3.61 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

We can see that it was executed our program 7 times and the average execution time is 22 milliseconds with a 3.6 milliseconds variation between the executions.

In case you want similar capabilities, for comparing two (or more) pieces of code you fastero may be worth your time.

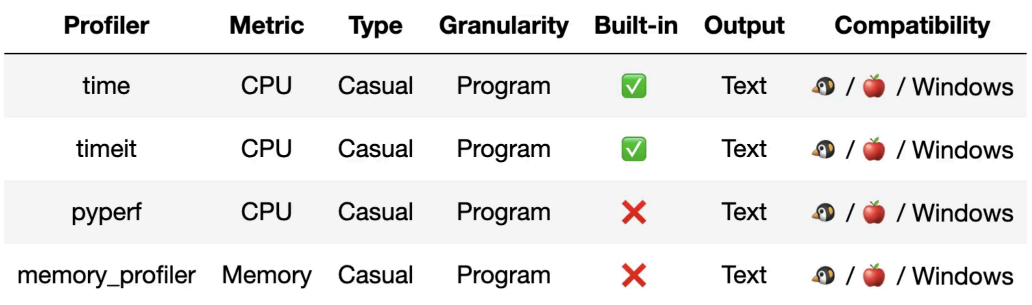
The last tool for casual profiling I am going to cover today is `memit` magic, which helps us measure the process memory. It not built-in in Python, so installation is required, but after we install it, we can use IPython magic.



```
%memit sentence_correction('grofilingg is not rocet Sgience')
peak memory: 143.93 MiB, increment: 0.69 MiB
```

The peak memory is the largest memory used by the process, and the increment is how much this specific call added to memory consumption. We can see that correcting one sentence increases memory consumption in 0.69 Mib.

There are a few more casual profiling tools, and the casual profiling landscape looks as follows:



Profiler	Metric	Type	Granularity	Built-in	Output	Compatibility
time	CPU	Casual	Program	✅	Text	🐧 / 🍎 / Windows
timeit	CPU	Casual	Program	✅	Text	🐧 / 🍎 / Windows
pyperf	CPU	Casual	Program	❌	Text	🐧 / 🍎 / Windows
memory_profiler	Memory	Casual	Program	❌	Text	🐧 / 🍎 / Windows

