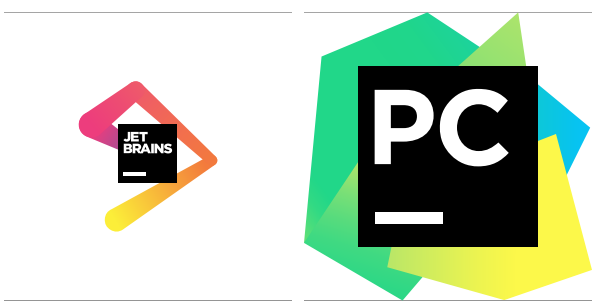
# Profiling the Unprofilable

PyCon APAC 2016, Seoul

Dmitry Trofimov

@DmitryTrofimov

14 August 2016



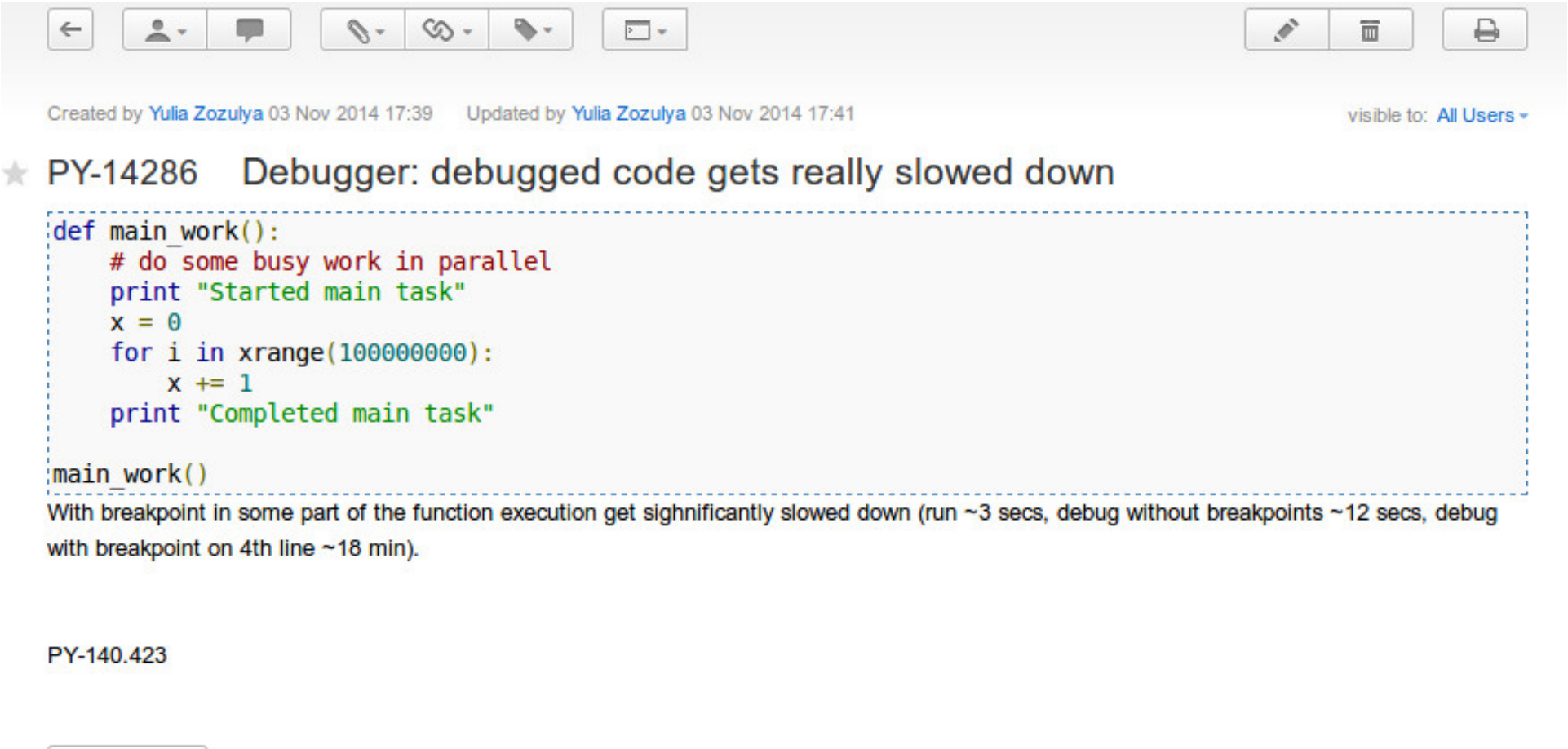
# About me

- Dmitry Trofimov
- I work for JetBrains.
- I am the Team Lead and developer of PyCharm IDE
- I am interested in Python run-time: execution, debugging, profiling

