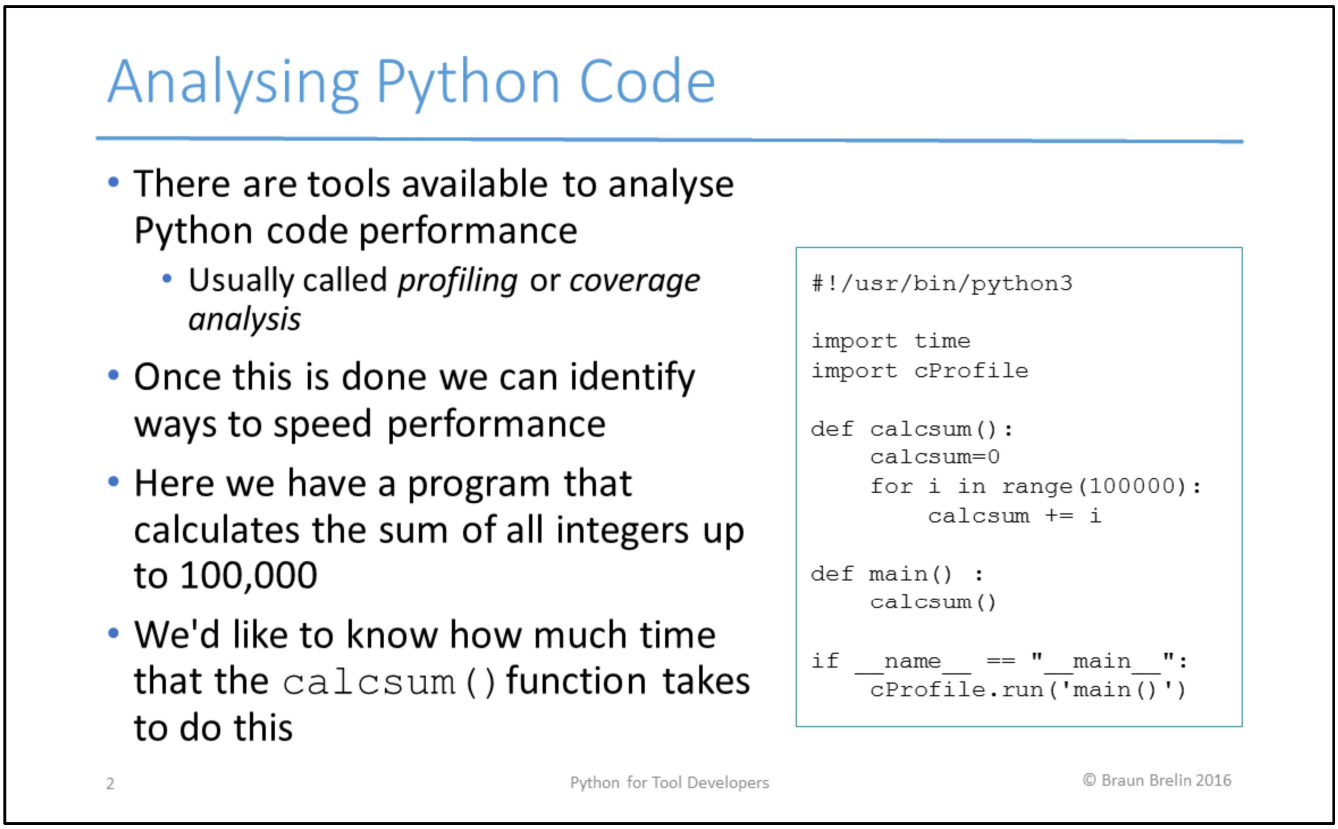
Module 7

Speeding Up Your Code



Analysing Python Code

Let's look now at how we can use some different tools to analyse your Python code with an eye towards improving performance. This is usually called *profiling* or *coverage analysis*. In this module, we'll discuss some methods to examine your code to find out where it is spending most of its time. Once we know that, we can get some ideas of where we can refactor things to speed up its performance.

Let's take a look at a simple Python program:

```
#!/usr/bin/python3

import time
import cProfile

def calcsum():
    calcsum=0
    for i in range(100000):
        calcsum += i

def main() :
    calcsum()

if __name__ == "__main__":
    cProfile.run('main()')
```

Here we have a program that calculates the sum of all integers up to 100,000. We'd like to know how much time that the `calcsum()` function takes to do this. Let's examine a couple of ways to do this.

