

# Statistical Profiling

(and other neat features of the `sys` module)

# Outline

1. Why statistical profiling?
2. Various python tools available.
3. Details of `ox_profile` statistical profiler.

# Outline

1. Why statistical profiling?
2. Various python tools available.
3. Details of `ox_profile` statistical profiler.

## **Goal:**

- learn how to use `ox_profile`, other profilers

# Outline

1. Why statistical profiling?
2. Various python tools available.
3. Details of `ox_profile` statistical profiler.

## Goal:

- learn how to use `ox_profile`, other profilers
- or write your own!

# What is profiling?

From python.org:

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

Profiling is useful to find out which parts of program are slow.

# Deterministic Profiling

```
>>> import cProfile, re # could also use pure python `profile`  
>>> cProfile.run('re.compile("foo|bar")')
```

gives:

197 function calls (192 primitive calls) in 0.002 seconds

Ordered by: standard name

ncalls	totttime	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)

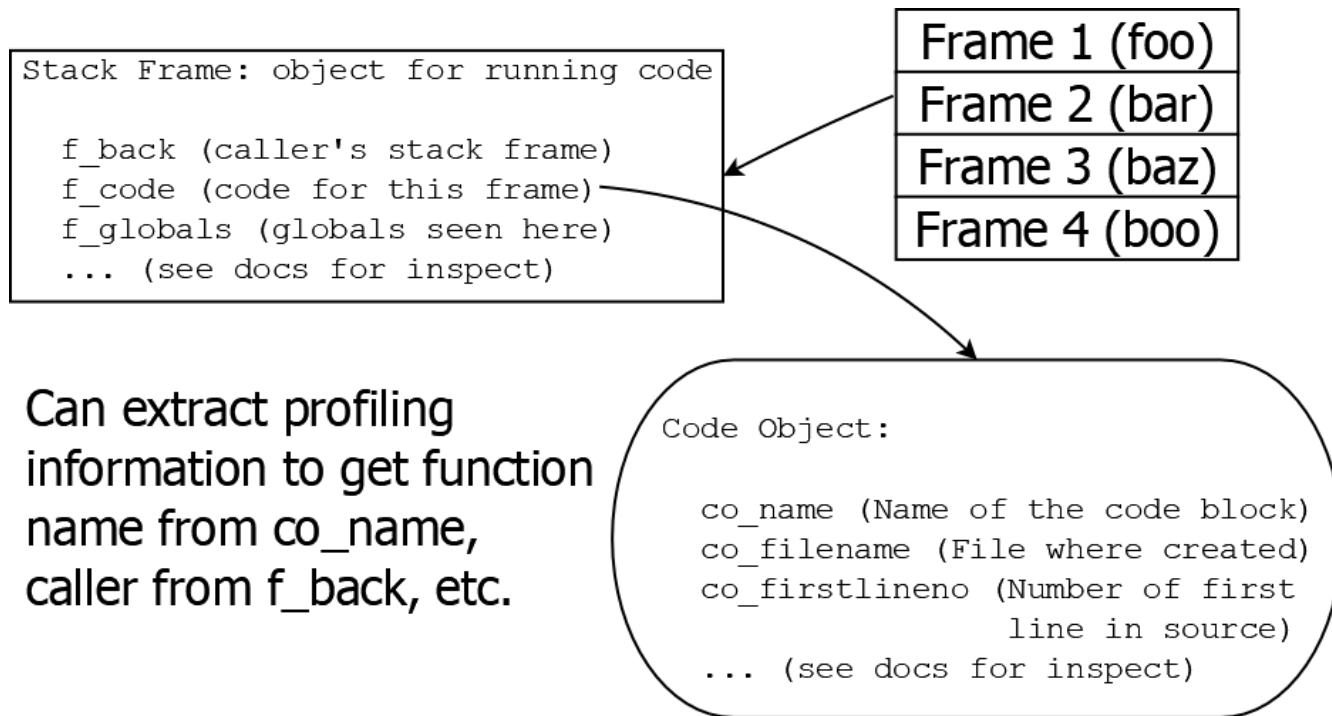
# Deterministic profiling hooks

The `sys` module provides:

- `setprofile`: execute function on each call.
- `settrace`: like `setprofile` + each line

