파이썬 프로파일러

소스 코드: Lib/profile.py 및 Lib/pstats.py

프로파일러 소개

cProfile과 profile은 파이썬 프로그램의 *결정론적 프로파일링 (deterministic profiling)*을 제공합니다. *프로파일 (profile)*은 프로그램의 여러 부분이 얼마나 자주 그리고 얼마나 오랫동안 실행되었는지를 기술하는 통계 집합입니다. 이러한 통계는 pstats 모듈을 통해 보고서로 포매팅 할 수 있습니다.

파이썬 표준 라이브러리는 같은 프로파일링 인터페이스의 두 가지 구현을 제공합니다:

1. cProfile이 대부분 사용자에게 권장됩니다; 오래 실행되는 프로그램을 프로파일링하는 데 적합한 합리적인 부하를 주는 C 확장입니다. Brett Rosen과 Ted Czotter가 제공한 lsprof를 기반으로 합니다.
2. profile은 순수 파이썬 모듈이고 이 인터페이스를 cProfile이 모방했습니다. 하지만, 프로파일링 되는 프로그램에 상당한 부하를 추가합니다. 어떤 방식으로 프로파일러를 확장하려고 한다면, 이 모듈을 사용하면 작업이 더 쉬울 수 있습니다. Jim Roskind가 원래 설계하고 작성했습니다.

참고: 프로파일러 모듈은 벤치마킹 목적(이를 위해서는 합리적으로 정확한 결과를 주는 timeit이 있습니다)이 아니라 주어진 프로그램에 대한 실행 프로파일을 제공하도록 설계되었습니다. 이것은 특히 C 코드에 대한 파이썬 코드 벤치마킹에 적용됩니다: 프로파일러는 파이썬 코드에 부하를 가하지만, C 수준 함수에는 그렇지 않아서 C 코드는 모든 파이썬 코드보다 빨라 보입니다.

즉석 사용자 설명서

이 섹션은 "설명서를 읽고 싶지 않은" 사용자를 위해 제공됩니다. 매우 간단한 개요를 제공하며, 사용자가 기존 응용 프로그램에서 프로파일링을 빠르게 수행할 수 있도록 합니다.

단일 인자를 취하는 함수를 프로파일링하려면, 이렇게 할 수 있습니다:

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")')
```

(시스템에서 cProfile을 사용할 수 없으면 대신 profile을 사용하십시오.)

위의 작업은 re.compile()을 실행하고 다음과 같은 프로파일 결과를 인쇄합니다:

```
         197 function calls (192 primitive calls) in 0.002 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.001    0.001 <string>:1(<module>)
        1    0.000    0.000    0.001    0.001 re.py:212(compile)
        1    0.000    0.000    0.001    0.001 re.py:268(_compile)
        1    0.000    0.000    0.000    0.000 sre_compile.py:172(_compile_charset)
        1    0.000    0.000    0.000    0.000 sre_compile.py:201(_optimize_charset)
        4    0.000    0.000    0.000    0.000 sre_compile.py:25(_identityfunction)
      3/1    0.000    0.000    0.000    0.000 sre_compile.py:33(_compile)
```

첫 번째 줄은 197개의 호출이 관찰되었음을 나타냅니다. 이 호출 중 192개는 *프리미티브(primitive)*였으며, 이는 호출이 재귀를 통해 유발되지 않았음을 의미합니다. 다음 줄: Ordered by: standard name 은 가장 오른쪽 열의 텍스트

ncalls
: 호출 수.

tottime
: 주어진 함수에서 소비된 총 시간 (서브 함수 호출에 든 시간은 제외합니다)

percall
: tottime을 ncalls로 나눈 몫

cumtime
: 이 함수와 모든 서브 함수에서 소요된 누적 시간 (호출에서 종료까지). 이 수치는 재귀 함수에서도 *정확*합니다.

percall
: cumtime을 프리미티브 호출로 나눈 몫

filename:lineno(function)
: 각 함수의 해당 데이터를 제공합니다 – 파일명:줄 번호(함수)

첫 번째 열에 두 개의 숫자가 있으면 (예를 들어 3/1), 함수가 재귀 되었음을 의미합니다. 두 번째 값은 프리미티브 호출 수이고 앞엣것은 총 호출 수입니다. 함수가 재귀 되지 않으면, 이 두 값은 같으며, 한 숫자만 인쇄됩니다.

프로파일 실행의 끝에 출력을 인쇄하는 대신, run() 함수에 파일명을 지정하여 결과를 파일에 저장할 수 있습니다:

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")', 'restats')
```

pstats.Stats 클래스는 파일에서 프로파일 결과를 읽고 다양한 방식으로 포맷합니다.

cProfile과 profile을 스크립트로 호출하여 다른 스크립트를 프로파일링 할 수도 있습니다. 예를 들면:

```
python -m cProfile [-o output_file] [-s sort_order] (-m module | myscript.py)
```

-o는 stdout 대신 파일에 프로파일 결과를 씁니다.