*The best theory is inspired by practice. The best practice is inspired by theory.*
*Donald Knuth*

https://youtrack.jetbrains.com/issue/PY-14286

# https://youtrack.jetbrains.com/issue/PY-14286



Created by Yulia Zozulya 03 Nov 2014 17:39 Updated by Yulia Zozulya 03 Nov 2014 17:41 visible to: All Users

## PY-14286 Debugger: debugged code gets really slowed down

```
def main_work():
    # do some busy work in parallel
    print "Started main task"
    x = 0
    for i in xrange(100000000):
        x += 1
    print "Completed main task"

main_work()
```

With breakpoint in some part of the function execution get sighnificantly slowed down (run ~3 secs, debug without breakpoints ~12 secs, debug with breakpoint on 4th line ~18 min).

PY-140.423

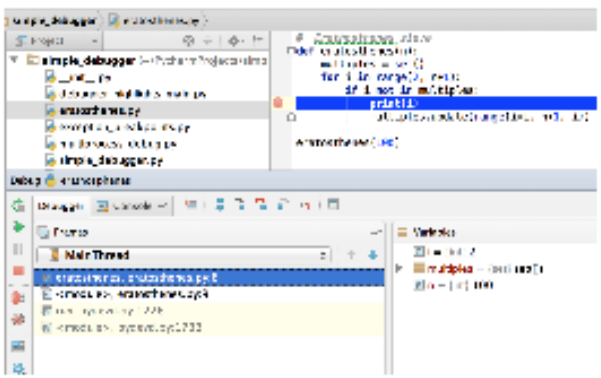
# PyCharm debugger

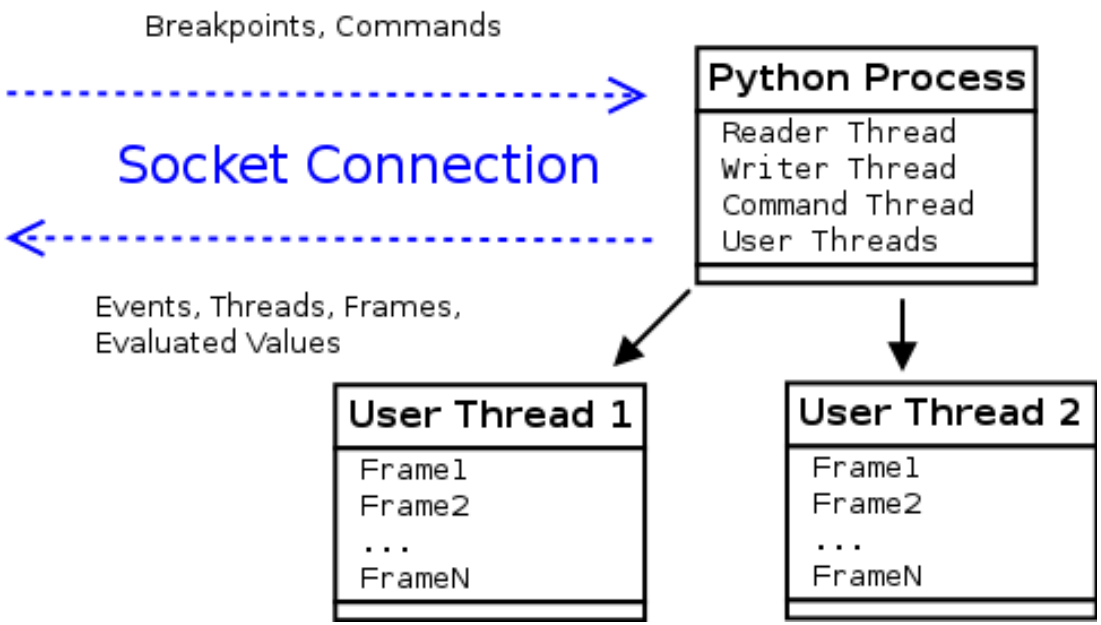
https://github.com/fabioz/PyDev.Debugger

# PyCon APAC 2015: Python Debugger Uncovered

https://www.youtube.com/watch?v=DHf-6gW3-qs

# PyCharm Debugger





# Trace Function

**sys.settrace(*tracefunc*)¶**

Set the system's trace function, which allows you to implement a Python source code debugger in Python. settrace() for each thread being debugged.