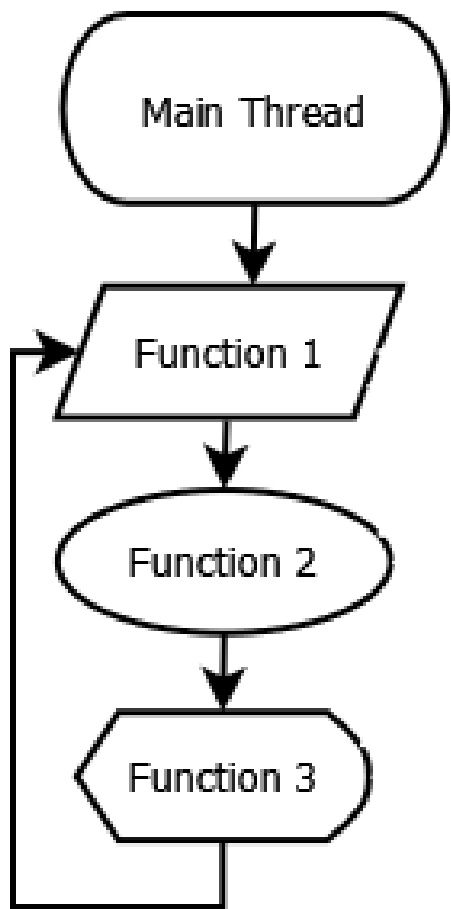
Used as a hook to inspect stack frame.

# Python Stack Frame

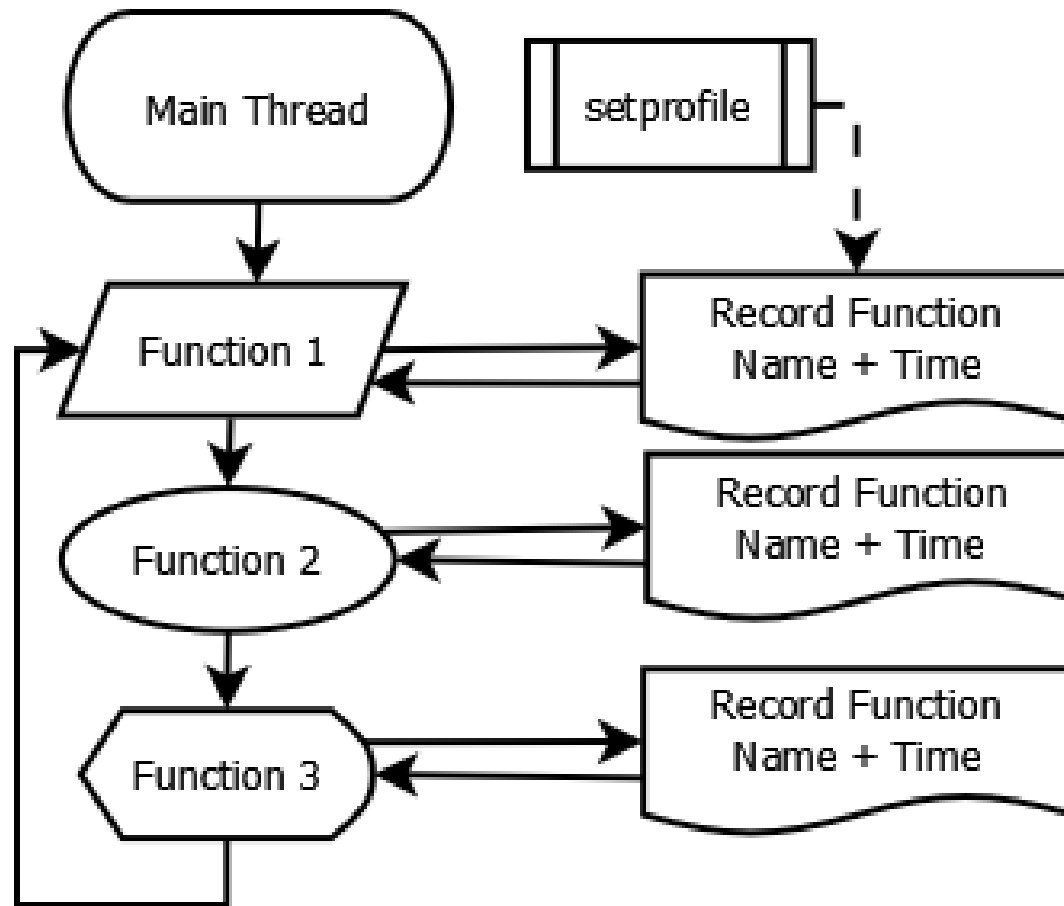




# Program Diagram



# Program Diagram w/Profiler



# Drawbacks to deterministic profiling

1. **SLOW**: hooks run on each line or each call!