The casual profiling landscape

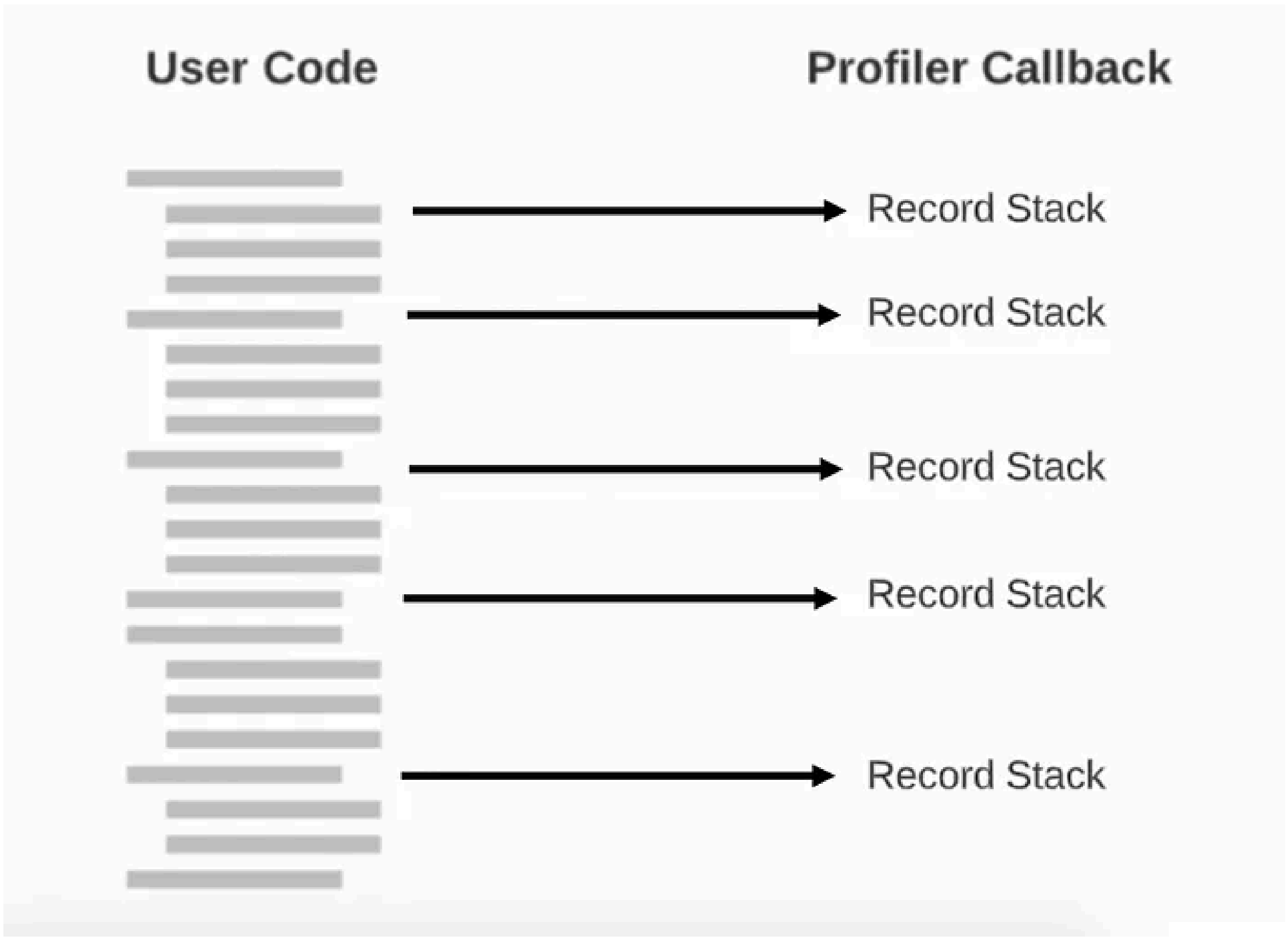
Casual profilers have the following pros and cons

- They are really easy to use.
- They allow us to understand whether a problem exists.
- But we can't use them to pinpoint the bottleneck.

The fact that we can't pinpoint the bottleneck is crucial and leads us to the next set of profilers, offline profilers.