Trace functions should have three arguments: *frame*, *event*, and *arg*. *frame* is the current stack frame. *event* is a string: 'call', 'line', 'return', 'exception', 'c_call', 'c_return', or 'c_exception'. *arg* depends on the event type.

The trace function is invoked (with *event* set to 'call') whenever a new local scope is entered; it should return a reference to a local trace function to be used that scope, or None if the scope shouldn't be traced.

```
def trace_function(self, frame, event, arg):
    filename = frame.f_code.co_filename
    line = frame.f_lineno

    breakpoints_for_file = debugger.breakpoints.get(filename)
    if breakpoints_for_file:
        breakpoint = breakpoints_for_file[line]
        ...
```

# Issue PY-14286

## Type: Performance

## Subsystem: Debugger

| | |
|---|---|
| Run | 3 seconds |
| Debug | 12 seconds |
| Debug + breakpoint | ~18 minutes |

# Sample code

```
def main_work():
    # do some busy work in parallel
    print("Started main task")
    x = 0
    for i in xrange(100000000):
        x += 1
    print("Completed main task")

main_work()
```

# 1. Reproducing

# 2. Analysis

- Normal run
- Debug without breakpoints

- Debug + BP in the function
- Debug + BP in the same file
- ~~Debug + BP in some other file~~
  = 4 different cases

# Debugger

| Execution Mode | Speed |
|---|---|
| Run | fast |
| Debug: no BP | fast |
| Debug: BP in main_work | slow |
| Debug: BP in file | slow |

*You can't improve what you can't measure*
*W. Edwards Deming*

# 3. Measurement

```
import time


def main_work():
    print("Started main task")
    x = 0
    t = time.time()
    for i in range(1000000):
        x += 1
    print("Completed main task in %0.3fs" % (time.time() - t))

main_work()
```

# PyCharm debugger

| Execution Mode | Duration |
|---|---|
| Run | 0.09s |
| Debug: no BP | 0.11s |
| Debug: BP in main_work | 9s |
| Debug: BP in file | 9s |

# 3. Something to compare with

## pdb

- Although less functional, but does almost the same
- Written in Python
- Is in standard library

# 4. Benchmarking

| | **PyCharm** | **pdb** |
|---|---|---|
| Run | 0.09s | 0.09s |
| Debug: no breakpoints | 0.11s | 0.10s |
| Debug: BP in main_work | 9s | 5s |
| Debug: BP in the file | 9s | 5s |

# 5. Find a bottleneck

## Profile

# What is Profile?

# Profile

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

Let's use a Python profiler.

# Python profilers

- cProfile
- yappi
- line_profiler

# cProfile

- Part of the standard library
- Written in C
- The default choice for Python profiling

# yappi

- Written in C
- Does the same as cProfile
- Can profile separate threads

# line_profiler

- Written in Cython
- Provides statistics about lines

# cProfile

Let's use it.

Unprofilable?

# cProfile

cProfile provides deterministic profiling of Python programs.

What does 'deterministic profiling' means?

# There are 2 major types of profilers

- Tracing(deterministic) profilers
- Sampling(Statistical) profilers

# Tracing profilers

aka deterministic profilers aka Event-based profilers
Events: events like c_{call,return,exception},
python_{call,return,exception}.

**sys.setprofile(*profilefunc*)¶**

Set the system's profile function, which allows you to implement a Python source code profiler in Python. See chapter The Python Profilers for more information on the Python profiler. The system's profile function is called similarly to the system's trace function (see `settrace()`), but it isn't called for each executed line of code (only on call and return, but the return event is reported even when an exception has been set).

| | |
|---|---|
| cProfile | setprofile |
| yappi | setprofile |
| line_profiler | settrace |

Unprofilable?

# Sampling(Statistical) profilers

- Collect call stack samples regularly
- Less accurate and specific
- Add almost no overhead

# Python statistical profilers

- statprof
- plop
- Intel Vtune Amplifier
- vmprof

# statprof

- Written in Python
- Last update: 2012

# plop

- Written in Python
- "a work in progress"

# Intel Vtune Amplifier

- Has low overhead
- Profiles Python and native code
- Proprietary and not open-source
- Doesn't work on MacOSX

# vmprof

- Written in C -- has low overhead
- Works on Linux, Mac, Windows
- Supports Python 2.7, 3.5, and PyPy
- Free and open-source

# Use vmprof

Bottleneck found, what next?

