Profiling & Intro To Design Patterns

COMP3522 OBJECT ORIENTED PROGRAMMING 2

WEEK 8

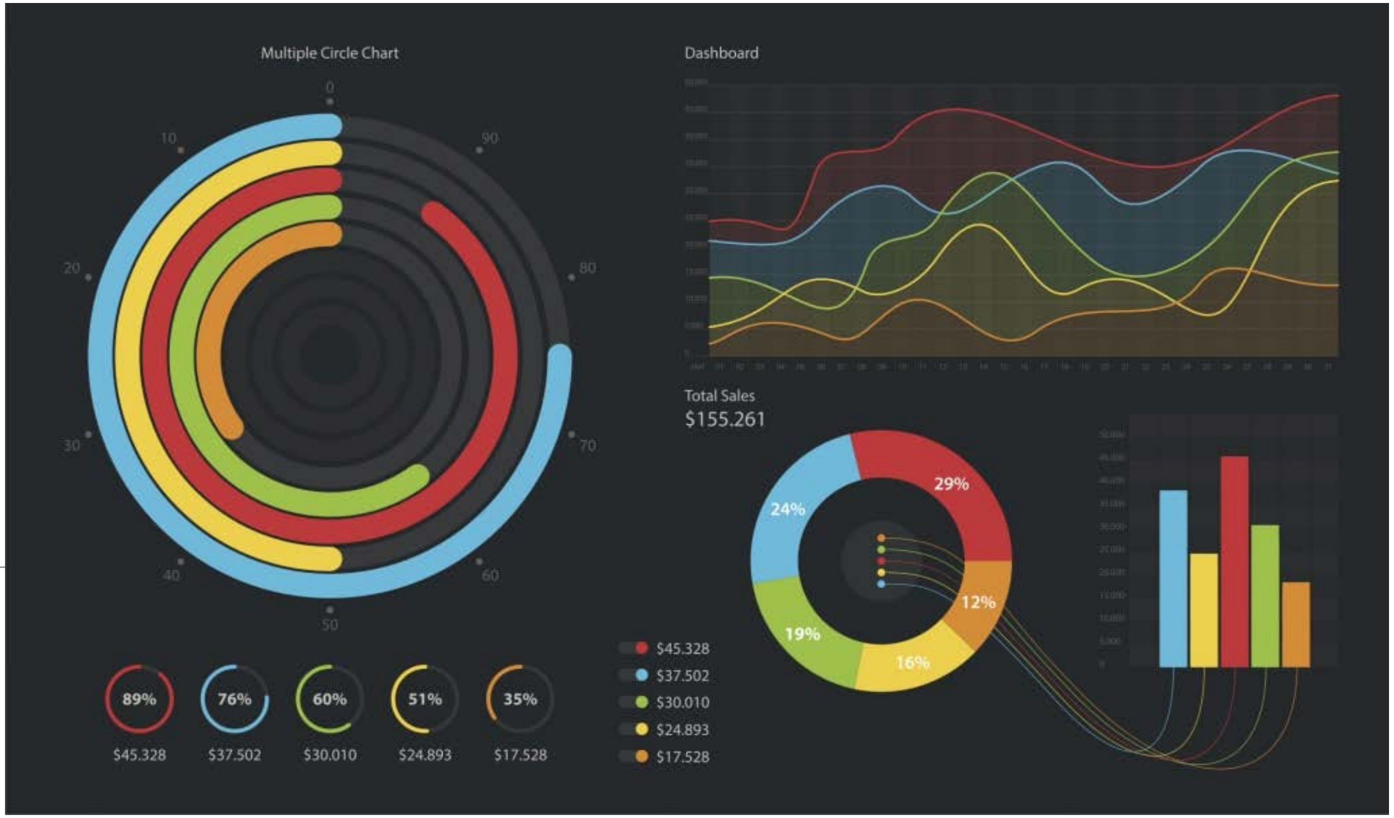
Recap – Last time

- Generators
- Lambda expressions

COMP 3522

Profiling

I PUT THIS IMAGE THERE CAUSE IT LOOKS COOL AND STATISTICAL. I DON'T EVEN KNOW IF ITS PROFILING.



What is profiling?

Profiling is what we call analyzing our program's runtime performance:

- Space complexity
- Time complexity
- Usage of particular instructions
- Frequency and duration of function calls

Profiling is a form of dynamic program analysis

We build a profile with a **profiler**

We use the profile to aid in **program optimization**

Our goal is **correct, dependable, efficient (fast)** software.

Instrumentation

Profiling works by *instrumenting* the program source code or a binary executable

Instrumentation measures the level of a product’s performance

Instrumentation incorporates (you don’t need to memorize these fancy words):

- Code tracing
- Debugging
- Performance counters
- Logging, etc.

Profiler

We profile our code using a code profiler

Profilers use a variety of methods in addition to instrumentation:

- Events
- Statistics
- Simulations
- Hardware interrupts
- OS hooks
- Performance counters…

Profilers can produce profiles

A profile is a statistical summary of the events observed

The statistics are often displayed as annotations beside the source code, like this:

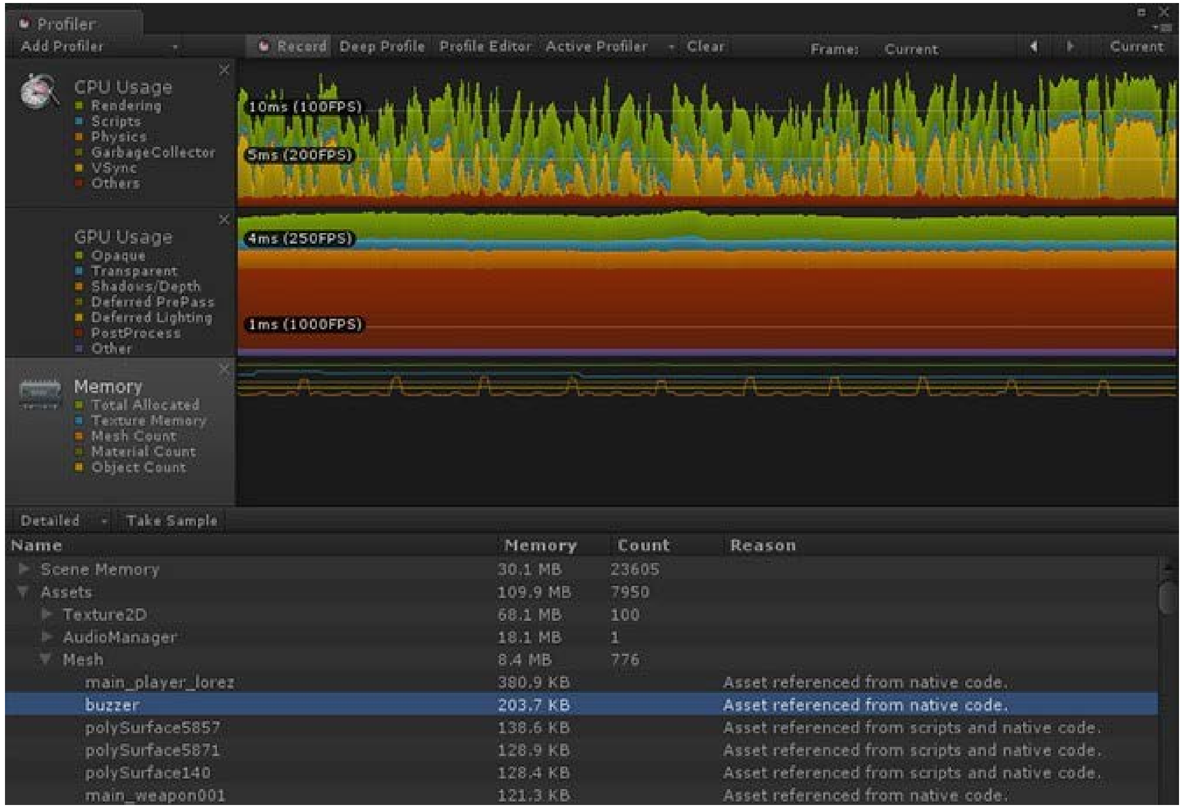
```
/* ------------ source------------------------- count */
0001             IF X = "A"                     0055
0002                THEN DO
0003                  ADD 1 to XCOUNT           0032
0004                ELSE
0005             IF X = "B"                     0055
```

Profilers can produce traces

A trace is a stream of recorded events

We often use traces for parallel programs in order to understand how things are happening

A summary profile is usually sufficient for sequential programs



Unity Game Engine - Profiler

Building the profile

Profilers run during a program's execution (**dynamic analysis**)

The program execution must be interrupted by the profiler so that it can collect information

This can sometimes have a non-trivial effect on time measurements

We say that the **<u>resolution</u>** is "limited"

It's still helpful to us, though!

Profilers – 2 Types

Statistical Profiling

Deterministic Profiling

Statistical profiling

The Python profiler does ***not*** use statistical profiling

Statistical profiling randomly samples the instruction pointer and deduces where time is being spent

This involves less overhead because it doesn't require instrumentation

But it provides only *relative indications* of where the time was spent

Deterministic profiling

The Python profilers we can use are from the **cprofile** and **profile** modules

(Because there's a module for everything in Python, of course!)

Python's profilers use ***deterministic profiling***

This means that all function call, function return, and exception events are monitored

Precise times are recorded for the intervals between these events

This requires overhead that can skew results. (Usually not a problem).

How does Profiling help?