-s는 출력을 정렬할 sort_stats() 정렬 값 중 하나를 지정합니다. 이는 -o가 제공되지 않은 경우에만 적용됩니다.

-m은 스크립트 대신 모듈이 프로파일링 되도록 지정합니다.

버전 3.7에 추가: -m 옵션을 cProfile에 추가했습니다.

버전 3.8에 추가: -m 옵션을 profile에 추가했습니다.

pstats 모듈의 Stats 클래스에는 프로파일 결과 파일에 저장된 데이터를 조작하고 인쇄하기 위한 다양한 메서드가 있습니다:

```
import pstats
from pstats import SortKey
p = pstats.Stats('restats')
p.strip_dirs().sort_stats(-1).print_stats()
```

strip_dirs() 메서드는 모든 모듈 이름에서 외부 경로를 제거했습니다. sort_stats() 메서드는 인쇄되는 표준 모듈/줄/이름 문자열에 따라 모든 항목을 정렬했습니다. print_stats() 메서드는 모든 통계를 인쇄했습니다. 다음과 같은 정렬 호출을 시도할 수 있습니다:

```
p.sort_stats(SortKey.NAME)
p.print_stats()
```

첫 번째 호출은 실제로 함수 이름으로 목록을 정렬하고, 두 번째 호출은 통계를 인쇄합니다. 다음은 몇 가지 흥미로운 실험 호출입니다:

```
p.sort_stats(SortKey.CUMULATIVE).print_stats(10)
```

이것은 함수에서의 누적 시간을 기준으로 프로파일을 정렬한 다음, 가장 중요한 10개의 줄만 인쇄합니다. 시간이 걸리는 알고리즘을 이해하려면, 위의 줄을 사용하십시오.

어떤 함수가 많이 반복되고 많은 시간이 걸리는지 알고 싶다면, 다음을 수행하여:

```
p.sort_stats(SortKey.TIME).print_stats(10)
```

각 함수 내에서 소비한 시간에 따라 정렬한 다음, 상위 10개 함수에 대한 통계를 인쇄하십시오.

다음과 같은 것도 시도해 볼 수 있습니다:

```
p.sort_stats(SortKey.FILENAME).print_stats('__init__')
```

이렇게 하면 모든 통계가 파일 이름으로 정렬된 다음, 클래스 초기화(init) 메서드에 대한 통계만 인쇄됩니다 (이들의 철자가 __init__이기 때문입니다). 마지막 예로, 다음을 시도해 볼 수 있습니다:

```
p.sort_stats(SortKey.TIME, SortKey.CUMULATIVE).print_stats(.5, 'init')
```

이 줄은 주 시간 키와 누적 시간 보조 키로 통계를 정렬한 다음, 일부 통계를 인쇄합니다. 구체적으로, 목록을 먼저 원래 크기의 50%(.5)로 줄인 다음, init를 포함하는 줄만 유지되고, 그 서브 서브 목록이 인쇄됩니다.

어떤 함수가 위의 함수를 호출했는지 궁금하다면, 이제 다음과 같이 할 수 있습니다 (p는 여전히 마지막 기준에 따라 정렬됩니다):

```
p.print_callers(.5, 'init')
```

그러면 나열된 각 함수에 대한 호출자 목록을 얻습니다.