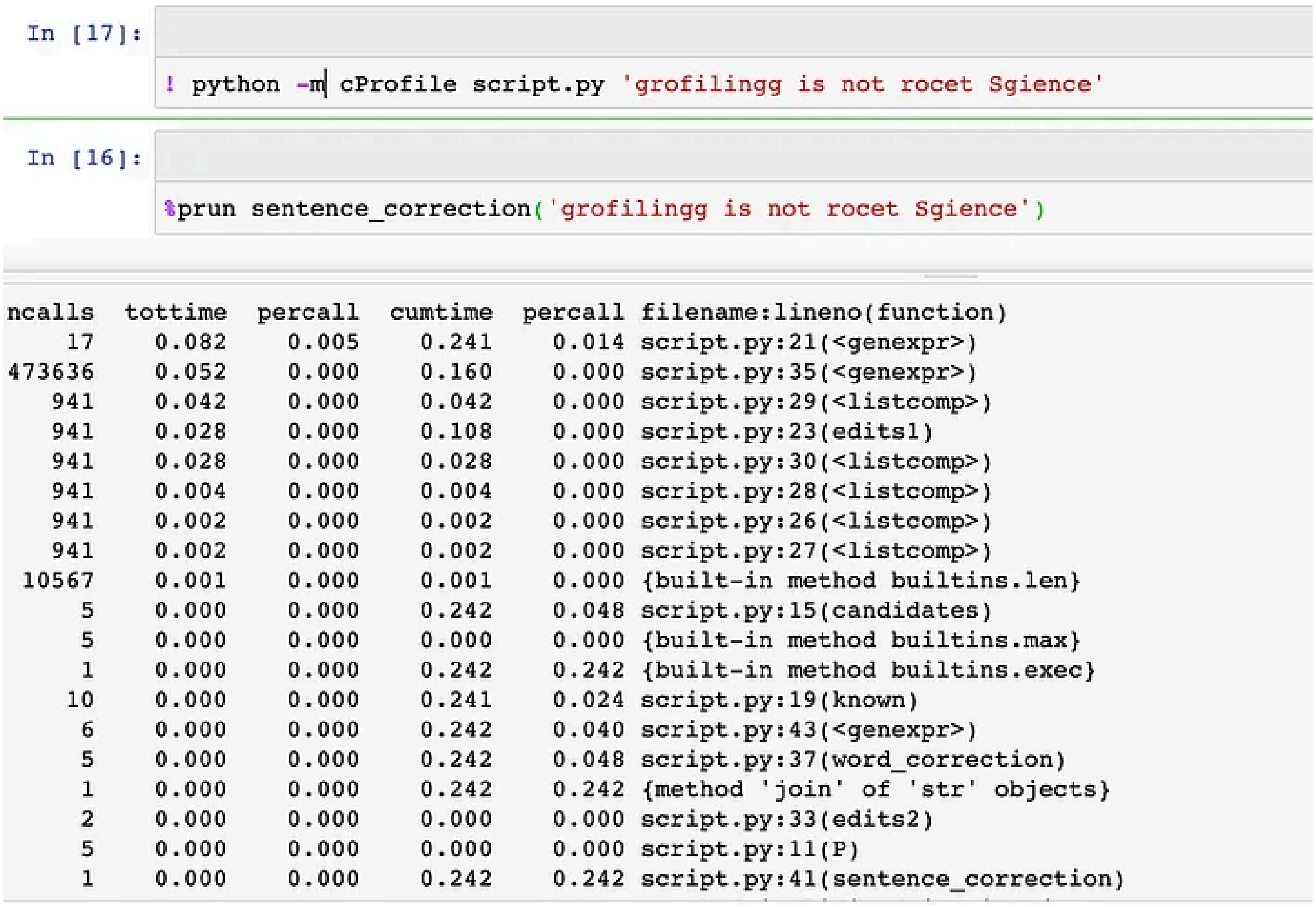
Offline Profilers

Offline profilers help us to understand the behavior of our program by tracking events like function calls, exceptions, and line executions. Because they execute hooks on specific events, they are deterministic and add significant overhead, which makes them more suitable for local debugging.



Python lets you specify a callback on specific interpreter events using sys.setprofile or sys.settrace. The callback records the stack.

The first tool we going to cover is the `cProfile`, which traces every function call and allows us to identify time-consuming functions. By default, it measures the process CPU but allows us to specify our own measurement. It is built-in in Python and comes with IPython magic, no additional installation is required. `cProfile` works only on the Python level and doesn't work across multiple processes.