Call count statistics can be used to identify:

- Bugs (surprisingly high calls)
- Inline-expansion points

  places where we can reduce the overhead of calling functions and adding to the stack trace by replacing a function call with the body of the function call.

Internal time statistics can be used to identify “hot loops” that need to be optimized.
(This is a technical way of saying identify functions that take too long and optimize them.)

Cumulative time statistics can be used to identify high-level errors in algorithms.
(This is a technical way of saying find inefficient code logic spanning multiple functions and files and re-designing it.)

It all boils down to speed*. How fast can we make our program while ensuring it still works correctly?

* It’s never, ever fast enough.

cProfile.

Let’s profile something.

We can do it in 2 ways:

- Command Line (super useful)
- PyCharm (Professional Edition, available via education accounts)

cProfile – profiling a statement.

```
>>> import cProfile
>>> import re
>>> help(cProfile)
>>> help(re)
>>> help(cProfile.run)
```

cProfile – profiling a statement

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

test2.py

cProfile results

Q: What’s all this?

A: Stats from executing your program

Q: How are we supposed to read this?

A: I’ll show you

Q: Can’t we use PyCharm for this?

A: Yes can be done in command line or PyCharm

```
>>> import cProfile
>>> import re
>>> cProfile.run('re.compile("foo|bar")')
         214 function calls (207 primitive calls) in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        2    0.000    0.000    0.000    0.000 enum.py:284(__call__)
        2    0.000    0.000    0.000    0.000 enum.py:526(__new__)
        1    0.000    0.000    0.000    0.000 enum.py:836(__and__)
        1    0.000    0.000    0.000    0.000 re.py:232(compile)
        1    0.000    0.000    0.000    0.000 re.py:271(_compile)
        1    0.000    0.000    0.000    0.000 sre_compile.py:249(_compile_charset)
        1    0.000    0.000    0.000    0.000 sre_compile.py:276(_optimize_charset)
        2    0.000    0.000    0.000    0.000 sre_compile.py:453(_get_iscased)
        1    0.000    0.000    0.000    0.000 sre_compile.py:461(_get_literal_prefix)
        1    0.000    0.000    0.000    0.000 sre_compile.py:492(_get_charset_prefix)
        1    0.000    0.000    0.000    0.000 sre_compile.py:536(_compile_info)
        2    0.000    0.000    0.000    0.000 sre_compile.py:595(isstring)
        1    0.000    0.000    0.000    0.000 sre_compile.py:598(_code)
      3/1    0.000    0.000    0.000    0.000 sre_compile.py:71(_compile)
        1    0.000    0.000    0.000    0.000 sre_compile.py:759(compile)
        3    0.000    0.000    0.000    0.000 sre_parse.py:111(__init__)
        7    0.000    0.000    0.000    0.000 sre_parse.py:160(__len__)
       18    0.000    0.000    0.000    0.000 sre_parse.py:164(__getitem__)
        7    0.000    0.000    0.000    0.000 sre_parse.py:172(append)
      3/1    0.000    0.000    0.000    0.000 sre_parse.py:174(getwidth)
        1    0.000    0.000    0.000    0.000 sre_parse.py:224(__init__)
        8    0.000    0.000    0.000    0.000 sre_parse.py:233(__next)
        2    0.000    0.000    0.000    0.000 sre_parse.py:249(match)
        6    0.000    0.000    0.000    0.000 sre_parse.py:254(get)
        1    0.000    0.000    0.000    0.000 sre_parse.py:286(tell)
        1    0.000    0.000    0.000    0.000 sre_parse.py:417(_parse_sub)
        2    0.000    0.000    0.000    0.000 sre_parse.py:475(_parse)
        1    0.000    0.000    0.000    0.000 sre_parse.py:76(__init__)
        2    0.000    0.000    0.000    0.000 sre_parse.py:81(groups)
        1    0.000    0.000    0.000    0.000 sre_parse.py:903(fix_flags)
        1    0.000    0.000    0.000    0.000 sre_parse.py:919(parse)
        1    0.000    0.000    0.000    0.000 {built-in method _sre.compile}
        1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
       25    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
    29/26    0.000    0.000    0.000    0.000 {built-in method builtins.len}
        2    0.000    0.000    0.000    0.000 {built-in method builtins.max}
        9    0.000    0.000    0.000    0.000 {built-in method builtins.min}
        6    0.000    0.000    0.000    0.000 {built-in method builtins.ord}
       48    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        5    0.000    0.000    0.000    0.000 {method 'find' of 'bytearray' objects}
        1    0.000    0.000    0.000    0.000 {method 'items' of 'dict' objects}


>>>
```

cProfile results

The first line indicates that 214 calls were monitored

Of those calls, 207 were *primitive*

Primitive calls are not recursive calls

```
>>> import cProfile
>>> import re
>>> cProfile.run('re.compile("foo|bar")')
         214 function calls (207 primitive calls) in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        2    0.000    0.000    0.000    0.000 enum.py:284(__call__)
        2    0.000    0.000    0.000    0.000 enum.py:526(__new__)
        1    0.000    0.000    0.000    0.000 enum.py:836(__and__)
        1    0.000    0.000    0.000    0.000 re.py:232(compile)
        1    0.000    0.000    0.000    0.000 re.py:271(_compile)
        1    0.000    0.000    0.000    0.000 sre_compile.py:249(_compile_charset)
        1    0.000    0.000    0.000    0.000 sre_compile.py:276(_optimize_charset)
        2    0.000    0.000    0.000    0.000 sre_compile.py:453(_get_iscased)
        1    0.000    0.000    0.000    0.000 sre_compile.py:461(_get_literal_prefix)
        1    0.000    0.000    0.000    0.000 sre_compile.py:492(_get_charset_prefix)
        1    0.000    0.000    0.000    0.000 sre_compile.py:536(_compile_info)
        2    0.000    0.000    0.000    0.000 sre_compile.py:595(isstring)
        1    0.000    0.000    0.000    0.000 sre_compile.py:598(_code)
      3/1    0.000    0.000    0.000    0.000 sre_compile.py:71(_compile)
        1    0.000    0.000    0.000    0.000 sre_compile.py:759(compile)
        3    0.000    0.000    0.000    0.000 sre_parse.py:111(__init__)
        7    0.000    0.000    0.000    0.000 sre_parse.py:160(__len__)
       18    0.000    0.000    0.000    0.000 sre_parse.py:164(__getitem__)
        7    0.000    0.000    0.000    0.000 sre_parse.py:172(append)
      3/1    0.000    0.000    0.000    0.000 sre_parse.py:174(getwidth)
        1    0.000    0.000    0.000    0.000 sre_parse.py:224(__init__)
        8    0.000    0.000    0.000    0.000 sre_parse.py:233(__next)
        2    0.000    0.000    0.000    0.000 sre_parse.py:249(match)
        6    0.000    0.000    0.000    0.000 sre_parse.py:254(get)
        1    0.000    0.000    0.000    0.000 sre_parse.py:286(tell)
        1    0.000    0.000    0.000    0.000 sre_parse.py:417(_parse_sub)
        2    0.000    0.000    0.000    0.000 sre_parse.py:475(_parse)
        1    0.000    0.000    0.000    0.000 sre_parse.py:76(__init__)
        2    0.000    0.000    0.000    0.000 sre_parse.py:81(groups)
        1    0.000    0.000    0.000    0.000 sre_parse.py:903(fix_flags)
        1    0.000    0.000    0.000    0.000 sre_parse.py:919(parse)
ance}
ts}
ofiler' objects}
jects}
s}

>>>
```