First Approach - Timing

- The first approach is to simply time the function by using the `time` module in Python
- We take our `calcsum1` method and time the relevant code
  - In this case the `loop` iteration over 10,000 integers
  - Then summing them up manually
- Start the timer with the `start = time.time()`
- End with the `finish=time.time()`
- It is then a trivial operation to subtract the start time from the finish time and get the resultant delta
- We can even use this method to compare two different ways of doing the summation algorithm

3 Python for Tool Developers

Timing

The first approach is to simply time the function by using the time module in Python. Let's do this and take a look at our re-written program.

```
#!/usr/bin/python3

import time

def calcsum1():
    start=0
    finish=0
    calcsum = 0
    start = time.time()
    for i in range(10000):
        calcsum += 1
    finish = time.time()
    return finish - start

def main():
    timetorun = calcsum1()
    timetorun = timetorun * 100000
    print("%.2f" % (timetorun))

if __name__ == "__main__":
    main()
```

Note here that we've taken our `calcsum1` method and timed the relevant code, in this case the loop iteration over 10,000 integers and summing them up manually.

We start the timer with the `start = time.time()` and end with the `finish=time.time()`. It is then a trivial operation to subtract the start time from the finish time and get the resultant delta. We can even use this method to compare two different ways of doing the summation algorithm.

```
#!/usr/bin/python3

import time

def calcsum1():
    start=0
    finish=0
    calcsum = 0
    start = time.time()
    for i in range(10000):
        calcsum += 1
    finish = time.time()
    return finish - start

def calcsum():
    start=0
    finish=0
    start = time.time()
    calc = sum(range(10000))
    finish = time.time()
    return finish - start


def main():
    timetorun = calcsum1()
    timetorun = timetorun * 100000
    print("%.2f" % (timetorun))

    timetorun = calcsum()
    timetorun = timetorun * 100000
    print("%.2f" % (timetorun))
if __name__ == "__main__":
    main()
```

The Output

- This clearly shows that the `builtin sum` function gives far superior performance for any sort of summation

```
64.75
18.33
```

- This methodology, however, has serious flaws:
  - The Python garbage collection algorithm is also running, which may very well affect how the algorithm performs
  - We only call the function and time it once
  - It is preferable to run the function repeatedly and take an average
  - This better reflects a real world scenario where other external events may affect the runtime environment of the program

5 Python for Tool Developers

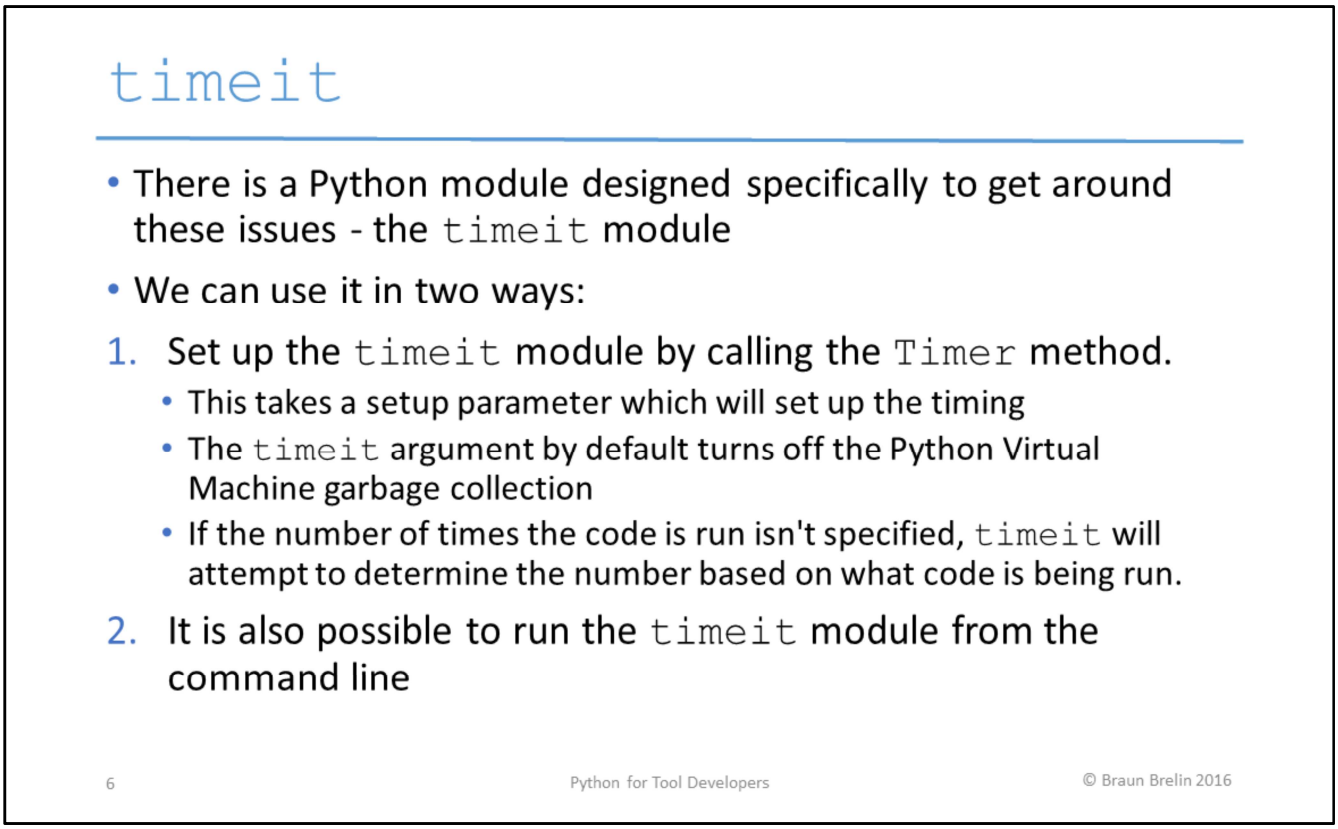
The output of this program is:

```
64.75
18.33
```