더 많은 기능을 원하면, 매뉴얼을 읽거나, 다음 함수가 무엇인지 추측하십시오:

```
p.print_callees()
p.add('restats')
```

스크립트로 호출될 때, pstats 모듈은 프로파일 덤프를 읽고 검사하기 위한 통계 브라우저입니다. 간단한 줄 지향 인터페이스(cmd를 사용하여 구현되었습니다)와 대화식 도움말이 있습니다.

profile과 cProfile 모듈 레퍼런스

profile과 cProfile 모듈은 모두 다음 함수를 제공합니다:

profile.**run**(*command*, *filename=None*, *sort=- 1*)

이 함수는 exec() 함수에 전달할 수 있는 단일 인자와 선택적 파일 이름을 취합니다. 모든 경우에 이 루틴은 다음을 실행합니다:

```
exec(command, __main__.__dict__, __main__.__dict__)
```

그리고 실행으로부터 프로파일링 통계를 수집합니다. 파일 이름이 없으면, 이 함수는 자동으로 Stats 인스턴스를 만들고 간단한 프로파일링 보고서를 인쇄합니다. 정렬 값이 지정되면, 이 Stats 인스턴스로 전달되어 결과 정렬 방법을 제어합니다.

profile.**runctx**(*command*, *globals*, *locals*, *filename=None*, *sort=-1*)

이 함수는 run()과 유사하며, ***command*** 문자열에 대한 전역(globals)과 지역(locals) 딕셔너리를 제공하기 위한 인자가 추가되었습니다. 이 루틴은 다음을 실행합니다:

```
exec(command, globals, locals)
```

그리고 위의 run() 함수에서와같이 프로파일링 통계를 수집합니다.

class profile.**Profile**(*timer=None*, *timeunit=0.0*, *subcalls=True*, *builtins=True*)

이 클래스는 일반적으로 cProfile.run() 함수가 제공하는 것보다 프로파일링에 대한 더 세밀한 제어가 필요할 때만 사용됩니다.

timer 인자를 통해 코드를 실행하는 데 걸리는 시간을 측정하기 위한 사용자 정의 타이머를 제공할 수 있습니다. 현재 시각을 나타내는 단일 숫자를 반환하는 함수여야 합니다. 숫자가 정수이면, ***timeunit***는 각 시간 단위의 지속 시간을 지정하는 승수를 지정합니다. 예를 들어, 타이머가 밀리초 단위로 측정된 시간을 반환하면 시간 단위는 .001 입니다.

Profile 클래스를 직접 사용하면 프로파일 데이터를 파일에 쓰지 않고도 프로파일 결과를 포맷할 수 있습니다:

```
import cProfile, pstats, io
from pstats import SortKey
pr = cProfile.Profile()
pr.enable()
# ... do something ...
pr.disable()
s = io.StringIO()
sortby = SortKey.CUMULATIVE
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
ps.print_stats()
print(s.getvalue())
```

Profile 클래스는 컨텍스트 관리자로도 사용될 수 있습니다 (cProfile 모듈에서만 지원됩니다. 컨텍스트 관리자 형을 참조하십시오):

```
import cProfile

with cProfile.Profile() as pr:
    # ... do something ...

    pr.print_stats()
```

버전 3.8에서 변경: 컨텍스트 관리자 지원이 추가되었습니다.

enable()

프로파일링 데이터 수집을 시작합니다. cProfile에만 있습니다.

disable()

프로파일링 데이터 수집을 중지합니다. cProfile에만 있습니다.

create_stats()

프로파일링 데이터 수집을 중지하고 결과를 내부적으로 현재 프로파일로 기록합니다.

print_stats(*sort=-1*)

현재 프로파일을 기반으로 Stats 객체를 만들고 결과를 stdout에 인쇄합니다.

현재 프로파일의 결과를 *filename*에 씁니다.

run(*cmd*)

exec()를 통해 cmd를 프로파일 합니다.

runctx(*cmd*, *globals*, *locals*)

지정된 전역과 지역 환경으로 exec()를 통해 cmd를 프로파일 합니다.

runcall(*func*, */*, **args*, ***kwargs*)

func(*args, **kwargs)를 프로파일 합니다

프로파일링은 호출된 명령/함수가 실제로 반환하는 경우에만 작동함에 유의하십시오. 인터프리터가 종료되면 (예를 들어 호출된 명령/함수 실행 중 sys.exit() 호출을 통해) 아무런 프로파일링 결과도 인쇄되지 않습니다.

Stats 클래스

프로파일러 데이터의 분석은 Stats 클래스를 사용하여 수행됩니다.

class pstats.**Stats**(**filenames or profile*, *stream=sys.stdout*)

이 클래스 생성자는 *filename*(또는 파일명의 리스트)이나 Profile 인스턴스에서 "통계 객체"의 인스턴스를 만듭니다. 출력은 *stream*에 의해 지정된 스트림으로 인쇄됩니다.