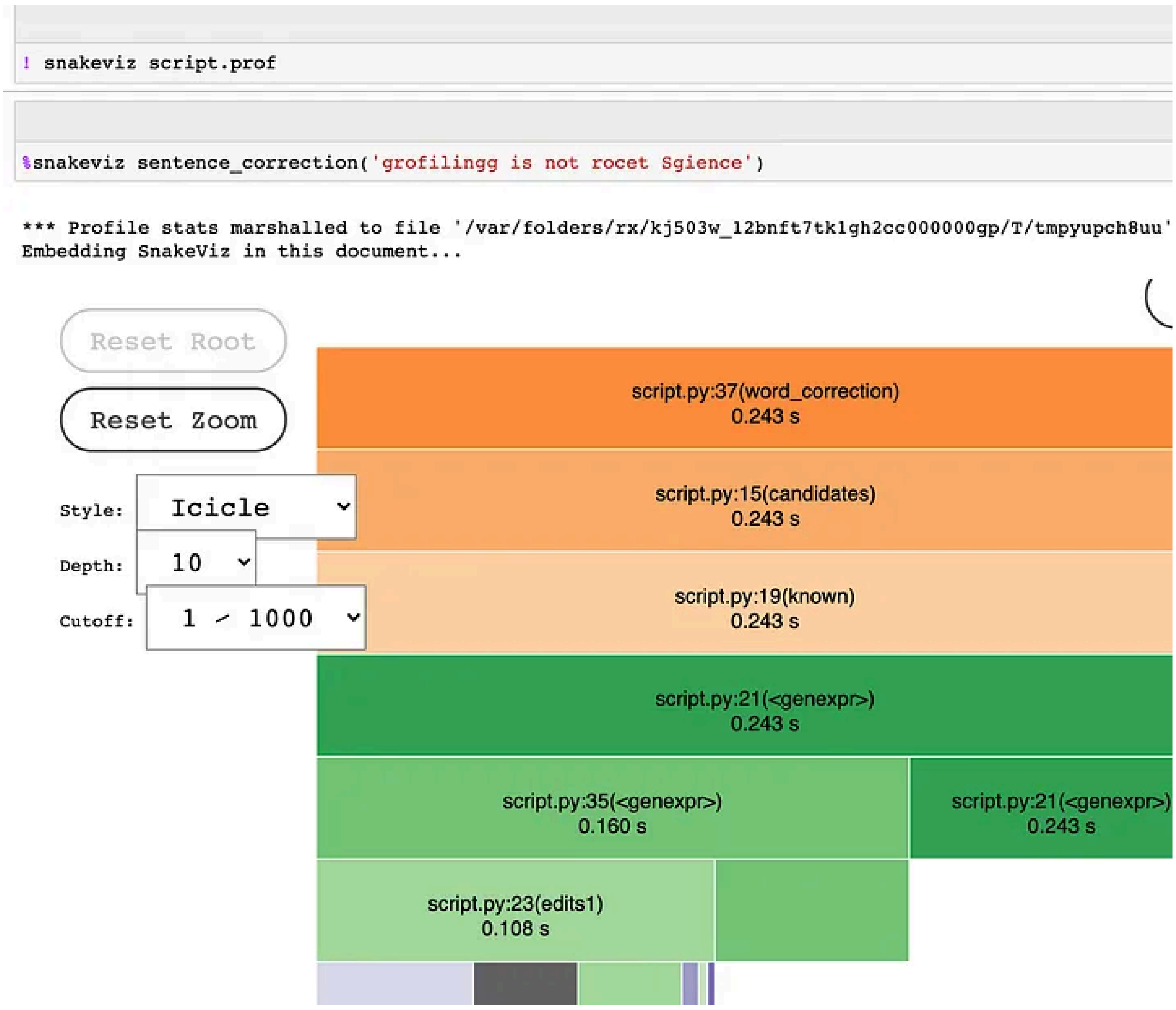


```
In [17]:  ! python -m cProfile script.py 'grofilingg is not rocet Sgience'

In [16]:  %prun sentence_correction('grofilingg is not rocet Sgience')
```

```
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    17    0.082    0.005    0.241    0.014 script.py:21(<genexpr>)
473636    0.052    0.000    0.160    0.000 script.py:35(<genexpr>)
   941    0.042    0.000    0.042    0.000 script.py:29(<listcomp>)
   941    0.028    0.000    0.108    0.000 script.py:23(edits1)
   941    0.028    0.000    0.028    0.000 script.py:30(<listcomp>)
   941    0.004    0.000    0.004    0.000 script.py:28(<listcomp>)
   941    0.002    0.000    0.002    0.000 script.py:26(<listcomp>)
   941    0.002    0.000    0.002    0.000 script.py:27(<listcomp>)
 10567    0.001    0.000    0.001    0.000 {built-in method builtins.len}
     5    0.000    0.000    0.242    0.048 script.py:15(candidates)
     5    0.000    0.000    0.000    0.000 {built-in method builtins.max}
     1    0.000    0.000    0.242    0.242 {built-in method builtins.exec}
    10    0.000    0.000    0.241    0.024 script.py:19(known)
     6    0.000    0.000    0.242    0.040 script.py:43(<genexpr>)
     5    0.000    0.000    0.242    0.048 script.py:37(word_correction)
     1    0.000    0.000    0.242    0.242 {method 'join' of 'str' objects}
     2    0.000    0.000    0.000    0.000 script.py:33(edits2)
     5    0.000    0.000    0.000    0.000 script.py:11(P)
     1    0.000    0.000    0.242    0.242 script.py:41(sentence_correction)
```

We can see our sentence_correction function took 0.24 seconds. All of it spent on the candidate function, and when we drill down, the edits1 takes 0.1 seconds (40%).