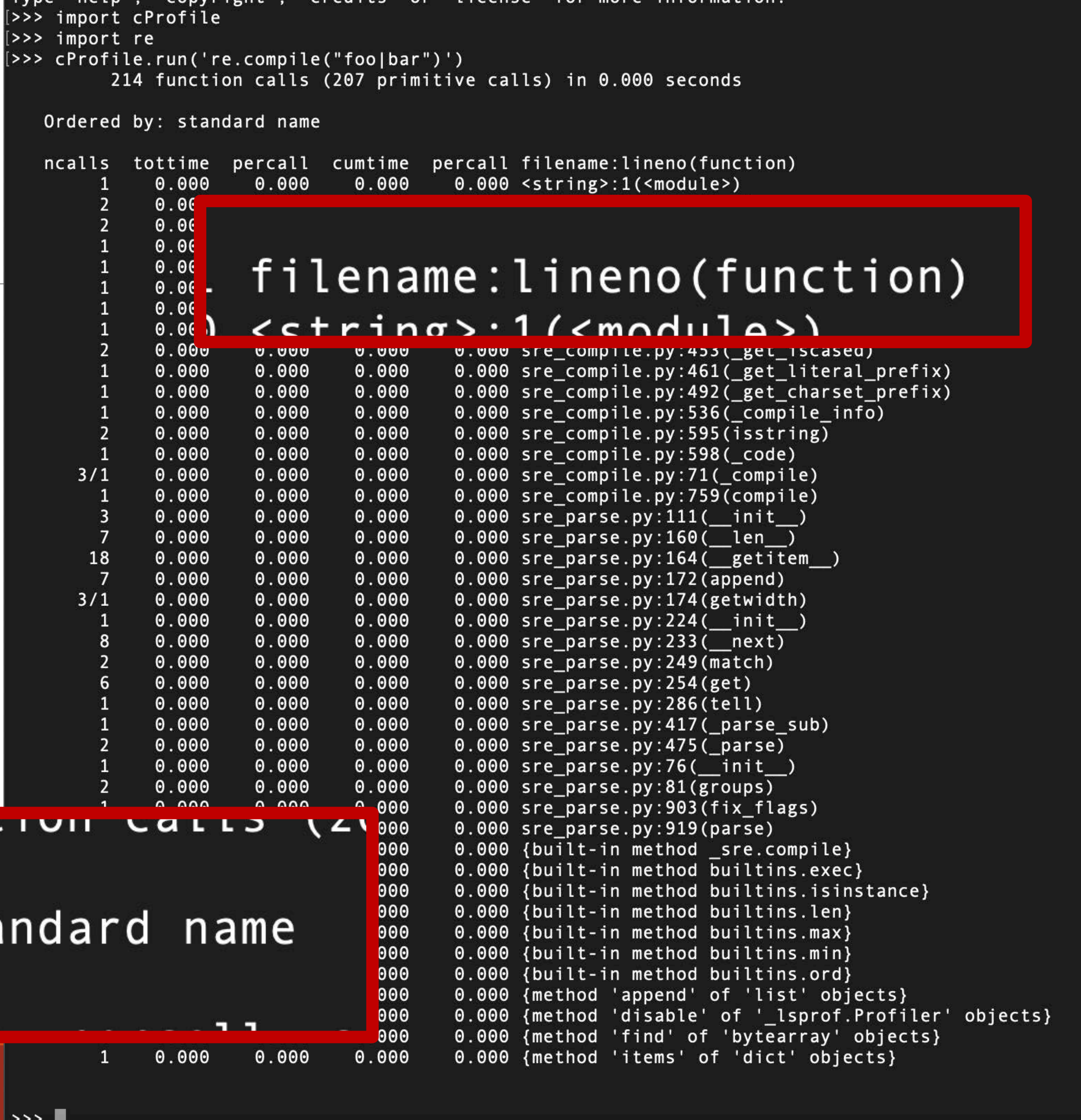


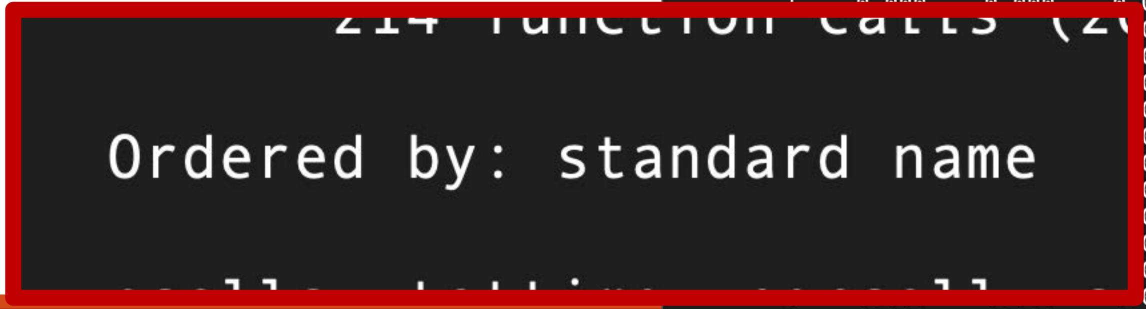
```
>> import re
>> cProfile.run('re.compile("foo|bar")')
         214 function calls (207 primitive calls) in 0.000 seconds

  Ordered by: standard name
```

cProfile results

The next line tells us that the text string in the far right column was used to sort the output

In this case the string is filename:lineno(function)





cProfile results

The first column heading is ncalls

This is (wait for it…) the number of times the function was called

```
>>> import cProfile
>>> import re
>>> cProfile.run('re.compile("foo|bar")')
         214 function calls (207 primitive calls) in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        2    0.000    0.000    0.000    0.000 enum.py:284(__call__)
        2    0.000    0.000    0.000    0.000 enum.py:526(__new__)
        1    0.000    0.000    0.000    0.000 enum.py:836(__and__)
        1    0.000    0.000    0.000    0.000 re.py:232(compile)
        1    0.000    0.000    0.000    0.000 re.py:271(_compile)
        1    0.000    0.000    0.000    0.000 sre_compile.py:249(_compile_charset)
        1    0.000    0.000    0.000    0.000 sre_compile.py:276(_optimize_charset)
        2    0.000    0.000    0.000    0.000 sre_compile.py:453(_get_iscased)
        1    0.000    0.000    0.000    0.000 sre_compile.py:461(_get_literal_prefix)
        1    0.000    0.000    0.000    0.000 sre_compile.py:492(_get_charset_prefix)
        1    0.000    0.000    0.000    0.000 sre_compile.py:536(_compile_info)
        2    0.000    0.000    0.000    0.000 sre_compile.py:595(isstring)
        1    0.000    0.000    0.000    0.000 sre_compile.py:598(_code)
      3/1    0.000    0.000    0.000    0.000 sre_compile.py:71(_compile)
        1    0.000    0.000    0.000    0.000 sre_compile.py:759(compile)
        3    0.000    0.000    0.000    0.000 sre_parse.py:111(__init__)
        7    0.000    0.000    0.000    0.000 sre_parse.py:160(__len__)
       18    0.000    0.000    0.000    0.000 sre_parse.py:164(__getitem__)
        7    0.000    0.000    0.000    0.000 sre_parse.py:172(append)
      3/1    0.000    0.000    0.000    0.000 sre_parse.py:174(getwidth)
        1    0.000    0.000    0.000    0.000 sre_parse.py:224(__init__)
        8    0.000    0.000    0.000    0.000 sre_parse.py:233(__next)
        2    0.000    0.000    0.000    0.000 sre_parse.py:249(match)
        6    0.000    0.000    0.000    0.000 sre_parse.py:254(get)
        1    0.000    0.000    0.000    0.000 sre_parse.py:286(tell)
        1    0.000    0.000    0.000    0.000 sre_parse.py:417(_parse_sub)
        2    0.000    0.000    0.000    0.000 sre_parse.py:475(_parse)
        1    0.000    0.000    0.000    0.000 sre_parse.py:76(__init__)
        2    0.000    0.000    0.000    0.000 sre_parse.py:81(groups)
        1    0.000    0.000    0.000    0.000 sre_parse.py:903(fix_flags)
        1    0.000    0.000    0.000    0.000 sre_parse.py:919(parse)
        1    0.000    0.000    0.000    0.000 {built-in method _sre.compile}
        1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
       25    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
    29/26    0.000    0.000    0.000    0.000 {built-in method builtins.len}
        2    0.000    0.000    0.000    0.000 {built-in method builtins.max}
        9    0.000    0.000    0.000    0.000 {built-in method builtins.min}
        6    0.000    0.000    0.000    0.000 {built-in method builtins.ord}
       48    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        5    0.000    0.000    0.000    0.000 {method 'find' of 'bytearray' objects}
        1    0.000    0.000    0.000    0.000 {method 'items' of 'dict' objects}


>>>
```

cProfile results

The second column heading is tottime

This represents the total time spent in the function

It does not include time spent in calls to sub-functions


```
>>> import cProfile
>>> import re
>>> cProfile.run('re.compile("foo|bar")')
         214 function calls (207 primitive calls) in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        2    0.000    0.000    0.000    0.000 enum.py:284(__call__)
        2    0.000    0.000    0.000    0.000 enum.py:526(__new__)
        1    0.000    0.000    0.000    0.000 enum.py:836(__and__)
        1    0.000    0.000    0.000    0.000 re.py:232(compile)
        1    0.000    0.000    0.000    0.000 re.py:271(_compile)
        1    0.000    0.000    0.000    0.000 sre_compile.py:249(_compile_charset)
        1    0.000    0.000    0.000    0.000 sre_compile.py:276(_optimize_charset)
        2    0.000    0.000    0.000    0.000 sre_compile.py:453(_get_iscased)
        1    0.000    0.000    0.000    0.000 sre_compile.py:461(_get_literal_prefix)
        1    0.000    0.000    0.000    0.000 sre_compile.py:492(_get_charset_prefix)
        1    0.000    0.000    0.000    0.000 sre_compile.py:536(_compile_info)
        2    0.000    0.000    0.000    0.000 sre_compile.py:595(isstring)
        1    0.000    0.000    0.000    0.000 sre_compile.py:598(_code)
      3/1    0.000    0.000    0.000    0.000 sre_compile.py:71(_compile)
        1    0.000    0.000    0.000    0.000 sre_compile.py:759(compile)
        3    0.000    0.000    0.000    0.000 sre_parse.py:111(__init__)
        7    0.000    0.000    0.000    0.000 sre_parse.py:160(__len__)
       18    0.000    0.000    0.000    0.000 sre_parse.py:164(__getitem__)
        7    0.000    0.000    0.000    0.000 sre_parse.py:172(append)
      3/1    0.000    0.000    0.000    0.000 sre_parse.py:174(getwidth)
        1    0.000    0.000    0.000    0.000 sre_parse.py:224(__init__)
        8    0.000    0.000    0.000    0.000 sre_parse.py:233(__next)
        2    0.000    0.000    0.000    0.000 sre_parse.py:249(match)
        6    0.000    0.000    0.000    0.000 sre_parse.py:254(get)
        1    0.000    0.000    0.000    0.000 sre_parse.py:286(tell)
        1    0.000    0.000    0.000    0.000 sre_parse.py:417(_parse_sub)
        2    0.000    0.000    0.000    0.000 sre_parse.py:475(_parse)
        1    0.000    0.000    0.000    0.000 sre_parse.py:76(__init__)
        2    0.000    0.000    0.000    0.000 sre_parse.py:81(groups)
        1    0.000    0.000    0.000    0.000 sre_parse.py:903(fix_flags)
        1    0.000    0.000    0.000    0.000 sre_parse.py:919(parse)
        1    0.000    0.000    0.000    0.000 {built-in method _sre.compile}
        1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
       25    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
    29/26    0.000    0.000    0.000    0.000 {built-in method builtins.len}
        2    0.000    0.000    0.000    0.000 {built-in method builtins.max}
        9    0.000    0.000    0.000    0.000 {built-in method builtins.min}
        6    0.000    0.000    0.000    0.000 {built-in method builtins.ord}
       48    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        5    0.000    0.000    0.000    0.000 {method 'find' of 'bytearray' objects}
        1    0.000    0.000    0.000    0.000 {method 'items' of 'dict' objects}


>>>
```