위의 생성자에 의해 선택된 파일은 해당 버전의 profile이나 cProfile에 의해 만들어졌어야 합니다. 구체적으로, 이 프로파일러의 향후 버전에서 보장되는 파일 호환성은 *없으며*, 다른 프로파일러에서 생성된 파일이나 다른 운영 체제에서 실행되는 같은 프로파일러의 실행과 호환되지 않습니다. 여러 파일이 제공되면, 동일한 함수에 대한 모든 통계가 통합되므로, 여러 프로세스에 대한 전체 뷰를 단일 보고서에서 고려할 수 있습니다. 추가 파일을 기존 Stats 객체의 데이터와 결합해야 하면, add() 메서드를 사용할 수 있습니다.

파일에서 프로파일 데이터를 읽는 대신, cProfile.Profile이나 profile.Profile 객체를 프로파일 데이터 소스로 사용할 수 있습니다.

Stats 객체에는 다음과 같은 메서드가 있습니다:

strip_dirs()

Stats 클래스에 대한 이 메서드는 파일 이름에서 모든 선행 경로 정보를 제거합니다. 80열 이내에 (가깝게) 맞게 출력물의 크기를 줄이는 데 매우 유용합니다. 이 메서드는 객체를 수정하고, 제거된 정보는 손실됩니다. 제거 조작을 수행한 후, 객체는 객체 초기화와 로드 직후와 마찬가지로 "임의의" 순서로 항목을 가진 것으로 간주합니다. strip_dirs()로 인해 두 함수 이름이 구별할 수 없게 되면 (같은 파일 이름의 같은 줄에 있고, 함수 이름도 같습니다), 이 두 항목에 대한 통계는 단일 항목으로 누적됩니다.

add(**filenames*)

Stats 클래스의 이 메서드는 추가 프로파일링 정보를 현재 프로파일링 객체에 누적합니다. 인자는 해당 버전의 profile.run()이나 cProfile.run()으로 만들어진 파일명을 참조해야 합니다. 동일한 이름을 가진 (파일, 줄, 이름) 함수에 대한 통계는 자동으로 단일 함수 통계에 축적됩니다.

dump_stats(*filename*)

Stats 객체에 로드된 데이터를 *filename*이라는 파일에 저장합니다. 파일이 없으면 만들어지고, 이미 존재하면 덮어씁니다. 이것은 profile.Profile과 cProfile.Profile 클래스에 있는 같은 이름의 메서드와 동등합니다.

sort_stats(**keys*)

이 메서드는 제공된 기준에 따라 Stats 객체를 정렬하여 수정합니다. 인자는 정렬 기준을 식별하는 문자열

둘 이상의 키가 제공되면, 그 앞에 선택된 모든 키가 같을 때 추가 키가 보조 기준으로 사용됩니다. 예를 들어, sort_stats(SortKey.NAME, SortKey.FILE)은 함수 이름에 따라 모든 항목을 정렬하고, 함수 이름이 같으면 파일 이름으로 정렬합니다.

문자열 인자의 경우, 약어가 모호하지 않은 한, 모든 키 이름에 약어를 사용할 수 있습니다.

유효한 문자열과 SortKey는 다음과 같습니다:

유효한 문자열 인자	유효한 열거형 인자	의미
'calls'	SortKey.CALLS	호출 수
'cumulative'	SortKey.CUMULATIVE	누적 시간
'cumtime'	해당 없음	누적 시간
'file'	해당 없음	파일 이름
'filename'	SortKey.FILENAME	파일 이름
'module'	해당 없음	파일 이름
'ncalls'	해당 없음	호출 수
'pcalls'	SortKey.PCALLS	프리미티브 호출 수
'line'	SortKey.LINE	줄 번호
'name'	SortKey.NAME	함수 이름
'nfl'	SortKey.NFL	이름/파일/줄
'stdname'	SortKey.STDNAME	표준 이름
'time'	SortKey.TIME	내부 시간
'tottime'	해당 없음	내부 시간