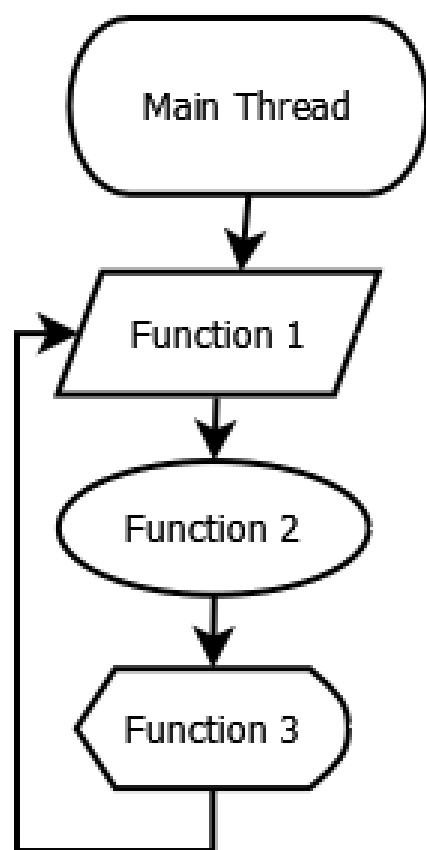
# Drawbacks to deterministic profiling

1. **SLOW**: hooks run on each line or each call!
2. See #1: usually can't profile in production.

# Drawbacks to deterministic profiling

1. **SLOW**: hooks run on each line or each call!
2. See #1: usually can't profile in production.
3. Not thread-aware:  
"it must be registered using `settrace()`  
for each thread being debugged"  
(similar for `setprofile()`).

