LET'S BUILD A PYTHON PROFILER IN 25 LOC!

by Noam Elfanbaum

ABOUT ME

- Data Engineering lead at Bluevine
- Help organise PywebIL & Pycon Israel
- Find me online at @noamelf & noamelf.com

WHAT IS A PROGRAM PROFILE?

> *A profile is a set of statistics that describes how often and for how long various parts of the program executed.*

PYTHON'S (STDLIB) PROFILERS

- Python has 2 builtin profilers in stdlib:
  - profile - an early pure Python implementation
  - cProfile - a C extended profiler for better performance

Profiling demo

```python
# profiling_demo.py
def super_power(x):
    return x ** x ** x


def count_digits(num):
    return len(str(num))


result = super_power(7)
digit_count = count_digits(result)
print(f'Num of digits: {digit_count}')
```

Optimized demo

```python
# optimized_demo.py
import math


def super_power(x):
    return x ** x ** x


def count_digits(num):
    return int(math.log10(num)) + 1


result = super_power(7)
digit_count = count_digits(result)
print(digit_count)
```

IT'S MAGIC! RIGHT?

- When inside a Python program you have pretty easy access to its stack.
- Most profilers run as part of your Python process.

Accessing the process call stack

```python
# stack_access.py
import sys
import traceback

def show_stack():
    for _, call_stack in sys._current_frames().items():
        for frame in traceback.extract_stack(call_stack):
            print(f'{frame.filename}:{frame.lineno}:'
                  f'{frame.name} - "{frame.line}"')

def bar():
    show_stack()

bar()
```

TYPES OF TRIGGERS -> TYPE OF PROFILES

- There are two types of profilers that differ upon their triggers:
    - Deterministic profilers - triggered on function/line called (like profile/cProfile)
    - Statistical profilers - triggered on a time interval

HOW DO DETERMINISTIC PROFILERS WORK?

- Python let you specify a callback that gets run when interpreter events happen:
  - `sys.setprofile` - triggered when a function or a line of code is called
  - `sys.settrace` - triggered only when a function is called
- When the callback gets called, it records the stack for later analysis.

Using setprofile

```python
# setprofile.py
import sys
import traceback


def profiler(call_stack, event, arg):
    line = traceback.extract_stack(call_stack)[0]
    print(f'event: {event} | arg: {arg} | line: {line.line}')


sys.setprofile(profiler)
print('Hello world!')
```

HOW DO STATISTICAL PROFILERS WORK?

- Statistical profilers sample the program on a given interval.
- One way to implement the sampling is to ask the OS kernel to interrupt the program on a given interval.

Using OS signals to trigger sampling

```
# statistical_sampling.py
import atexit, signal, traceback

def print_handler(signum, call_stack):
    line = traceback.extract_stack(call_stack)[0]
    print(f'line: {line.line} (signum={signum})')

def start(handler, interval=1):
    signal.signal(signal.SIGPROF, handler)
    signal.setitimer(signal.ITIMER_PROF, interval, interval)
    atexit.register(lambda: signal.setitimer(
        signal.ITIMER_PROF, 0))

start(print_handler)
result = 7 ** 7 ** 7
print(len(str(result)))
```

WHEN TO USE WHICH PROFILER?

- Statistical profilers:
    - Low, controllable and predicted overhead is possible by optimizing the sampling interval
    - Less accurate result since it, by design, misses function/line calls.
    - More suitable for continuous, low impact production monitoring.

- Deterministic profilers:
    - Introduces a fixed amount of latency for every function call / line of code executed.
    - Collects the exact program execution stack
    - More suitable for interactive/local debugging.

NOW, LET'S BUILD A (NAIVE) STATISTICAL PROFILER IN 25 LOC!

Statistical Flame graph Profiler

```python
import atexit
import collections
import signal
import traceback

stats = collections.defaultdict(int)


def _sample(_, call_stack):
    stack = traceback.extract_stack(call_stack)
    formatted_stack = ';'.join(line.line for line in stack)
    stats[formatted_stack] += 1


def start(interval=0.005):
    signal.signal(signal.SIGPROF, _sample)
```

Simple usage of sfProfiler

```python
import sfProfiler


def calc(x):
    return x ** x


def main():
    calc(100_000)
    calc(200_000)


sfProfiler.start()
main()
print(sfProfiler.format_stats())
```

A more complex example

```python
import sys
import tempfile
import urllib.request

import sfProfiler

def save_pep(pep):
    url = f'https://www.python.org/dev/peps/pep-{pep:04}'
    with urllib.request.urlopen(url) as request:
        html = request.read()
    with tempfile.TemporaryFile('wb') as f:
        f.write(html)


def main():
    for pep in range(4):
```

REFERENCE

- Brendan Gregg's Flame Graph tool
- Juila Evans post about Ruby and Python profilers
- Nylas performance post where they explain how they built a homemade performance monitoring service (recommended!).
- Python's profilers docs
- This talk can be found on Github

Thanks!