It's a lot of text to handle, and a bit encrypted for first comers. For this reason, the next tool was invented, `SnakeViz`. `SnakeViz` takes `cProfile` output and creates intuitive visualizations. It not built-in for Python, so installation is required, but after we install it, we can use IPython magic.



```
! snakeviz script.prof
```

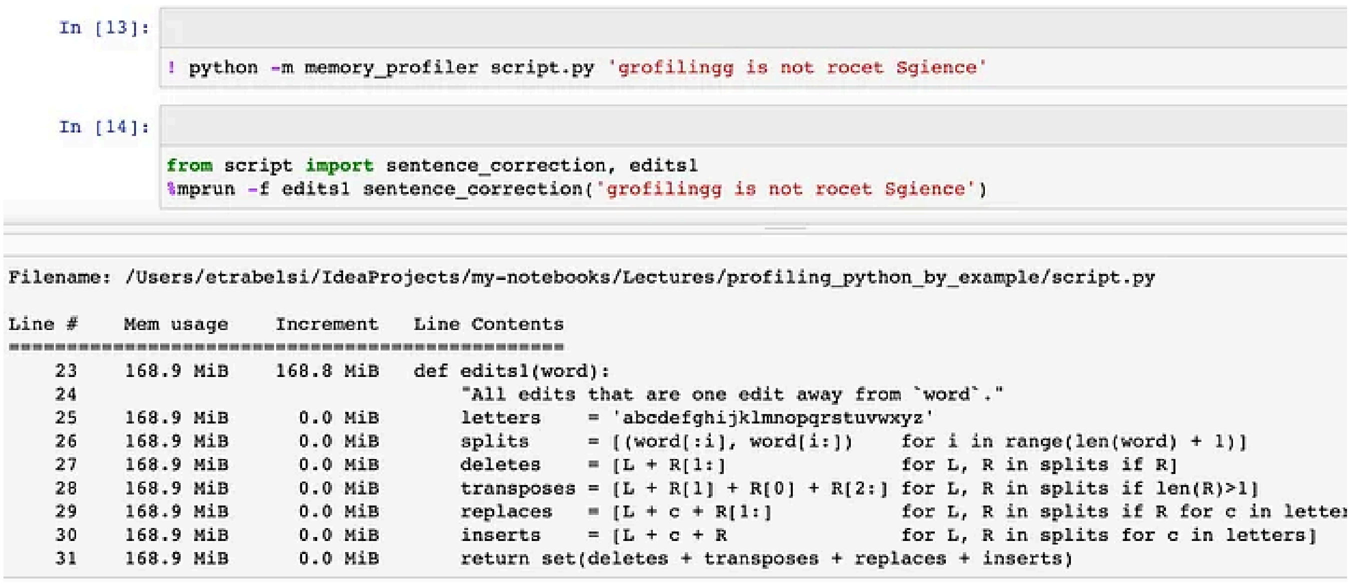
```
%snakeviz sentence_correction('grofilingg is not rocet Sgience')
```

```
*** Profile stats marshalled to file '/var/folders/rx/kj503w_12bnft7tk1gh2cc000000gp/T/tmpyupch8uu'
Embedding SnakeViz in this document...
```

Reset Root
Reset Zoom
Style: Icicle
Depth: 10
Cutoff: 1 / 1000
script.py:37(word_correction) 0.243 s
script.py:15(candidates) 0.243 s
script.py:19(known) 0.243 s
script.py:21(<genexpr>) 0.243 s
script.py:35(<genexpr>) 0.160 s
script.py:21(<genexpr>) 0.243 s
script.py:23(edits1) 0.108 s

It's much easier to see that all of it spent on the candidate function, and when we drill down, the edits1 takes 0.1 seconds (40%). Also, we can filter and change how many layers are being presented

The last tool for casual profiling I am going to cover today is `memory_profiler`. As the name suggests, it allows us to measure the effect of each line in a function on the memory footprint. It not built-in for Python, so installation is required, but after we install it, we can use IPython magic.