# Statistical Profiler (Main Thread)

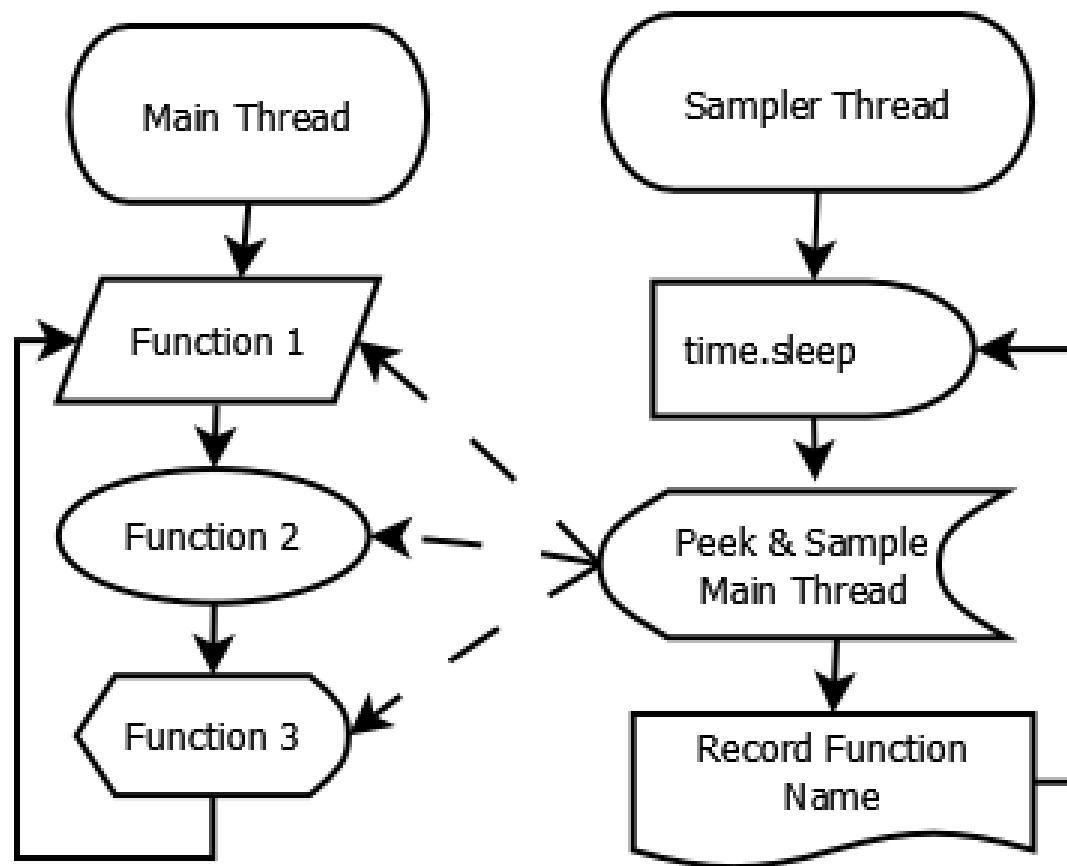


Statistical Profiler periodically checks program state.

Records a "sample" of what function program is running.

Only runs "occasionally" so overhead is low.

# Statistical Profiler (Sampling)



# POSIX timer as Sampler

- Low level timer interrupts your process
- Periodically check call stack + record it.
- `stat_prof`, `plop` (not available on Windows).



# POSIX ptrace as Sampler

- Threads via ptrace.
- Lets one program control/inspect other.
- Main/Sampler are different programs!
- pyflame (not available on Windows).

# Why do we care about non-POSIX?

# Why do we care about non-POSIX?

1. Nice to have "portable" implementation.

# Why do we care about non-POSIX?

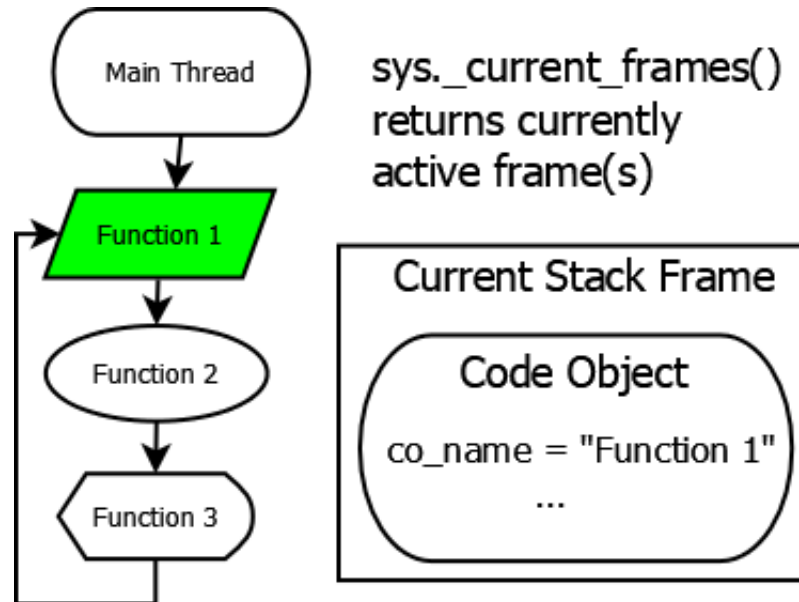
1. Nice to have "portable" implementation.
2. Using only python features may be easier to understand/modify.
  - Understandable implementation essential if run in production.

# Why do we care about non-POSIX?

1. Nice to have "portable" implementation.
2. Using only python features may be easier to understand/modify.
  - Understandable implementation essential if run in production.
3. Consumer software (e.g., games) often run in Windows.