cProfile results

The third column heading is percall

percall =

tottime / numcalls

We see that the time per call is negligible

(I expect library code to be highly optimized!)


```
>>> import cProfile
>>> import re
>>> cProfile.run('re.compile("foo|bar")')
         214 function calls (207 primitive calls) in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        2    0.000    0.000    0.000    0.000 enum.py:284(__call__)
        2    0.000    0.000    0.000    0.000 enum.py:526(__new__)
        1    0.000    0.000    0.000    0.000 enum.py:836(__and__)
        1    0.000    0.000    0.000    0.000 re.py:232(compile)
        1    0.000    0.000    0.000    0.000 re.py:271(_compile)
        1    0.000    0.000    0.000    0.000 sre_compile.py:249(_compile_charset)
        1    0.000    0.000    0.000    0.000 sre_compile.py:276(_optimize_charset)
        2    0.000    0.000    0.000    0.000 sre_compile.py:453(_get_iscased)
        1    0.000    0.000    0.000    0.000 sre_compile.py:461(_get_literal_prefix)
        1    0.000    0.000    0.000    0.000 sre_compile.py:492(_get_charset_prefix)
        1    0.000    0.000    0.000    0.000 sre_compile.py:536(_compile_info)
        2    0.000    0.000    0.000    0.000 sre_compile.py:595(isstring)
        1    0.000    0.000    0.000    0.000 sre_compile.py:598(_code)
      3/1    0.000    0.000    0.000    0.000 sre_compile.py:71(_compile)
        1    0.000    0.000    0.000    0.000 sre_compile.py:759(compile)
        3    0.000    0.000    0.000    0.000 sre_parse.py:111(__init__)
        7    0.000    0.000    0.000    0.000 sre_parse.py:160(__len__)
       18    0.000    0.000    0.000    0.000 sre_parse.py:164(__getitem__)
        7    0.000    0.000    0.000    0.000 sre_parse.py:172(append)
      3/1    0.000    0.000    0.000    0.000 sre_parse.py:174(getwidth)
        1    0.000    0.000    0.000    0.000 sre_parse.py:224(__init__)
        8    0.000    0.000    0.000    0.000 sre_parse.py:233(__next)
        2    0.000    0.000    0.000    0.000 sre_parse.py:249(match)
        6    0.000    0.000    0.000    0.000 sre_parse.py:254(get)
        1    0.000    0.000    0.000    0.000 sre_parse.py:286(tell)
        1    0.000    0.000    0.000    0.000 sre_parse.py:417(_parse_sub)
        2    0.000    0.000    0.000    0.000 sre_parse.py:475(_parse)
        1    0.000    0.000    0.000    0.000 sre_parse.py:76(__init__)
        2    0.000    0.000    0.000    0.000 sre_parse.py:81(groups)
        1    0.000    0.000    0.000    0.000 sre_parse.py:903(fix_flags)
        1    0.000    0.000    0.000    0.000 sre_parse.py:919(parse)
        1    0.000    0.000    0.000    0.000 {built-in method _sre.compile}
        1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
       25    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
    29/26    0.000    0.000    0.000    0.000 {built-in method builtins.len}
        2    0.000    0.000    0.000    0.000 {built-in method builtins.max}
        9    0.000    0.000    0.000    0.000 {built-in method builtins.min}
        6    0.000    0.000    0.000    0.000 {built-in method builtins.ord}
       48    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        5    0.000    0.000    0.000    0.000 {method 'find' of 'bytearray' objects}
        1    0.000    0.000    0.000    0.000 {method 'items' of 'dict' objects}

>>>
```

cProfile results

The fourth column heading is cumtime (cumulative time).

This is the cumulative time spent in this and all subfunctions

Counts from invocation till exit

This figure is accurate even for recursive functions.


```
>>> import cProfile
>>> import re
>>> cProfile.run('re.compile("foo|bar")')
         214 function calls (207 primitive calls) in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        2    0.000    0.000    0.000    0.000 enum.py:284(__call__)
        2    0.000    0.000    0.000    0.000 enum.py:526(__new__)
        1    0.000    0.000    0.000    0.000 enum.py:836(__and__)
        1    0.000    0.000    0.000    0.000 re.py:232(compile)
        1    0.000    0.000    0.000    0.000 re.py:271(_compile)
        1    0.000    0.000    0.000    0.000 sre_compile.py:249(_compile_charset)
        1    0.000    0.000    0.000    0.000 sre_compile.py:276(_optimize_charset)
        2    0.000    0.000    0.000    0.000 sre_compile.py:453(_get_iscased)
        1    0.000    0.000    0.000    0.000 sre_compile.py:461(_get_literal_prefix)
        1    0.000    0.000    0.000    0.000 sre_compile.py:492(_get_charset_prefix)
        1    0.000    0.000    0.000    0.000 sre_compile.py:536(_compile_info)
        2    0.000    0.000    0.000    0.000 sre_compile.py:595(isstring)
        1    0.000    0.000    0.000    0.000 sre_compile.py:598(_code)
      3/1    0.000    0.000    0.000    0.000 sre_compile.py:71(_compile)
        1    0.000    0.000    0.000    0.000 sre_compile.py:759(compile)
        3    0.000    0.000    0.000    0.000 sre_parse.py:111(__init__)
        7    0.000    0.000    0.000    0.000 sre_parse.py:160(__len__)
       18    0.000    0.000    0.000    0.000 sre_parse.py:164(__getitem__)
        7    0.000    0.000    0.000    0.000 sre_parse.py:172(append)
      3/1    0.000    0.000    0.000    0.000 sre_parse.py:174(getwidth)
        1    0.000    0.000    0.000    0.000 sre_parse.py:224(__init__)
        8    0.000    0.000    0.000    0.000 sre_parse.py:233(__next)
        2    0.000    0.000    0.000    0.000 sre_parse.py:249(match)
        6    0.000    0.000    0.000    0.000 sre_parse.py:254(get)
        1    0.000    0.000    0.000    0.000 sre_parse.py:286(tell)
        1    0.000    0.000    0.000    0.000 sre_parse.py:417(_parse_sub)
        2    0.000    0.000    0.000    0.000 sre_parse.py:475(_parse)
        1    0.000    0.000    0.000    0.000 sre_parse.py:76(__init__)
        2    0.000    0.000    0.000    0.000 sre_parse.py:81(groups)
        1    0.000    0.000    0.000    0.000 sre_parse.py:903(fix_flags)
        1    0.000    0.000    0.000    0.000 sre_parse.py:919(parse)
        1    0.000    0.000    0.000    0.000 {built-in method _sre.compile}
        1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
       25    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
    29/26    0.000    0.000    0.000    0.000 {built-in method builtins.len}
        2    0.000    0.000    0.000    0.000 {built-in method builtins.max}
        9    0.000    0.000    0.000    0.000 {built-in method builtins.min}
        6    0.000    0.000    0.000    0.000 {built-in method builtins.ord}
       48    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        5    0.000    0.000    0.000    0.000 {method 'find' of 'bytearray' objects}
        1    0.000    0.000    0.000    0.000 {method 'items' of 'dict' objects}

>>>
```

cProfile results

The fifth column heading is percall again. Wait. What?