# 6. Optimization

- Design
- Algorithms and data structures

# Why debug without breakpoints works so much faster?

```
def trace_function(self, frame, event, arg):
    filename = frame.f_code.co_filename
    line = frame.f_lineno

    breakpoints_for_file = debugger.breakpoints.get(filename)
    if breakpoints_for_file:
        breakpoint = breakpoints_for_file[line]
        if breakpoint is not None:
            ... # handle breakpoint
        return trace_function
    else:
        return None
```

**sys.settrace(*tracefunc*)¶**

The trace function is invoked (with *event* set to `'call'`) whenever a new local scope is entered; it should return a reference to a local trace function to be used that scope, or `None` if the scope shouldn't be traced.

The local trace function should return a reference to itself (or to another function for further tracing in that scope), or `None` to turn off tracing in that scope.

```
def trace_function(self, frame, event, arg):
    filename = frame.f_code.co_filename
    line = frame.f_lineno

    breakpoints_for_file = debugger.breakpoints.get(filename)

    for breakpoint in breakpoints_for_file.itervalues():
        if breakpoint.func_name == frame.f_code.co_name:
            break
    else:
        return None

    ...
```

# A. Algorithmic optimization

| | **Baseline** | **Optimization A** |
|---|---|---|
| Run | 0.09s | 0.09s |
| Debug: no breakpoints | 0.11s | 0.11s |
| Debug: BP in main_work | 9s | 9s |
| Debug: BP in the file | 9s | 0.11s |

# 6. Optimization

- Design
- Algorithms and data structures
- Source code

# Line profiling

# B. Source optimization

| | **Opt A** | **Opt A+B** |
|---|---|---|
| Run | 0.09s | 0.09s |
| Debug: no breakpoints | 0.11s | 0.11s |
| Debug: BP in main_work | 9s | 8s |
| Debug: BP in the file | 0.11s | 0.11s |

# 6. Optimization

- Design
- Algorithms and data structures
- Source code
- Build
- Compile
- Assembly
- Run time

Is optimization reached it's limit?

Go beyond Python!

# Rewrite everything in C?

- Compatibility with Jython, IronPython, PyPy etc
- Avoid code duplication
- Python is much better language then C

# Cython

- Static compiler for Python
- Gives the combined power of Python and C

```
def primes(int kmax):
    cdef int n, k, i
    cdef int p[1000]
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            p[k] = n
            result.append(n)
        n = n + 1
    return result
```

```
def trace_dispatch(self, frame, str event, arg):
        cdef str filename;
        cdef bint is_exception_event;
        cdef bint has_exception_breakpoints;
        cdef bint can_skip;
        cdef PyDBAdditionalThreadInfo info;
        cdef int step_cmd;
        cdef int line;
        cdef str curr_func_name;
        cdef bint exist_result;
```