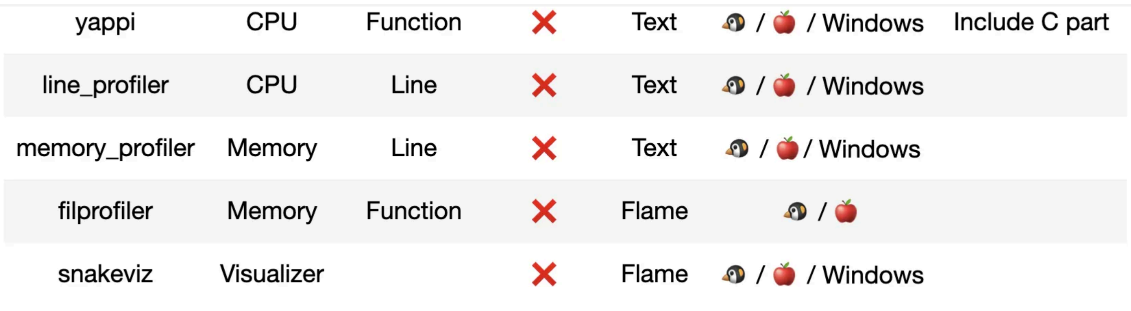


```
In [13]:  ! python -m memory_profiler script.py 'grofilingg is not rocet Sgience'

In [14]:  from script import sentence_correction, edits1
          %mprun -f edits1 sentence_correction('grofilingg is not rocet Sgience')
```

```
Filename: /Users/etrabelsi/IdeaProjects/my-notebooks/Lectures/profiling_python_by_example/script.py

Line #    Mem usage    Increment   Line Contents
================================================
    23    168.9 MiB    168.8 MiB   def edits1(word):
    24                                 "All edits that are one edit away from `word`."
    25    168.9 MiB      0.0 MiB       letters    = 'abcdefghijklmnopqrstuvwxyz'
    26    168.9 MiB      0.0 MiB       splits     = [(word[:i], word[i:])    for i in range(len(word) + 1)]
    27    168.9 MiB      0.0 MiB       deletes    = [L + R[1:]               for L, R in splits if R]
    28    168.9 MiB      0.0 MiB       transposes = [L + R[1] + R[0] + R[2:] for L, R in splits if len(R)>1]
    29    168.9 MiB      0.0 MiB       replaces   = [L + c + R[1:]           for L, R in splits if R for c in lette
    30    168.9 MiB      0.0 MiB       inserts    = [L + c + R               for L, R in splits for c in letters]
    31    168.9 MiB      0.0 MiB       return set(deletes + transposes + replaces + inserts)
```

We can see that we monitor the edits1 function memory footprint. We can see that we begin the function with 169 MBS and that each line doesn't add a lot of overhead. This leads us to believe most of the memory consumption is due to our vocabulary.

In some cases it will be easier to profile memory using filprofiler or memray. But in general there are a lot more offline profilers, and the offline profiling landscape looks as follows:



yappi	CPU	Function	❌	Text	🐧 / 🍎 / Windows	Include C part
line_profiler	CPU	Line	❌	Text	🐧 / 🍎 / Windows	
memory_profiler	Memory	Line	❌	Text	🐧 / 🍎/ Windows	
filprofiler	Memory	Function	❌	Flame	🐧 / 🍎	
snakeviz	Visualizer		❌	Flame	🐧 / 🍎 / Windows	