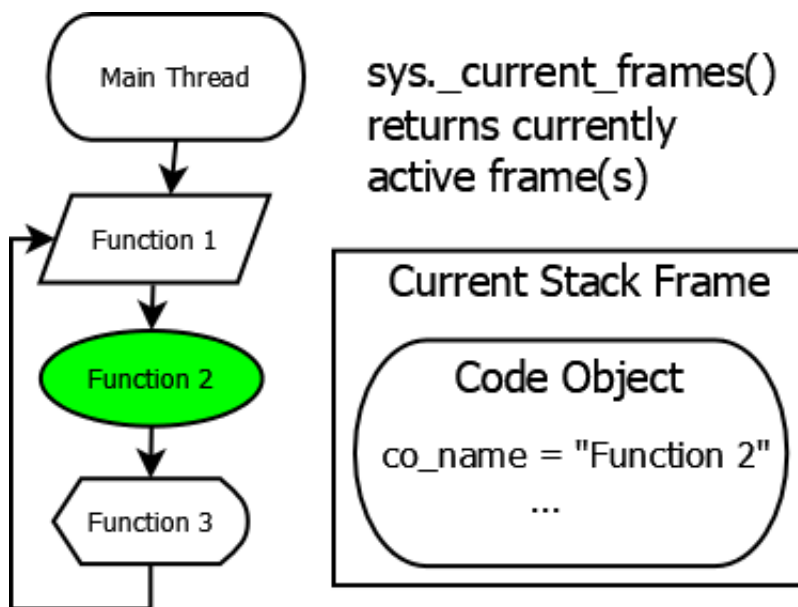
# Python `sys._current_frames`

- lists each thread's current frame.
- `ox_profile`, `pprofile`, (works on windows!)



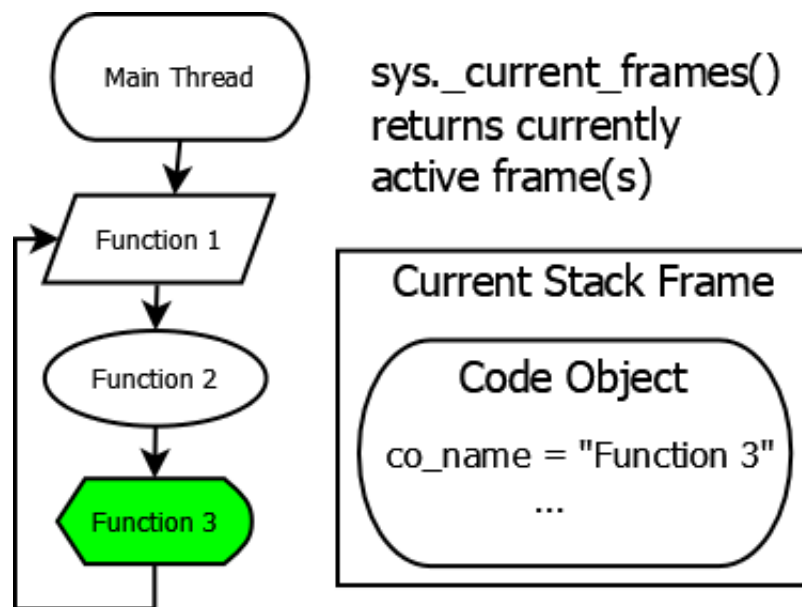
# \_current\_frames (Function 2)

- lists each thread's current frame.
- `ox_profile`, `pprofile`, (works on windows!)



# `_current_frames` (Function 3)

- lists each thread's current frame.
- `ox_profile`, `pprofile`, (works on windows!)





# Logistics: How do we profile?

Install via the usual

```
$ pip install ox_profile
```

Then start your python interpreter and:

```
>>> from ox_profile.core.launchers import SimpleLauncher
>>> profiler = SimpleLauncher.launch() # Create + start
>>> # call some functions
>>> print(profiler.show())             # Show results
>>> profiler.cancel()                 # Turn off
```

# Sampling loop

Oversimplified illustration of sampler loop:

```
while 1:  
    time.sleep(self.interval)  
    for id, frame in sys._current_frames().items():  
        self.my_db[frame.f_code.co_name] += 1
```

Sampling loop runs in separate thread.

# Sampling Interval

Control overhead based on sampling interval:

```
>>> profiler.set_interval(5) # changes profiler.interval
```

- 5 second sampling = negligible overhead.
- Trade-off between various goals:
  - accuracy, collection time, overhead
- May want to also add random jitter

# Profiler output

```
>>> profiler.show()
```

Output shows:

- Function name (and module).
- How many times function was seen.