This output clearly shows that the `builtin sum` function is far superior in terms of performance for doing any sort of summation.

This methodology, however, has serious flaws. For one thing, the Python garbage collection algorithm is also running, which may very well affect how the algorithm performs. Also, we only call the function and time it once.

It's far preferable to run the function numerous times and take an average of the times. This better reflects a real world scenario where other external events may affect the runtime environment of the program.



timeit

In order to fix this, we turn to another Python module designed specifically to get around these issues, the `timeit` module. Let's take a look at some code that showcases the `timeit` module.

```
#!/usr/bin/python3
import timeit

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

def main():
    i = 5
    t = timeit.Timer(setup = 'from __main__ import fib', stmt =
'fib(5)')
    print ('Value of n = %.d\nPure python %.2f usec/pass' %
(i,t.timeit(number=100000)))

    outputs = t.repeat(number = 1000000, repeat = 3)
    for time_value in outputs:
        print ('Value of n = %.d\nPure python %.2f usec/pass' %
(i,time_value))

if __name__ == "__main__":
    main()
```

Here we see `timeit` being used in two different ways.

First, we set up the `timeit` module by calling the `Timer` method. This method takes a setup parameter which will set up the timing. The setup in this case is to import the `fib` function and make it available to our `timeit` instance. The `stmt` argument will indicate to the `timeit` module what code will be timed.

Some things to note

- The `timeit` argument by default turns off the Python Virtual Machine garbage collection. This can result in more accurate timings.
- Also, if the number of times the code is run isn't specified, `timeit` will attempt to determine the number based on what code is being run.

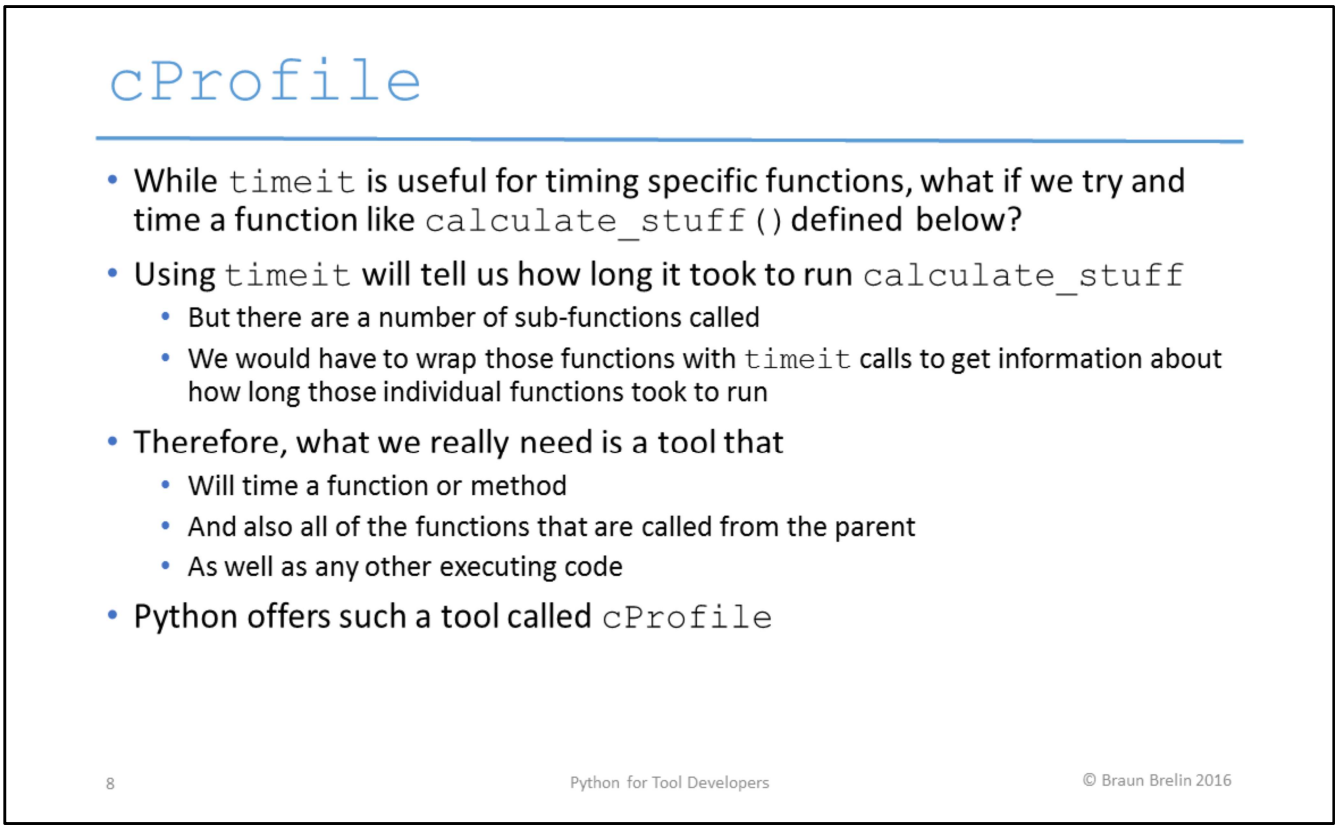
It is also possible to run the `timeit` module from the command line. Here's an example:

```
python -m timeit  -n 1000000 -r 100 '10/2.4'
```