The offline profiling landscape

Offline profilers have the following pros and cons

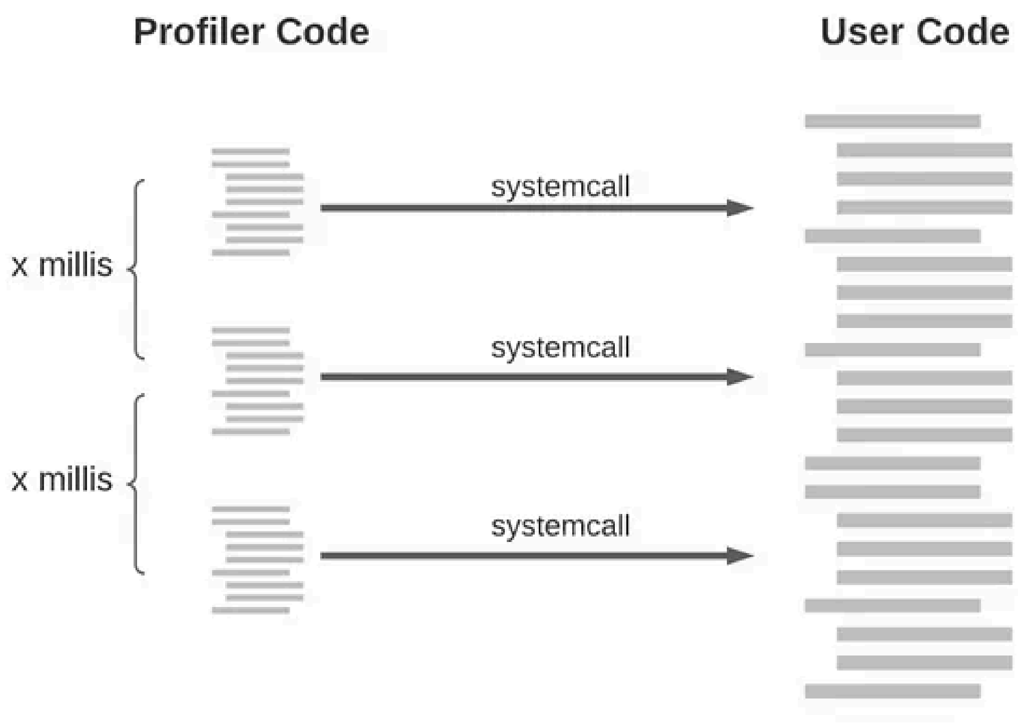
- They allow us to pinpoint the bottlenecks.
- They are deterministic.
- They have a high overhead.
- They are really easy to use.
- Their output can be noisy.
- They can't tell us which inputs are slow.

- They distort certain parts of the programs because it runs hooks only on specific events.

These bring us to our last type of profilers, online profilers.