Function	Hits	%
-----		
send(requests.sessions)	10082	2.8
request(requests.api)	7710	2.1
get(requests.api)	7710	2.1
<listcomp>(__main__)	7710	2.1
__call__(ox_profile.core.sampling)	7710	2.1
request(requests.sessions)	7698	2.1
send(requests.adapters)	6897	1.9
_make_request(urllib3.connectionpool)	6893	1.9
urlopen(urllib3.connectionpool)	6893	1.9
connect(urllib3.connection)	5602	1.5

# Basic theory of `sys._current_frames`

1. Start thread doing mostly `time.sleep`.
2. Periodically wake + `sys._current_frames()`
3. Record current stack frame(s) info.

In theory, you can write simple statistical profiler with just these.

# Basic theory of `sys._current_frames`

1. Start thread doing mostly `time.sleep`.
2. Periodically wake + `sys._current_frames()`
3. Record current stack frame(s) info.

In theory, you can write simple statistical profiler with just these.

In theory, there is no difference between theory and practice.

# Basic theory of `sys._current_frames`

1. Start thread doing mostly `time.sleep`.
2. Periodically wake + `sys._current_frames()`
3. Record current stack frame(s) info.

In theory, you can write simple statistical profiler with just these.

In theory, there is no difference between theory and practice.

- But, in practice, there is.

# Using `sys._current_frames`

In theory, iter `sys._current_frames()` + record.

```
for dummy_frame_id, frame in (sys._current_frames().items()):  
    self.my_db.record(measure_tool(frame))
```



# Using `sys._current_frames`

In theory, iter `sys._current_frames()` + record.

```
for dummy_frame_id, frame in (sys._current_frames().items()):  
    self.my_db.record(measure_tool(frame))
```

What could go wrong?

# Using `sys._current_frames`

In theory, iter `sys._current_frames()` + record.

```
for dummy_frame_id, frame in (sys._current_frames().items()):  
    self.my_db.record(measure_tool(frame))
```

What could go wrong?

- thread context could switch

# Using sys.\_current\_frames

In theory, iter `sys._current_frames()` + record.

```
for dummy_frame_id, frame in (sys._current_frames().items()):  
    self.my_db.record(measure_tool(frame))
```

What could go wrong?

- thread context could switch
  - attempt to read stale frame = **CRASH!**

# Using `sys._current_frames`

In theory, iter `sys._current_frames()` + record.

```
switch_interval = sys.getswitchinterval()
sys.setswitchinterval(10000)

for dummy_frame_id, frame in (sys._current_frames().items()):
    self.my_db.record(measure_tool(frame))

sys.setswitchinterval(switch_interval)
```

`sys.setswitchinterval` prevents context switch.

Now, what could go wrong?

# Using `sys._current_frames`

In theory, iter `sys._current_frames()` + record.

```
switch_interval = sys.getswitchinterval()
sys.setswitchinterval(10000)

for dummy_frame_id, frame in (sys._current_frames().items()):
    self.my_db.record(measure_tool(frame))

sys.setswitchinterval(switch_interval)
```

`sys.setswitchinterval` prevents context switch.

Now, what could go wrong?

- Exception prevents switch interval being reset!

# Using `sys._current_frames`

In theory, iter `sys._current_frames()` + record.

```
switch_interval = sys.getswitchinterval()
try:
    sys.setswitchinterval(10000)
    for dummy_frame_id, frame in (
        sys._current_frames().items()):
        self.my_db.record(measure_tool(frame))
finally:
    sys.setswitchinterval(switch_interval)
```

`sys.setswitchinterval` prevents context switch.

Need `try/finally` to protect thread logic.

Now, what could go wrong?

# Using `sys._current_frames`

In theory, iter `sys._current_frames()` + record.

```
switch_interval = sys.getswitchinterval()
try:
    sys.setswitchinterval(10000)
    for dummy_frame_id, frame in (
        sys._current_frames().items()):
        self.my_db.record(measure_tool(frame))
finally:
    sys.setswitchinterval(switch_interval)
```

`sys.setswitchinterval` prevents context switch.

Need `try/finally` to protect thread logic.

Now, what could go wrong?

- We are just getting started.

# Recording a sample

Simple function to record a sample in dict:

```
def record(self, measurement):  
    record = self.my_db.get(measurement.name, 0)  
    self.my_db[measurement.name] = record + 1
```

- keep dict of measurements
- find + increment hit for given function

What could go wrong?