통계의 모든 정렬은 내림차순이고 (가장 시간이 오래 걸리는 항목을 앞에 놓습니다), 이름, 파일 및 줄 번호 검색은 오름차순(알파벳 순서)임에 유의하십시오. SortKey.NFL과 SortKey.STDNAME 간의 미묘한 차이점은 표준 이름이 인쇄된 이름의 일종이라는 것입니다. 즉, 포함된 줄 번호가 이상한 방식으로 비교됩니다. 예를 들어, 줄 3, 20 및 40은 (파일 이름이 같으면) 문자열 순서 20, 3 및 40으로 나타납니다. 반면에 SortKey.NFL은 줄 번호를 숫자로 비교합니다. 실제로, sort_stats(SortKey.NFL)은 sort_stats(SortKey.NAME, SortKey.FILENAME, SortKey.LINE)과 같습니다.

이전 버전과의 호환성을 위해, 숫자 인자 -1, 0, 1 및 2가 허용됩니다. 이들은 각각 'stdname', 'calls', 'time' 및 'cumulative'로 해석됩니다. 이 이전 스타일 형식(숫자)을 사용하면, 오직 하나의 정렬 키(숫자키)만 사용되며, 추가 인자는 조용히 무시됩니다.

버전 3.7에 추가: SortKey 열거형을 추가했습니다.

reverse_order()

Stats 클래스에 대한 이 메서드는 객체 내 기본 리스트의 순서를 뒤집습니다. 기본적으로 오름차순 대 내림차순은 선택한 정렬 키에 따라 올바르게 선택됨에 유의하십시오.

print_stats(*restrictions)

Stats 클래스에 대한 이 메서드는 profile.run() 정의에 설명된 대로 보고서를 인쇄합니다.

인쇄 순서는 객체에서 수행된 마지막 sort_stats() 연산을 기반으로 합니다 (add()와 strip_dirs()의 경고가

(있다면) 제공된 인자를 사용하여 목록을 중요한 항목으로 줄일 수 있습니다. 처음에는, 목록이 전체 프로파일링 된 함수 집합이 됩니다. 각 제한(restriction)은 정수(줄 수 선택)나 0.0과 1.0을 포함하고 그사이의 십진 소수(줄의 백분율을 선택), 또는 정규식으로 해석되는 문자열(인쇄되는 표준 이름과 일치하는 패턴)입니다. 여러 제한이 제공되면, 순차적으로 적용됩니다. 예를 들면:

```
print_stats(.1, 'foo:')
```

는 먼저 인쇄를 목록의 처음 10%로 제한한 다음, 파일 이름 .*foo:의 일부인 함수만 인쇄합니다. 대조적으로, 명령:

```
print_stats('foo:', .1)
```

은 파일 이름이 .*foo: 인 모든 함수로 목록을 제한한 다음, 그중 처음 10%만 인쇄합니다.

print_callers(*restrictions*)

Stats 클래스에 대한 이 메서드는 프로파일 된 데이터베이스에 있는 각 함수를 호출한 모든 함수의 목록을 인쇄합니다. 순서는 print_stats()에서 제공한 순서와 동일하며, 제한 인자의 정의도 동일합니다. 각 호출자는 개별 줄로 보고됩니다. 통계를 생성한 프로파일러에 따라 형식이 약간 다릅니다:

- profile을 사용하면, 각 호출자 뒤에 숫자가 괄호 안에 표시되어 이 특정 호출이 몇 번이나 되었는지 표시됩니다. 편의를 위해, 두 번째 괄호로 묶지 않은 숫자는 오른쪽에 나오는 함수에서 소비한 누적 시간을 반복합니다.
- cProfile을 사용하면, 각 호출자 앞에 세 숫자가 나옵니다: 이 특정 호출이 발생한 횟수, 이 특정 호출자가 호출한 동안 현재 함수에서 소비 한 총 및 누적 시간.

print_callees(*restrictions*)

Stats 클래스에 대한 이 메서드는 표시된 함수에 의해 호출된 모든 함수의 목록을 인쇄합니다. 이러한 호출 방향 반전(호출한 대 호출된)을 제외하고, 인자와 순서는 print_callers() 메서드와 같습니다.

get_stats_profile()

이 메서드는 함수 이름을 FunctionProfile 인스턴스로 매핑하는 StatsProfile 인스턴스를 반환합니다. 각 FunctionProfile 인스턴스에는 함수 실행 시간, 호출 횟수 등의 함수의 프로파일 관련 정보가 있습니다.