The `-m` option tells Python to load the `timeit` module; the `-n` option tells Python to run the timed statement one million times and repeat this for one hundred times. Finally, the last line is the Python statement to run, in this case 10 divided by 2.4.

The output of this statement looks like this:

```
1000000 loops, best of 100: 0.0559 usec per loop
```



cProfile

While `timeit` is useful for timing specific functions, what if we try and time a function like `calculate_stuff()` defined below?

```
def calculate_stuff(n):
    calcsum = 0
    for i in range(10):
        calcsum += i
    if calcsum < 0:
        calcsum = abs(calcsum)
```

Using `timeit` will tell us how long it took to run `calculate_stuff`, but we see from the definition that there are a number of sub-functions that get called, including `the builtin range()` and *`abs()`* functions. We would have to wrap those functions with `timeit` calls as well in order to get information about how long those individual functions took to run.

Therefore, what we really need is a tool that will not only time a function or method, but also all of the functions that are called from the parent as well as any other executing code.

Python offers a tool called `cProfile`. This tool, when run, generates output about each function and sub-function in the program

Let's look at a very simple example

```
#!/usr/bin/python2.7

import time
import cProfile

def calcsum():
    calcsum=0
    for i in range(100000):
        calcsum += i

def main() :
    calcsum()

if __name__ == "__main__":
    cProfile.run('main()')
```

Note here that we import the cProfile and call cProfile.run with the main function as the argument. cProfile will generate a table of data that looks like this:

```
5 function calls in 0.006 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
        1    0.000    0.000    0.006    0.006
<string>:1(<module>)
        1    0.000    0.000    0.006    0.006
profile4.py:12(main)
        1    0.004    0.004    0.006    0.006
profile4.py:6(calcsum)
        1    0.000    0.000    0.000    0.000 {method
'disable' of '_lsprof.Profiler' objects}
        1    0.002    0.002    0.002    0.002 {range}
```

The output gives us the following:

`<x>` function calls in `<y>` seconds. A rather self-explanatory line saying that the profiler ran, in this case, five different functions in .006 seconds.

`Ordered by` means that the output was sorted by the name of the function.

Output Descriptions

Column Name	Description
Ncalls	How many times that particular function was called
Tottime	How long it took to execute each function *not including* sub functions
PerCall	The total time divided by the number of calls
CumTime	How long it took to execute each function *including* sub functions
PerCall	The cumulative time divided by the number of calls
Function name	The name of the function being profiled

The columns are described above.

The `run` method of the `cprofiler` also allows for a second argument which is a file name that the statistical information can be stored to. This is quite useful when manipulating the output with a module called `pstats()`.

Additionally, saving the profiler output to a file will also allow us to visualize the data with tools such as *snakeviz* and *runsnakerun*.

Another way to run `cProfile` is from the command line. Instead of calling it programmatically, we can do something like this:

```
python -m cProfile -o <profile output file> -s <sort order of
output> <script_name>
```

Note that the `-s` flag only works if you do not specify an output file to which to send the data.

Running the `pstats` module on the output allows us to sort the output in various ways as well as controlling the output of the statistical information. The Python documentation has more information about the `pstats` module.