# Recording sample changing dict

- Imagine getting profile results

```
def record(self, measurement):  
    record = self.my_db.get(measurement.name, 0)  
    self.my_db[measurement.name] = record + 1
```

- If record being called
  - change dict during iteration = **CRASH!**

# Recording a sample

```
def __init__(self):  
    self.db_lock = threading.Lock()  
  
def record(self, measurement):  
    with self.db_lock:  
        record = self.my_db.get(measurement.name, 0)  
        self.my_db[measurement.name] = record + 1
```

- Need to use `threading.Lock`
  - Must check lock in viewing `self.my_db`
- Prevents simultaneous access to `my_db`

What could go wrong?

# Recording a sample

```
def __init__(self):  
    self.db_lock = threading.Lock()  
  
def record(self, measurement):  
    with self.db_lock:  
        record = self.my_db.get(measurement.name, 0)  
        self.my_db[measurement.name] = record + 1
```

- Need to use `threading.Lock`
  - Must check lock in viewing `self.my_db`
- Prevents simultaneous access to `my_db`

What could go wrong?

- Unknown unknowns:
  - use `faulthandler.enable()` at startup

# Additional Issues

- Sampler thread still alive on exit program
  - Set `self.daemon = True` for sampler
- Minimum sleep time ~ 1--10 milliseconds
  - For `time.sleep` or wait on a thread event.
  - Granularity about 10 milliseconds.

# Using Flask

If you are running a flask web server, do:

```
from ox_profile.ui.flask.views import OX_PROF_BP
app.register_blueprint(OX_PROF_BP)
app.config['OX_PROF_USERS'] = {<admin_user_1>, <admin_user_2>, ...}
```

for easy way to profile production code:

- `ox_profile/unpause`: Start (or unpause)
- `ox_profile/pause`: Pause profiler
- `ox_profile/status`: Shows current results
- `ox_profile/set_interval`: sample frequency

# Additional Uses

What else can you do with `sys._current_frames()`?

# Additional Uses

What else can you do with `sys._current_frames()`?

Major damage that is very hard to track down!

# Additional Uses

What else can you do with `sys._current_frames()`?

Major damage that is very hard to track down!

Be careful!



# Idea: Data Snapshot

- Periodically snapshot specific function data
- Instead of `frame.f_code.co_name`,
- Just look at `frame.f_locals`

# Idea: Data Snapshot

- Periodically snapshot specific function data
- Instead of `frame.f_code.co_name`,
- Just look at `frame.f_locals`
  - Can even modify `f_locals`!

# Idea: Statistical Debugger

- Periodically snapshot debug info
  - full stack backtrace + locals
- Either dump these to logs
- or store locally and report on "failure".

# Idea: Statistical code coverage

- Like `coverage.py`, but in production
- Periodically turn on tracing
- After brief sample, turn back off

# Summary

1. Profiling find bottlenecks + improve your code.

# Summary

1. Profiling find bottlenecks + improve your code.
2. Statistical profiler samples code periodically
  - use to analyze production with low overhead.

## Summary

1. Profiling find bottlenecks + improve your code.
2. Statistical profiler samples code periodically
  - use to analyze production with low overhead.
3. Use `ox_profile` or write your own profiler.

## Summary

1. Profiling find bottlenecks + improve your code.
2. Statistical profiler samples code periodically
  - use to analyze production with low overhead.
3. Use `ox_profile` or write your own profiler.
4. `sys.setprofile`, `sys.settrace`, `sys._current_frames`



# Summary

1. Profiling find bottlenecks + improve your code.
2. Statistical profiler samples code periodically
  - use to analyze production with low overhead.
3. Use `ox_profile` or write your own profiler.
4. `sys.setprofile`, `sys.settrace`, `sys._current_frames`
5. Simple in theory; but be careful with threads:
  - `try/finally`, locks, switch interval, `faulthandler`

# Further investigations

- Slides, ox\_profile, etc. at [https://github.com/emin63/ox\\_profile](https://github.com/emin63/ox_profile)
  - Clone, fork, or file issues for questions.
- About presenter:
  - Name: Emin Martinian
  - Role: software, technology, finance consulting at [www.aocks.com](http://www.aocks.com)
  - Contact: emin.martinian@gmail.com