# C. Cython compile optimization

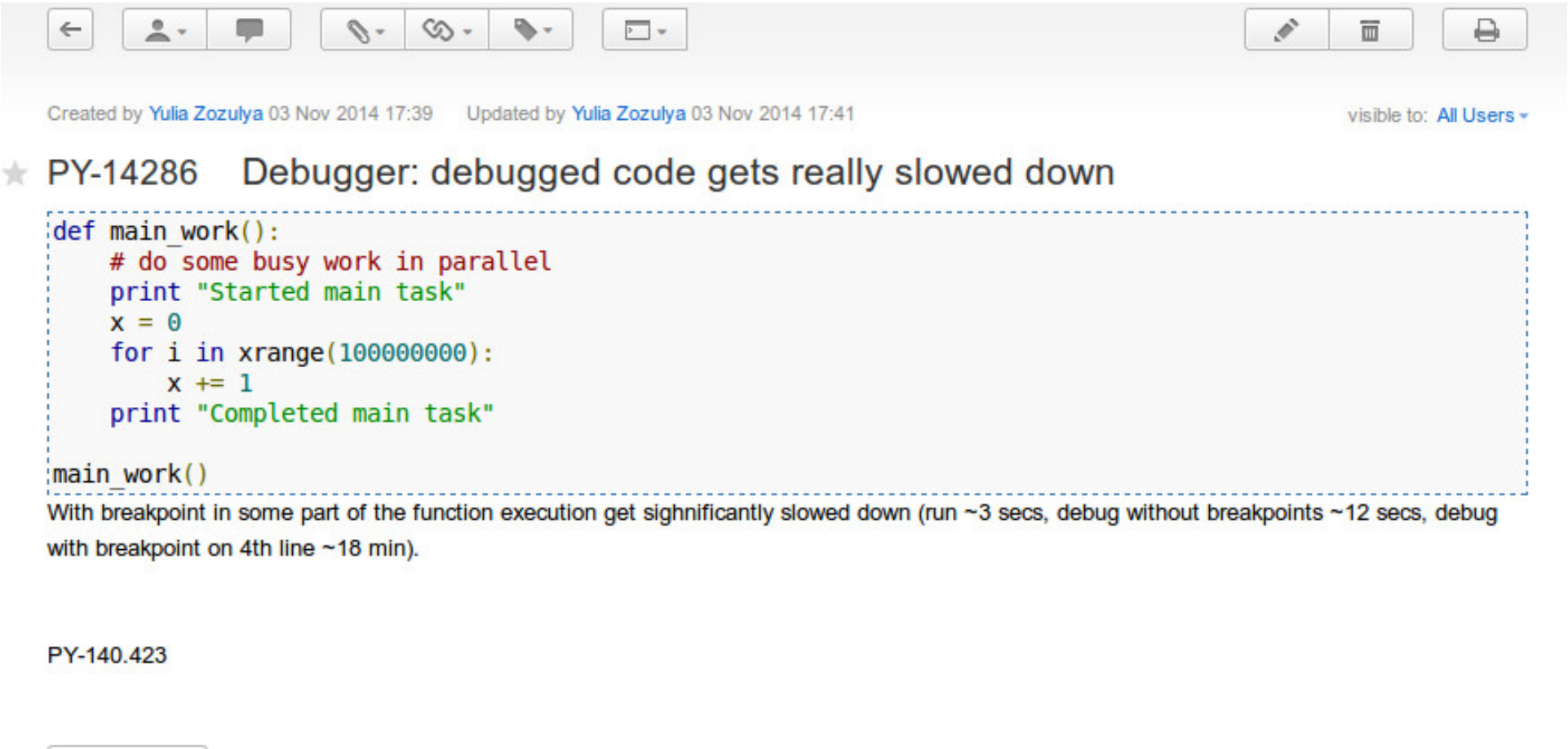
| | **Opt A+B** | **Opt A+B+C** |
|---|---|---|
| Run | 0.09s | 0.09s |
| Debug: no breakpoints | 0.11s | 0.11s |
| Debug: BP in main_work | 8s | 4s |
| Debug: BP in the file | 0.11s | 0.11s |

# 7. Compare results

| | **Baseline** | **pdb** | **A+B+C** |
|---|---|---|---|
| Run | 0.09s | 0.09s | 0.09s |
| Debug: no breakpoints | 0.11s | 0.10s | 0.11s |
| Debug: BP in main_work | 9s | 5s | 4s |
| Debug: BP in the file | 9s | 5s | 0.11s |

Unprofilable again?

# https://youtrack.jetbrains.com/issue/PY-14286



Created by Yulia Zozulya 03 Nov 2014 17:39 Updated by Yulia Zozulya 03 Nov 2014 17:41 visible to: All Users

## PY-14286 Debugger: debugged code gets really slowed down

```
def main_work():
    # do some busy work in parallel
    print "Started main task"
    x = 0
    for i in xrange(100000000):
        x += 1
    print "Completed main task"

main_work()
```

With breakpoint in some part of the function execution get sighnificantly slowed down (run ~3 secs, debug without breakpoints ~12 secs, debug with breakpoint on 4th line ~18 min).

PY-140.423

https://www.python.org/dev/peps/pep-0523

# PEP 523 -- Adding a frame evaluation API to CPython

| | |
|---|---|
| **PEP:** | 523 |
| **Title:** | Adding a frame evaluation API to CPython |
| **Author:** | Brett Cannon <brett at python.org>, Dino Viehland <dinov at microsoft.com> |
| **Status:** | Draft |
| **Type:** | Standards Track |
| **Created:** | 16-May-2016 |
| **Post-History:** | 16-May-2016 |

# Conclusion

- Use profilers to find bottlenecks in your code
- There are different profilers: each has own upsides
- Start to optimize things from higher level to lower
- To optimize Python on a lower level use Cython

## Links

- vmprof.readthedocs.io
- cython.org
- github.com/fabioz/PyDev.Debugger

# Q&A