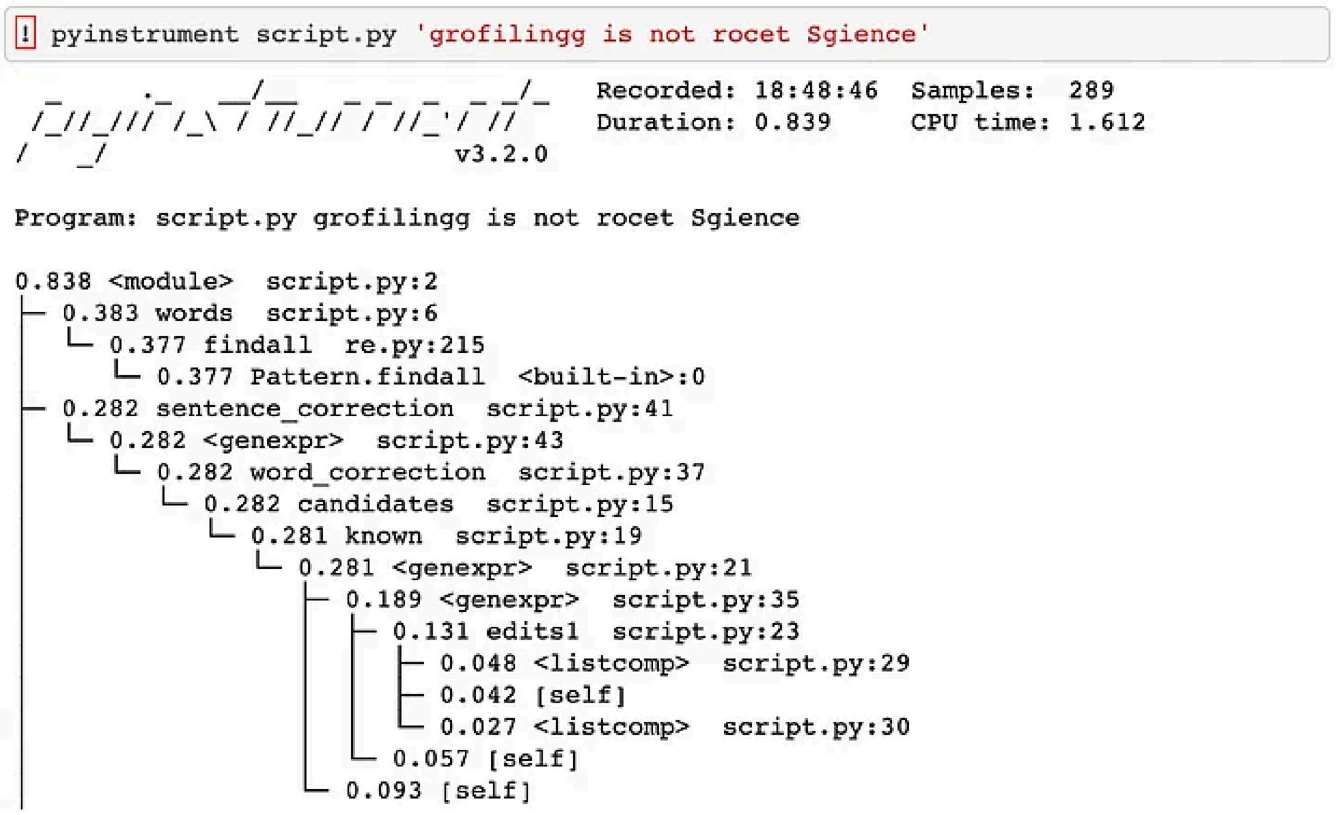
Online Profilers

Online profilers help us to understand the behavior of our program by sampling the program execution stack periodically. The idea behind it is that if a function is cumulatively slow, it will show up often, and if a function is fast, we won't see it at all. Because they are executed periodically, they are non-deterministic and add marginal overhead, which makes them more suitable for production use. We can control the accuracy by increasing the sample interval and adding more overhead.



We use the setitimer system call to sends the signal every X milliseconds to our Python program. Python lets us specify a signal handler to record our stack.

I am going to only cover `pyinstrument`, which measures the process CPU by sampling and recording statistics every one millisecond. It not built-in for Python, so installation is required, and it does not come with IPython magic.



```
! pyinstrument script.py 'grofilingg is not rocet Sgience'
  _     ._   __/__   _ _  _  _ _/_   Recorded: 18:48:46  Samples:  289
 /_//_/// /_\ / //_// / //_'/ //     Duration: 0.839     CPU time: 1.612
/   _/                      v3.2.0

Program: script.py grofilingg is not rocet Sgience

0.838 <module>  script.py:2
├─ 0.383 words  script.py:6
│  └─ 0.377 findall  re.py:215
│     └─ 0.377 Pattern.findall  <built-in>:0
├─ 0.282 sentence_correction  script.py:41
│  └─ 0.282 <genexpr>  script.py:43
│     └─ 0.282 word_correction  script.py:37
│        └─ 0.282 candidates  script.py:15
│           └─ 0.281 known  script.py:19
│              └─ 0.281 <genexpr>  script.py:21
│                 ├─ 0.189 <genexpr>  script.py:35
│                 │  ├─ 0.131 edits1  script.py:23
│                 │  │  ├─ 0.048 <listcomp>  script.py:29
│                 │  │  ├─ 0.042 [self]
│                 │  │  └─ 0.027 <listcomp>  script.py:30
│                 │  └─ 0.057 [self]
│                 └─ 0.093 [self]
│
```

We can see our sentence_correction function took 0.28 seconds. All of it spent on the candidate function, and when we drill down the edits1 takes 0.1 seconds (40%). We can see that we have much less function (and noise) in the hierarchy thanks to the statistical nature of pyinstrument.

There are many more online profilers out, and the online profiling landscape looks as follows:

Profiler	Metric	Granularity	Built-in	Output	Compatibility	Comments
pyinstrument	CPU	Function	❌	Flame/Text	🐧 / 🍎 / Windows	
python-flamegraph	CPU	Function	❌	Flame	🐧 / 🍎 / Windows	
pyspy	CPU	Function	❌	Text	🐧 / 🍎	Work on running proccess
vmprof	CPU	Line	❌	Text	🐧 / 🍎 / Windows	
austin	CPU/Memory	Function	❌	Flame/Text	🐧	Hard installation
stacksampler	Memory	Function	❌	Flame	🐧 / 🍎 / Windows	

The online profiling landscape

Online profilers have the following pros and cons

- They allow us to pinpoint the bottlenecks.
- They are not deterministic.
- Still, they introduce overhead (marginal).
- They are really easy to use.