This is another call measure

This one divides cumtime by primitive calls


```
Type "help", "copyright", "credits" or "license" for more information.
>>> import cProfile
>>> import re
>>> cProfile.run('re.compile("foo|bar")')
         214 function calls (207 primitive calls) in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        2    0.000    0.000    0.000    0.000 enum.py:284(__call__)
        2    0.000    0.000    0.000    0.000 enum.py:526(__new__)
        1    0.000    0.000    0.000    0.000 enum.py:836(__and__)
        1    0.000    0.000    0.000    0.000 re.py:232(compile)
        1    0.000    0.000    0.000    0.000 re.py:271(_compile)
        1    0.000    0.000    0.000    0.000 sre_compile.py:249(_compile_charset)
        1    0.000    0.000    0.000    0.000 sre_compile.py:276(_optimize_charset)
        2    0.000    0.000    0.000    0.000 sre_compile.py:453(_get_iscased)
        1    0.000    0.000    0.000    0.000 sre_compile.py:461(_get_literal_prefix)
        1    0.000    0.000    0.000    0.000 sre_compile.py:492(_get_charset_prefix)
        1    0.000    0.000    0.000    0.000 sre_compile.py:536(_compile_info)
        2    0.000    0.000    0.000    0.000 sre_compile.py:595(isstring)
        1    0.000    0.000    0.000    0.000 sre_compile.py:598(_code)
      3/1    0.000    0.000    0.000    0.000 sre_compile.py:71(_compile)
        1    0.000    0.000    0.000    0.000 sre_compile.py:759(compile)
        3    0.000    0.000    0.000    0.000 sre_parse.py:111(__init__)
        7    0.000    0.000    0.000    0.000 sre_parse.py:160(__len__)
       18    0.000    0.000    0.000    0.000 sre_parse.py:164(__getitem__)
        7    0.000    0.000    0.000    0.000 sre_parse.py:172(append)
      3/1    0.000    0.000    0.000    0.000 sre_parse.py:174(getwidth)
        1    0.000    0.000    0.000    0.000 sre_parse.py:224(__init__)
        8    0.000    0.000    0.000    0.000 sre_parse.py:233(__next)
        2    0.000    0.000    0.000    0.000 sre_parse.py:249(match)
        6    0.000    0.000    0.000    0.000 sre_parse.py:254(get)
        1    0.000    0.000    0.000    0.000 sre_parse.py:286(tell)
        1    0.000    0.000    0.000    0.000 sre_parse.py:417(_parse_sub)
        2    0.000    0.000    0.000    0.000 sre_parse.py:475(_parse)
        1    0.000    0.000    0.000    0.000 sre_parse.py:76(__init__)
        2    0.000    0.000    0.000    0.000 sre_parse.py:81(groups)
        1    0.000    0.000    0.000    0.000 sre_parse.py:903(fix_flags)
        1    0.000    0.000    0.000    0.000 sre_parse.py:919(parse)
        1    0.000    0.000    0.000    0.000 {built-in method _sre.compile}
        1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
       25    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
    29/26    0.000    0.000    0.000    0.000 {built-in method builtins.len}
        2    0.000    0.000    0.000    0.000 {built-in method builtins.max}
        9    0.000    0.000    0.000    0.000 {built-in method builtins.min}
        6    0.000    0.000    0.000    0.000 {built-in method builtins.ord}
       48    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        5    0.000    0.000    0.000    0.000 {method 'find' of 'bytearray' objects}
        1    0.000    0.000    0.000    0.000 {method 'items' of 'dict' objects}


>>>
```

cProfile results

When there are two values in the ncalls column, it means the function recursed:

The first number is the total number of calls

The second value is the number of primitive calls

- Primitive calls do NOT include recursive calls

3/1

- 3 total calls
- 1 primitive call
- (3-1) 2 recursive calls

```
>>> import cProfile
>>> import re
>>> cProfile.run('re.compile("foo|bar")')
         214 function calls (207 primitive calls) in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        2    0.000    0.000    0.000    0.000 enum.py:284(__call__)
        2    0.000    0.000    0.000    0.000 enum.py:526(__new__)
        1    0.000    0.000    0.000    0.000 enum.py:836(__and__)
        1    0.000    0.000    0.000    0.000 re.py:232(compile)
        1    0.000    0.000    0.000    0.000 re.py:271(_compile)
        1    0.000    0.000    0.000    0.000 sre_compile.py:249(_compile_charset)
        1    0.000    0.000    0.000    0.000 sre_compile.py:276(_optimize_charset)
        2    0.000    0.000    0.000    0.000 sre_compile.py:453(_get_iscased)
               00    0.000    0.000    0.000 sre_compile.py:461(_get_literal_prefix)
               00    0.000    0.000    0.000 sre_compile.py:492(_get_charset_prefix)
               00    0.000    0.000    0.000 sre_compile.py:536(_compile_info)
               00    0.000    0.000    0.000 sre_compile.py:595(isstring)
        1    0.000    0.000    0.000    0.000 sre_compile.py:598(_code)
      3/1    0.000    0.000    0.000    0.000 sre_compile.py:71(_compile)
        1    0.000    0.000    0.000    0.000 sre_compile.py:759(compile)
        3    0.000    0.000    0.000    0.000 sre_parse.py:111(__init__)
        7    0.000    0.000    0.000    0.000 sre_parse.py:160(__len__)
       18    0.000    0.000    0.000    0.000 sre_parse.py:164(__getitem__)
        7    0.000    0.000    0.000    0.000 sre_parse.py:172(append)
      3/1    0.000    0.000    0.000    0.000 sre_parse.py:174(getwidth)
        1    0.000    0.000    0.000    0.000 sre_parse.py:224(__init__)
        8    0.000    0.000    0.000    0.000 sre_parse.py:233(__next)
        2    0.000    0.000    0.000    0.000 sre_parse.py:249(match)
        6    0.000    0.000    0.000    0.000 sre_parse.py:254(get)
        1    0.000    0.000    0.000    0.000 sre_parse.py:286(tell)
        1    0.000    0.000    0.000    0.000 sre_parse.py:417(_parse_sub)
        2    0.000    0.000    0.000    0.000 sre_parse.py:475(_parse)
        1    0.000    0.000    0.000    0.000 sre_parse.py:76(__init__)
                          0.000    0.000    0.000 sre_parse.py:81(groups)
                          0.000    0.000    0.000 sre_parse.py:903(fix_flags)
                          0.000    0.000    0.000 sre_parse.py:919(parse)
                          0.000    0.000    0.000 {built-in method _sre.compile}
                          0.000    0.000    0.000 {built-in method builtins.exec}
       25    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
    29/26    0.000    0.000    0.000    0.000 {built-in method builtins.len}
        2    0.000    0.000    0.000    0.000 {built-in method builtins.max}
        9    0.000    0.000    0.000    0.000 {built-in method builtins.min}
        6    0.000    0.000    0.000    0.000 {built-in method builtins.ord}
       48    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        5    0.000    0.000    0.000    0.000 {method 'find' of 'bytearray' objects}
        1    0.000    0.000    0.000    0.000 {method 'items' of 'dict' objects}

>>>
```

3/1

29/26

We can save the results to a file

Instead of printing the output, we can save it to the file

cProfile.run accepts an optional second parameter, a filename:

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

Open the file. What do you see?

This isn’t helpful, Jeff

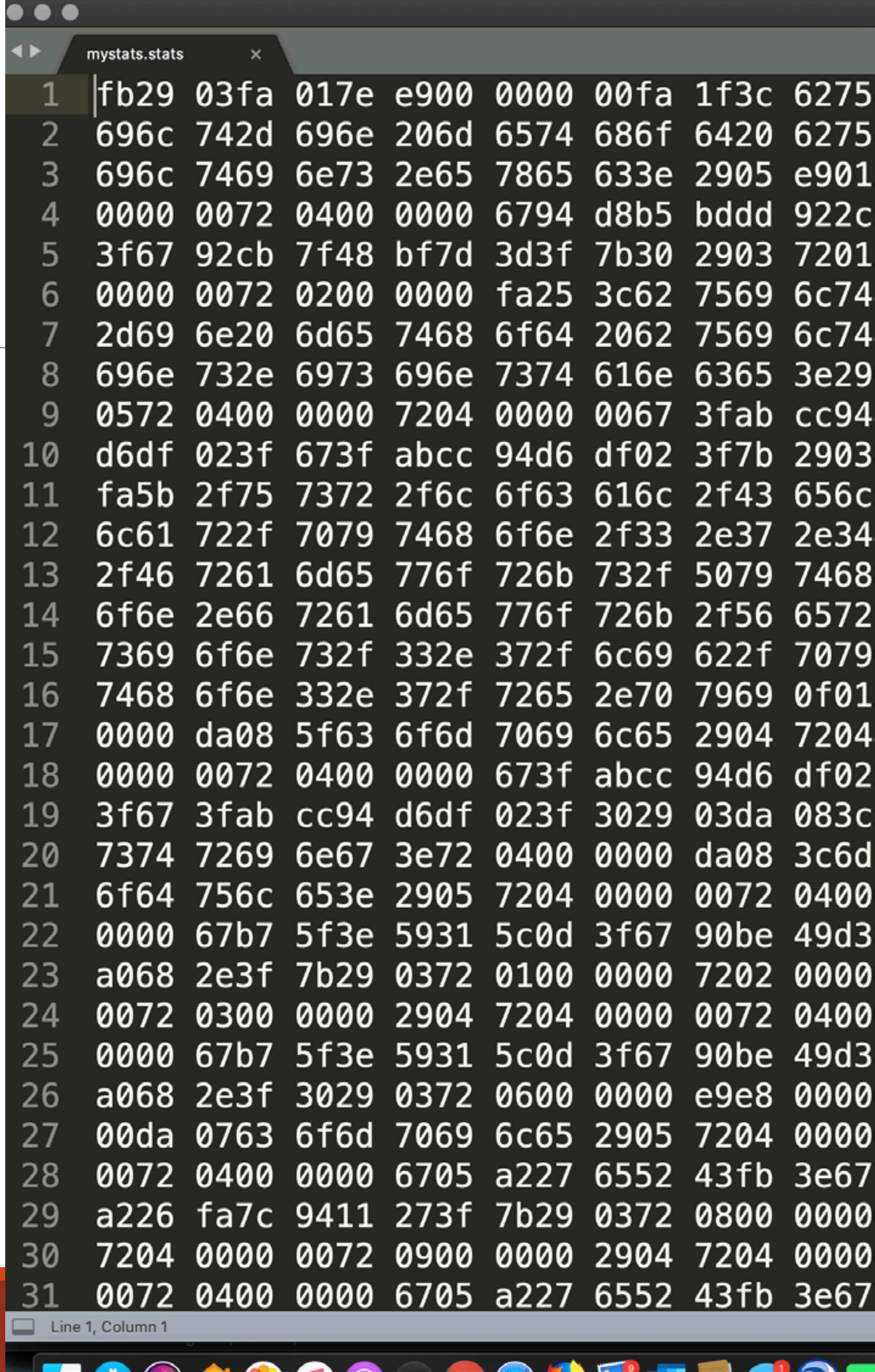
Did you open the file?