버전 3.9에 추가: 다음과 같은 데이터 클래스(dataclasses)를 추가했습니다: StatsProfile, FunctionProfile. 다음과 같은 함수를 추가했습니다: get_stats_profile.

결정론적 프로파일링이란 무엇입니까?

*결정론적 프로파일링(Deterministic profiling)*이라는 용어는 모든 *함수 호출*, *함수 반환* 및 *예외* 이벤트가 모니터링되고, (사용자 코드가 실행되는 시간 동안) 이러한 이벤트 사이의 간격에 대한 정확한 시간 측정이 이루어진다는 사실을 반영하기 위한 것입니다. 반면에, *통계적 프로파일링(statistical profiling)*(이 모듈에서는 수행하지 않습니다)은 유효 명령어 포인터를 무작위로 샘플링하여 시간이 소비되는 위치를 추론합니다. 후자의 기술은 전통적으로 (코드를 계측할 필요가 없기 때문에) 오버헤드가 적지만, 시간이 어디에서 소비되는지에 대한 상대적 표시만 제공합니다.

파이썬에서는, 실행 중에 인터프리터가 활성화되어있어서, 결정론적 프로파일링을 수행하기 위해 인스트루먼트 된 코드(instrumented code)가 필요하지 않습니다. 파이썬은 각 이벤트에 대해 자동으로 *훅(hook)*(선택적 콜백)을 제공합니다. 또한, 파이썬의 인터프리터 적인 성격은 실행에 이미 많은 오버헤드를 추가하는 경향이 있어서, 결정론적 프로파일링은 일반적인 응용 프로그램에서 작은 처리 오버헤드만 추가하는 경향이 있습니다. 결과적으로 결정론적 프로파일링은 그다지 비싸지 않으면서도, 파이썬 프로그램의 실행에 대한 광범위한 실행 시간 통계를 제공합니다.

호출 수 통계를 사용하여 코드의 버그(놀랄만한 횟수)를 식별하고, 가능한 인라인 확장 지점(높은 호출 횟수)을 식별할 수 있습니다. 내부 시간 통계를 사용하여 신중하게 최적화해야 하는 “핫 루프(hot loops)”를 식별할 수 있습니다. 누적

한계

타이밍 정보의 정확성과 관련하여 한 가지 제약이 있습니다. 정확성과 관련해서는 결정론적 프로파일러에 근본적인 문제가 있습니다. 가장 명백한 제약은 하부 “시계”가 (일반적으로) 약 .001 초의 속도로만 눈금이 변하는 것입니다. 따라서 하부 시계보다 더 정확한 측정은 없습니다. 충분한 측정을 수행하면, “에러”가 평균이 되어 사라지는 경향이 있습니다. 불행히도, 이 첫 번째 에러를 제거하면 두 번째 에러 원인이 발생합니다.

두 번째 문제는 이벤트가 디스패치 된 시점부터 프로파일러가 시간을 얻기 위해 호출하는 것이 실제로 시계의 상태를 *얻기*까지 “시간이 걸린다”는 것입니다. 마찬가지로, 프로파일러 이벤트 핸들러를 빠져나갈 때 시계값이 획득된 (그런 다음 저장됩니다) 시간부터, 사용자의 코드가 다시 실행될 때까지 어떤 지연이 발생합니다. 결과적으로, 여러 번 호출되거나, 많은 함수를 호출하는 함수는 일반적으로 이 에러를 누적합니다. 이러한 방식으로 누적되는 에러는 일반적으로 시계 정확도(1 눈금 미만)보다 작지만, 누적될 수 있어서 매우 중요해집니다.

오버헤드가 낮은 cProfile보다 profile에서 이 문제가 더 중요합니다. 이러한 이유로, profile은 특정 플랫폼에 대해 자신을 보정하는 방법을 제공하여 이 에러를 확률적으로 (평균적으로) 제거할 수 있습니다. 프로파일러가 보정된 후에는, 더 정확하지만 (최소한 최소 자승의 의미에서), 때로는 음수를 생성합니다 (호출 횟수가 예외적으로 낮고, 확률의 신들이 당신에게 등을 돌리면 :-).) 프로파일에서 음수를 보아도 너무 놀라지 *마십시오*. 프로파일러를 보정한 경우에* 만* 나타나야 하며, 결과는 실제로 보정하지 않은 것보다 낫습니다.