It’s unintelligible

This needs translation

pstats.Stats (a statistics object) to the rescue

- Invoke its constructor
- Pass the filename as a parameter to the constructor
- An optional second parameter, the output stream, defaults to sys.stdout



Try it!

```
import cProfile
import re
import pstats

cProfile.run('re.compile("foo|bar")', 'restats')
p = pstats.Stats('restats')
p.print_stats()
```

test2.py

Whoops, I almost forgot…

The Stats class has a variety of methods for manipulating and printing the data saved in a profile results file

strip_dirs() removes the extraneous path from all the module names

sort_stats() sorts the entries according to the standard module/line/name string that is printed

```
from pstats import SortKey

p.strip_dirs().sort_stats(SortKey.STDNAME).print_stats()
```

* https://news.artnet.com/market/yoshitomo-nara-hong-kong-1671325

Sorting results

The sort_stats( ) method accepts a string or a SortKey enum

The parameter identifies the basis of the sort

We can provide multiple keys that are applied in order.

Valid String Arg	Valid enum Arg	Meaning
`'calls'`	SortKey.CALLS	call count
`'cumulative'`	SortKey.CUMULATIVE	cumulative time
`'cumtime'`	N/A	cumulative time
`'file'`	N/A	file name
`'filename'`	SortKey.FILENAME	file name
`'module'`	N/A	file name
`'ncalls'`	N/A	call count
`'pcalls'`	SortKey.PCALLS	primitive call count
`'line'`	SortKey.LINE	line number
`'name'`	SortKey.NAME	function name
`'nfl'`	SortKey.NFL	name/file/line
`'stdname'`	SortKey.STDNAME	standard name
`'time'`	SortKey.TIME	internal time
`'tottime'`	N/A	internal time

Profiling a module – Command line

This is the syntax from the command line:

```
python -m cProfile [-o output_file] [-s sort_order] filename.py
```

-o is optional, if not provided the output will be displayed then and there.

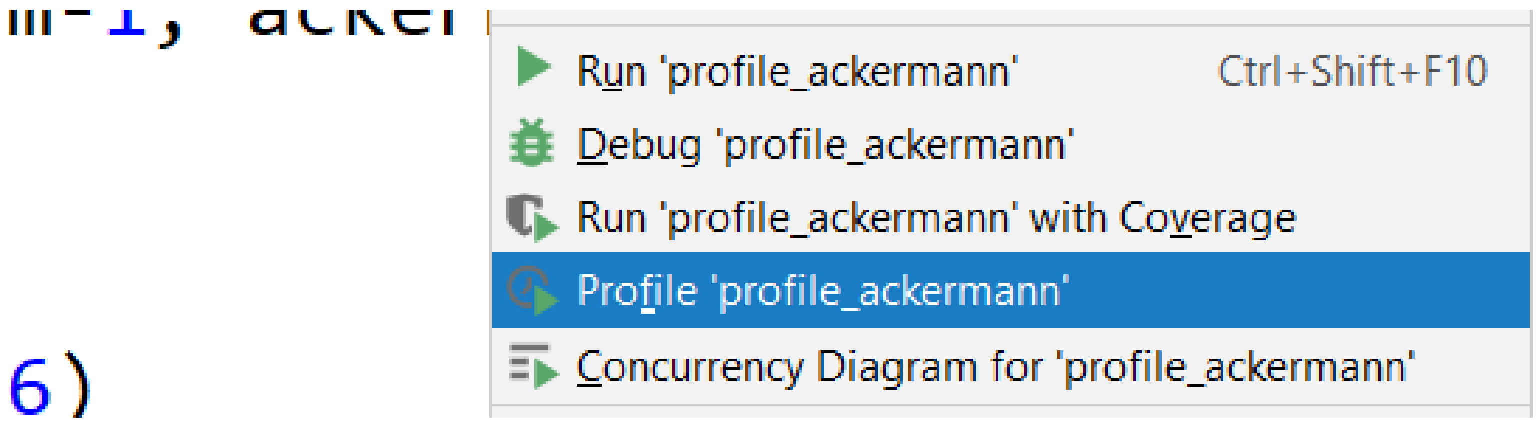
-s is the field by which we can sort the results

Example: profiles profile_ackermann.py sorted by function name, output directly to console

```
python -m cProfile -s name profile_ackermann.py
```

Profiling in Python using PyCharm

To profile in PyCharm, right click anywhere in your code and select “Profile ‘your module name’”

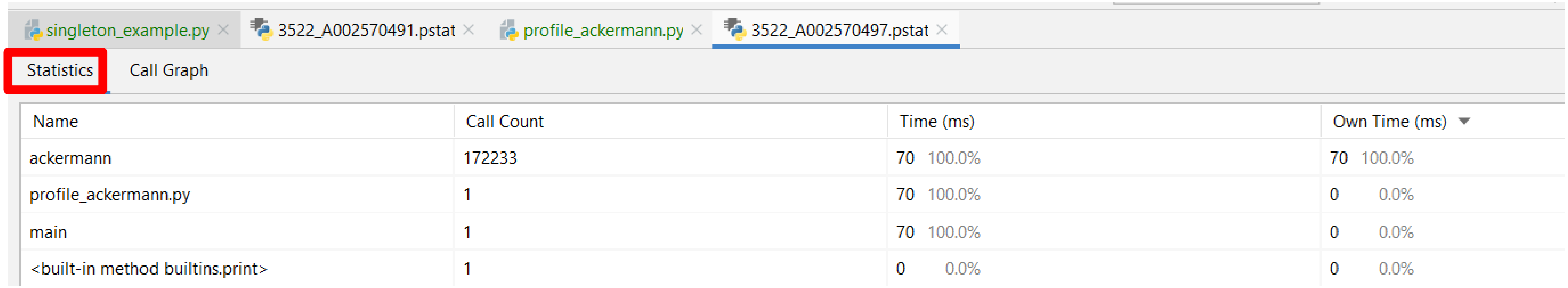


Profiling in Python using PyCharm

A new tab with the statistics of your code will appear with extension .pstat.

Select it to see stats including:

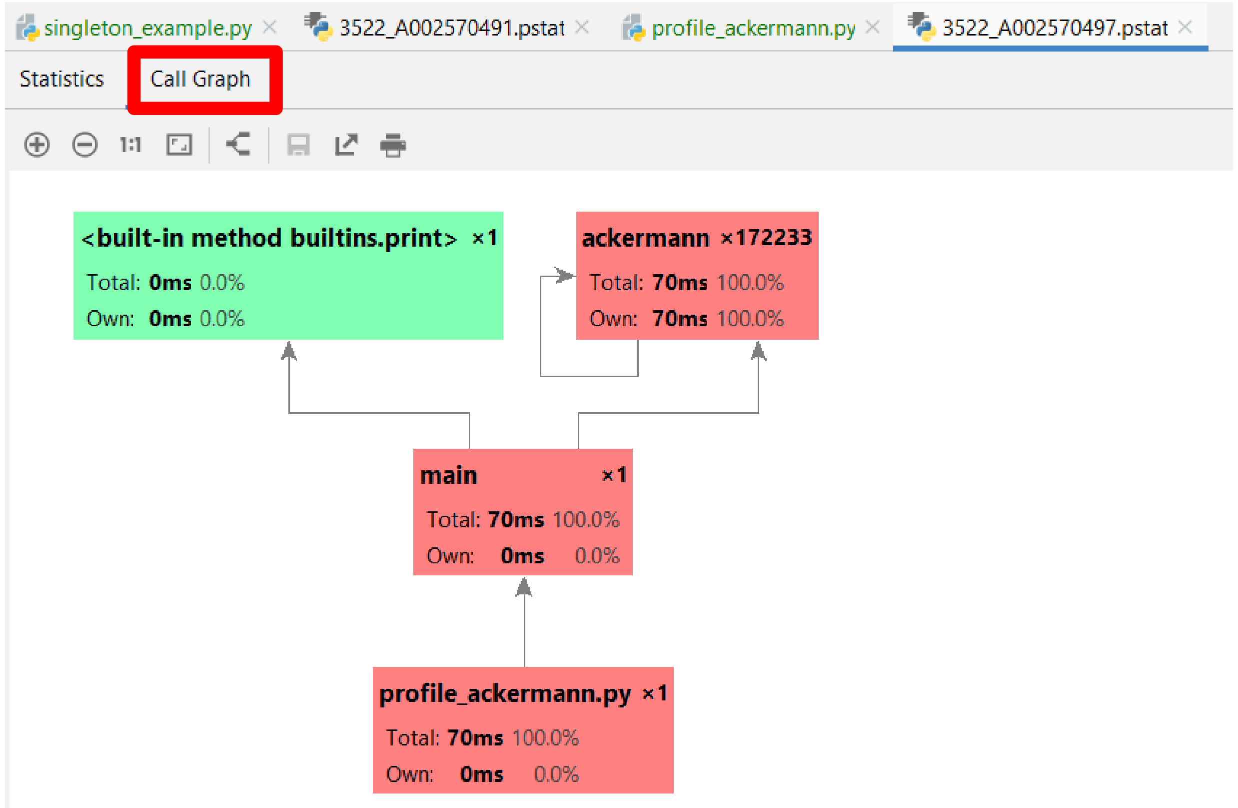
- Function name
- Call count
- Time (Time in function + children function time), Own time (Time in function)



Profiling in Python using PyCharm

The call graph shows a visual representation of the calls made on your code

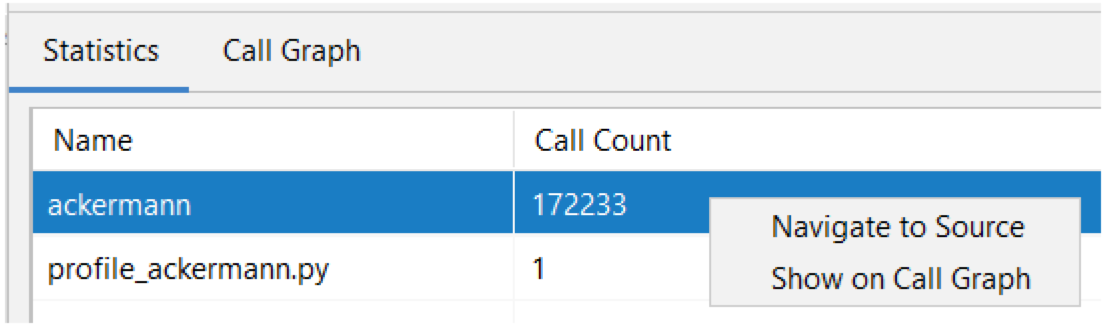
It includes the same information as the statistics tab
- Time
- Number of calls
- etc



Profiling in Python using PyCharm

To navigate to the source code of a certain function:

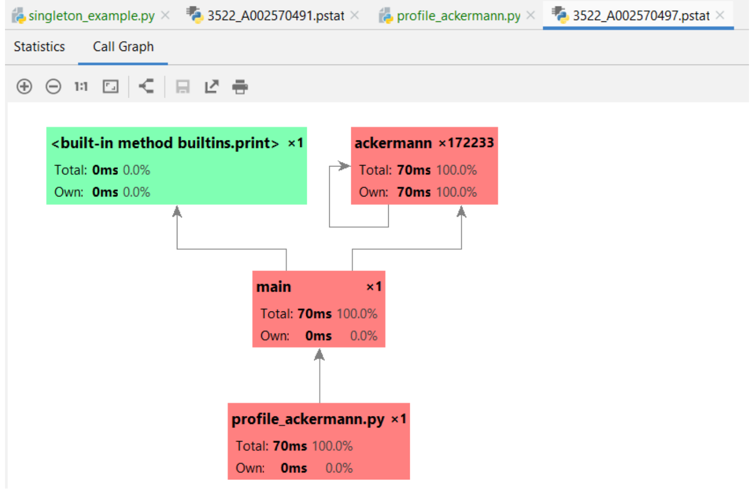
1. Right-click the corresponding entry on the Statistics tab,
2. choose Navigate to Source from the context menu



View the results: call graph

To navigate to the call graph of a certain function:

1. Right-click the corresponding entry on the Statistics tab
2. choose Show on Call Graph from the context menu.
3. The Call Graph tab opens with the function in question highlighted



Note the color codes on the Call Graph

- The functions marked **red** consume more time
- The fastest functions are **green**

Challenge Time

Let's test a function called the Ackermann function

This recursive function takes a very long time. Start by invoking it with Ackermann(1, 2). Dare to try Ackermann(3, 6)

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

* https://en.wikipedia.org/wiki/Ackermann_function

Challenge Time

Let’s test a function called the Ackermann function

This recursive function takes a very long time. Start by invoking it with Ackermann(1, 2). Dare to try Ackermann(3, 6)

```
import sys

def ackermann(m, n):
    if m == 0:
        return n + 1
    elif m > 0 and n == 0:
        return ackermann(m-1, 1)
    elif m > 0 and n > 0:
        return ackermann(m-1, ackermann(m, n-1))
```

```
def main():
    result = ackermann(3,6)
    print(result)
```

Start the profiling session

Do you blow your stack?

Modify your code to permit more recursive calls

The sys module has a handy dandy function for this

The highest possible limit is platform-dependent

This should be done with care, because an overly high limit can lead to a crash.

```
import sys
current = sys.getrecursionlimit() // For fun
sys.setrecursionlimit(limit)
```

Let's try it out and discuss

Grab the code samples from today: profile_ackermann.py

Navigate to folder containing the python file:

Use cProfile from command line: `python -m cProfile -s name profile_ackermann.py`

How long did this take?

How many function calls took place?

- Primitive?
- Recursive?

How many different functions were invoked to execute your code?

What else did you see that looks helpful or 'neat'?

Profiling a block of code

```
import cProfile, psats, io
from pstats import SortKey
pr = cProfile.Profile()
pr.enable()
# … code to be profiled comes here …
pr.disable()
s = io.StringIO()
sortby = SortKey.CUMULATIVE
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
ps.print_stats()  # print to the StringIO output stream ‘s’.
print(s.getvalue()) # print the output stream ‘s’
```

profile_code_fragment.py, profile_code_fragment_2.py

Want to learn more?

Check out the pstats module

https://docs.python.org/3.7/library/profile.html#module-pstats

The API is quite small and easy to learn

In fact, the API for cProfile is easy to learn

https://docs.python.org/3.7/library/profile.html#module-cProfile

A (very) brief look at Memory Profiling

Memory profiling looks at the memory footprint of your objects.

This is useful for identifying memory leaks.

A memory leak occurs when a program starts holding on to more and more memory over time.

This can be an issue especially on mobile devices and software deployed to low-end hardware.

We'll be using the Pympler module for this.

Let's install it.

```
pip3 install pympler
```

Memory profiling a block of code

```
from pympler import tracker

tr = tracker.SummaryTracker()
# .. code that uses objects that need to be tracked ..
tr.print_diff()
```

```
                  types |   # objects |   total size
======================= | =========== | ============
                    str |        2412 |    109.45 KB
                   list |        2466 |    106.57 KB
                    int |         178 |      2.46 KB
                   dict |           3 |    228     B
 function (store_info) |           1 |     68     B
                   cell |           2 |     40     B
                 method |           1 |     32     B
                  float |          -2 |    -32     B
                   code |          -2 |   -138     B
                  tuple |         -39 |  -1596     B
```

Check out the doc's for more information:

[* https://pympler.readthedocs.io/en/latest/intro.html#usage-examples](https://pympler.readthedocs.io/en/latest/intro.html#usage-examples)

auction_entities_wrap.py

Memory profiling objects of a specific Class

```
from pympler import classtracker


class_tr = classtracker.ClassTracker()
class_tr.track_class(ClassName)
class_tr.create_snapshot() # Before Snapshot
# .. code that uses objects that need to be tracked ..
class_tr.create_snapshot() # After Snapshot
class_tr.stats.print_summary()
```

Limitations: accuracy

There is a fundamental problem with profilers

- The underlying clock is only “ticking” once every millisecond
- No measurements can be more accurate than 0.001 seconds

There is also some lag:

- Delay from when an event is dispatched until the profiler’s call to get the time actually *gets* the state of the clock
- Functions called many times, or which call many sub-functions, tend to accumulate this error
- The error is usually less than the accuracy of the click (0.001 seconds) but it can accumulate and become significant

Design Patterns

What are Design Patterns

Common design solutions to common architectural problems.

Think of these as recipes.

While developing OOP programs you might come across common problems.

We have already seen a bunch of these!

What are Design Patterns?

Consider some common problems that you may come across when developing OO programs.

QUESTION 1
How do I iterate over a collection of objects without modifying the collection itself?

What are Design Patterns?

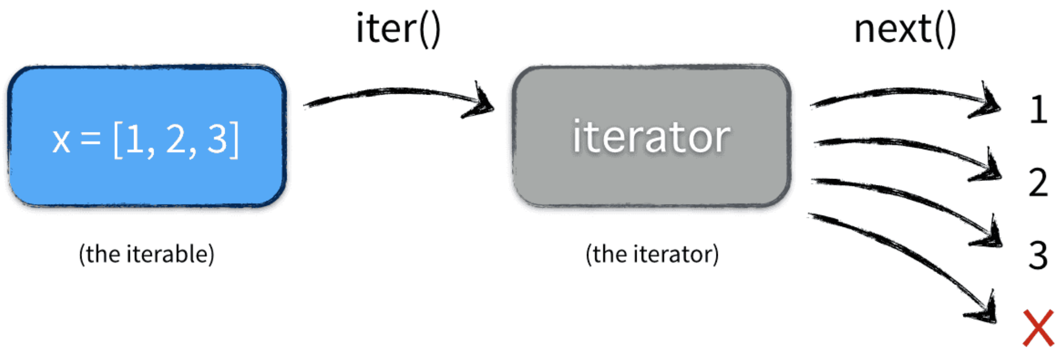
Consider some common problems that you may come across when developing OO programs.

QUESTION 1

How do I iterate over a collection of objects without modifying the collection itself?

Solution: The Iterator Pattern

Create a separate class known as the iterator which holds a reference to the iterables and can iterate over it separately. Give the iterables a method which returns an instance of the iterator.



What are Design Patterns?

Consider some common problems that you may come across when developing OO programs.

QUESTION 2

How do I notify a bunch of different kinds of object if the state of one part of the system changes without coupling that part of the system with the rest?

What are Design Patterns?