보정

profile 모듈의 프로파일러는 각 이벤트 처리 시간에서 상수를 빼서 시간 함수 호출의 오버헤드를 보상하고, 결과를 잘 보관합니다. 기본적으로, 상수는 0입니다. 다음 절차는 주어진 플랫폼에 대해 더 나은 상수를 얻는 데 사용될 수 있습니다 (한계를 참조하십시오).

```
import profile
pr = profile.Profile()
for i in range(5):
    print(pr.calibrate(10000))
```

The method executes the number of Python calls given by the argument, directly and again under the profiler, measuring the time for both. It then computes the hidden overhead per profiler event, and returns that as a float. For example, on a 1.8Ghz Intel Core i5 running macOS, and using Python’s time.process_time() as the timer, the magical number is about 4.04e-6.

이 반복의 목적은 상당히 일관된 결과를 얻는 것입니다. 컴퓨터가 *매우* 빠르거나, 타이머 함수의 해상도가 좋지 않으면, 일관된 결과를 얻기 위해 100000이나 심지어 1000000을 전달해야 할 수 있습니다.

일관된 대답을 얻었을 때, 세 가지 방법으로 사용할 수 있습니다:

```
import profile

# 1. Apply computed bias to all Profile instances created hereafter.
profile.Profile.bias = your_computed_bias

# 2. Apply computed bias to a specific Profile instance.
pr = profile.Profile()
pr.bias = your_computed_bias

# 3. Specify computed bias in instance constructor.
pr = profile.Profile(bias=your_computed_bias)
```

선택해야 한다면, 더 작은 상수를 선택하는 것이 좋습니다, 그러면 결과가 프로파일 통계에서 결과가 “덜 자주” 음수로 표시됩니다.

현재 시각을 결정하는 방법을 변경하려면 (예를 들어, 벽시계 시간이나 소요된 프로세스 시간을 사용하도록 만들려면), Profile 클래스 생성자에게 원하는 타이밍 함수를 전달합니다:

```
pr = profile.Profile(your_time_func)
```

결과 프로파일러는 your_time_func를 호출합니다. profile.Profile이나 cProfile.Profile 중 어느 것을 사용하느냐에 따라, your_time_func의 반환 값은 다르게 해석됩니다:

profile.Profile
: your_time_func는 단일 숫자를 반환하거나, 또는 (os.times()가 반환하는 것과 같이) 합계가 현재 시각인 숫자 리스트를 반환해야 합니다. 함수가 단일 시간 숫자를 반환하거나, 반환된 숫자의 리스트 길이가 2이면, 특히 빠른 버전의 디스패치 루틴을 얻게 됩니다.

  여러분이 선택한 타이머 함수에 대해 프로파일러 클래스를 보정해야 합니다 (보정을 참조하십시오). 대부분의 기계에서, 단일 정숫값을 반환하는 타이머는 프로파일링 중 낮은 오버헤드 측면에서 최상의 결과를 제공합니다. (os.times()는 부동 소수점 값의 튜플을 반환하므로 *꽤* 나쁩니다). 더 좋은 타이머를 가장 깨끗한 방식으로 대체하려면, 클래스를 파생하고 적절한 보정 상수와 함께 타이머 호출을 가장 잘 처리하는 대체 디스패치 메서드를 배선하십시오.

cProfile.Profile
: your_time_func는 단일 숫자를 반환해야 합니다. 정수를 반환하면, 시간 단위의 실제 지속 시간을 지정하는 두 번째 인자로 클래스 생성자를 호출할 수도 있습니다. 예를 들어, your_integer_time_func가 밀리초 단위로 측정된 시간을 반환하면, 다음과 같이 Profile 인스턴스를 구성합니다:

  ```
  pr = cProfile.Profile(your_integer_time_func, 0.001)
  ```

  cProfile.Profile 클래스를 보정할 수 없어서, 사용자 정의 타이머 함수는 주의해서 사용해야 하고 가능한 한 빨라야 합니다. 사용자 정의 타이머로 최상의 결과를 얻으려면, 내부 _lsprof 모듈의 C 소스에 하드 코딩해야 할 수도 있습니다.

파이썬 3.3은 time에 프로세스나 벽시계 시간을 정확하게 측정하는 데 사용할 수 있는 몇 가지 새로운 함수를 추가합니다. 예를 들어, time.perf_counter()를 참조하십시오.