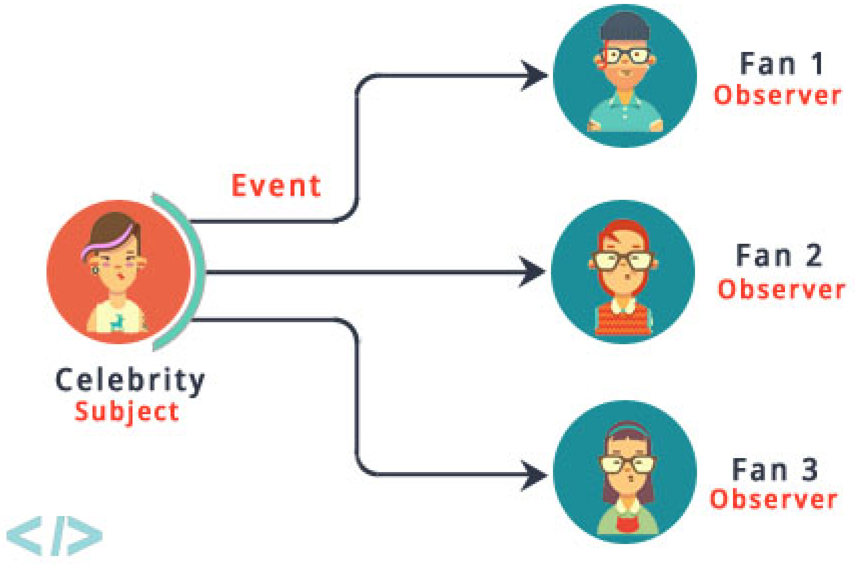
Consider some common problems that you may come across when developing OO programs.

QUESTION 2

How do I notify a bunch of different kinds of object if the state of one part of the system changes without coupling that part of the system with the rest?

SOLUTION: The Observer Pattern

Have all the different kinds of objects implement a common interface, make the '*core*' (the system that changes and notifies other objects) dependent on the interface and not the objects itself.



Design Patterns - Advantages

- Don’t re-invent the wheel, use a proven solution instead
- Are abstract and can be applied to different problems
- Communicate ideas and concepts between developers
- Language agnostic. Can be applied to most (if not all) OOP programs.

Design Patterns - Disadvantages

- Can make the system more complex making the system harder to maintain. Patterns are deceptively ‘simple’.
- The system may suffer from pattern overload.
- All patterns have some disadvantages and add constraints to a system. As a result a developer may need to add a constraint they did not plan for.
- Do not lead to direct code re-use.

Categorizing Design Patterns

- **Behavioural**

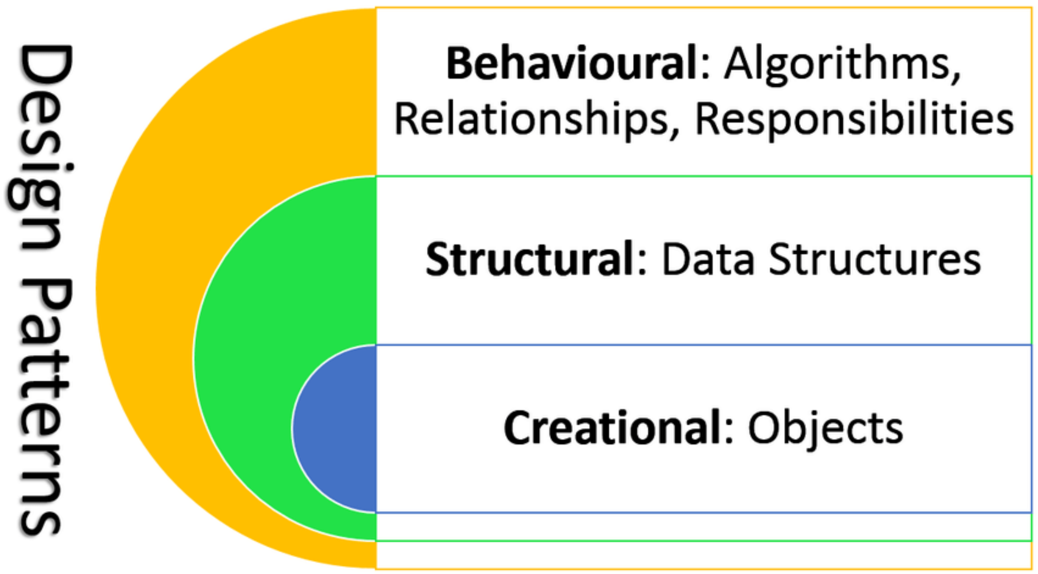
Focused on communication and interaction between objects. How do we get objects talking to each other while minimizing coupling?

- **Structural**

How do classes and objects combine to form structures in our programs? Focus on architecting to allow for maximum flexibility and maintainability.

- **Creational**

All about class instantiation. Different strategies and techniques to instantiate an object, or group of objects



Picking a Pattern

Step 1	• Understand the problem you are facing in terms of dependencies, modularity and abstract concepts.
Step 2	• Identify if this is a behavioural, structural or creational issue?
Step 3	• Are there any constraints that I need to follow?
Step 4	• Is there a simpler solution that works? If not, pick a pattern.

Singleton

WHEN ONE IS ENOUGH

Design pattern: a really easy one!

Creational design pattern

Sometimes we want to guarantee that only a **single instance** of a class will ever exist

We want to prevent more than one copy from being constructed

We must write code that enforces this rule

We want to employ the **Singleton Design Pattern**

Singleton pattern

1. **Instantiates** the object on its first use
2. **Ideally hides** a private initializer
3. **Reveals** a public get_instance function that returns a reference to a static instance of the class
4. **Provides** “global” access to a single object

Why/how do we use it?

Use the singleton pattern **when you need to have one and only one object of a type** in a system.

Singleton is a globally accessible class where we guarantee only a single instance is created

That's it.

Really, that's all there is to it.

Code sample (so easy!)

```python
class MySingleton:
    __instance = None

    @staticmethod
    def get_instance():
        if MySingleton.__instance is None:
            MySingleton()
        return MySingleton.__instance

    def add_num(self, n):
        MySingleton.__instance.data += n

    def __init__(self):
        if MySingleton.__instance is not None:
            raise Exception("This class is a singleton!")
        else:
            MySingleton.__instance = self
            MySingleton.__instance.data = 0
```

```python
s = MySingleton.get_instance()
s.add_num(6)
print(s, s.data)

s1 = MySingleton.get_instance()
s1.add_num(3)
print(s1, s1.data)

s2 = MySingleton.get_instance()
s2.add_num(2)
print(s2, s2.data)
```

singleton_example.cpp

Application – Game screen management

Game has multiple screens

- Start, gameplay UI, game over, store, etc

Different screens must be able to be displayed at various places in the code

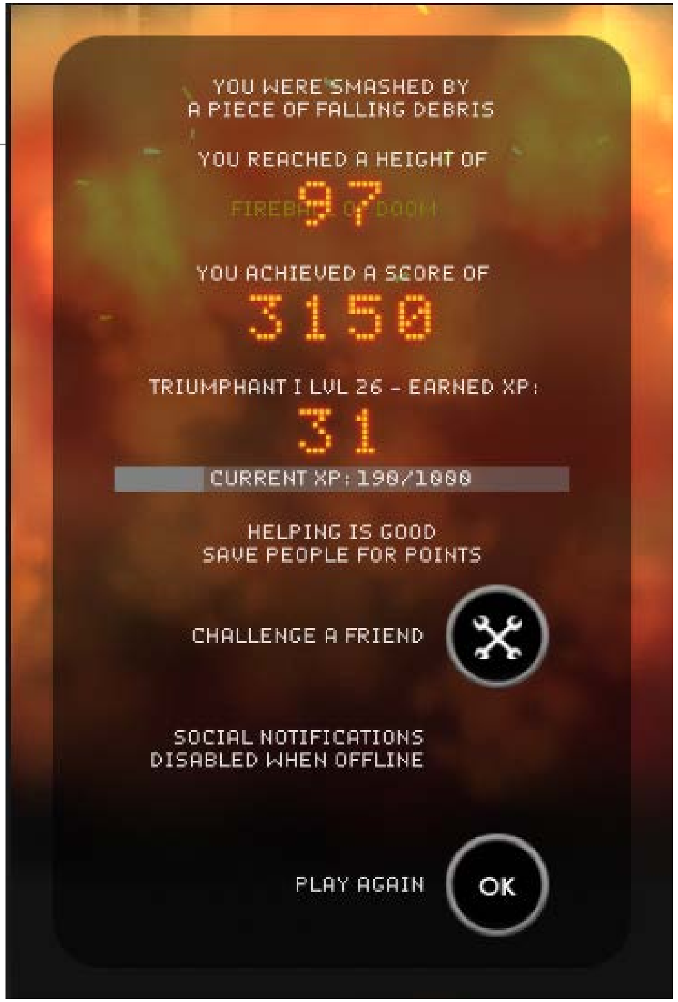
- Store class wants to show store screens
- Gameplay logic wants to show start/gameplay/game over
- Settings logic wants to show settings screen

Need a central place to call to load specific screens on demand

Mechanic Panic – Singleton screens example

GameState enum: Main menu
ScreenManager.getInstance().show(MainMenu);

GameState enum: Gameplay
ScreenManager.getInstance().show(Gameplay);



GameState enum: GameOver
ScreenManager.getInstance().show(Gameover);

That’s it for today!

No quiz on Friday

Next quiz is next Friday