Fantastic anti-patterns and where to find them: pinpointing performance bottlenecks 🐉

The talk & takeaways

- Real-life case study
- Pinpoint performance bottlenecks with:
  - a sampling profiler
  - flame graphs 🔥
- Performance anti-patterns with remediation
- Neat features in Python's standard library

About me

- Samuel Dion-Girardeau
- Software engineer, linguist
- ❤️ Python!
- Working at Delphia

Case study

- Crypto-assets trading backtesting program.
  - "How would strategy X perform?"
- Using the catalyst algorithmic trading library.
- Prototype, need to be able to iterate fast!

Expected 🙏

```
python backtest.py --start=2016-01 --end=2018-01
```

Then:

- Get a coffee
- Wait ~5 minutes
- Output: "Your strategy made 10x profits 📈!"
    - Future crypto-millionaire 💰

Actual 😬

```
python backtest.py --start=2016-01 --end=2018-01
```

Then:

- Get a coffee
- Wait ~1 hour
- Coffee is cold
- Output: "You lost 90% of your starting capital 📉"
    - Future crypto-bankrupt 💸

What's going on?

- Way too slow! Takes around an hour to run... 😴
- Even when the strategy code isn't doing anything!
- How do we even go about troubleshooting this?
    - Print statements with timing?
- What if the bottleneck is in a third-party library?

Solution: profiling, and flame graphs! 🔥

What is a profiler?

Program that analyses another program at runtime, and reports useful data, e.g. "what code was running".

Two major families:

- Instrumentation
- Sampling

Instrumentation

- Changes the target program with various hooks so it reports data
- Very accurate data points
- Costly
- Can affect the behaviour
- E.g. `coverage`

Sampling

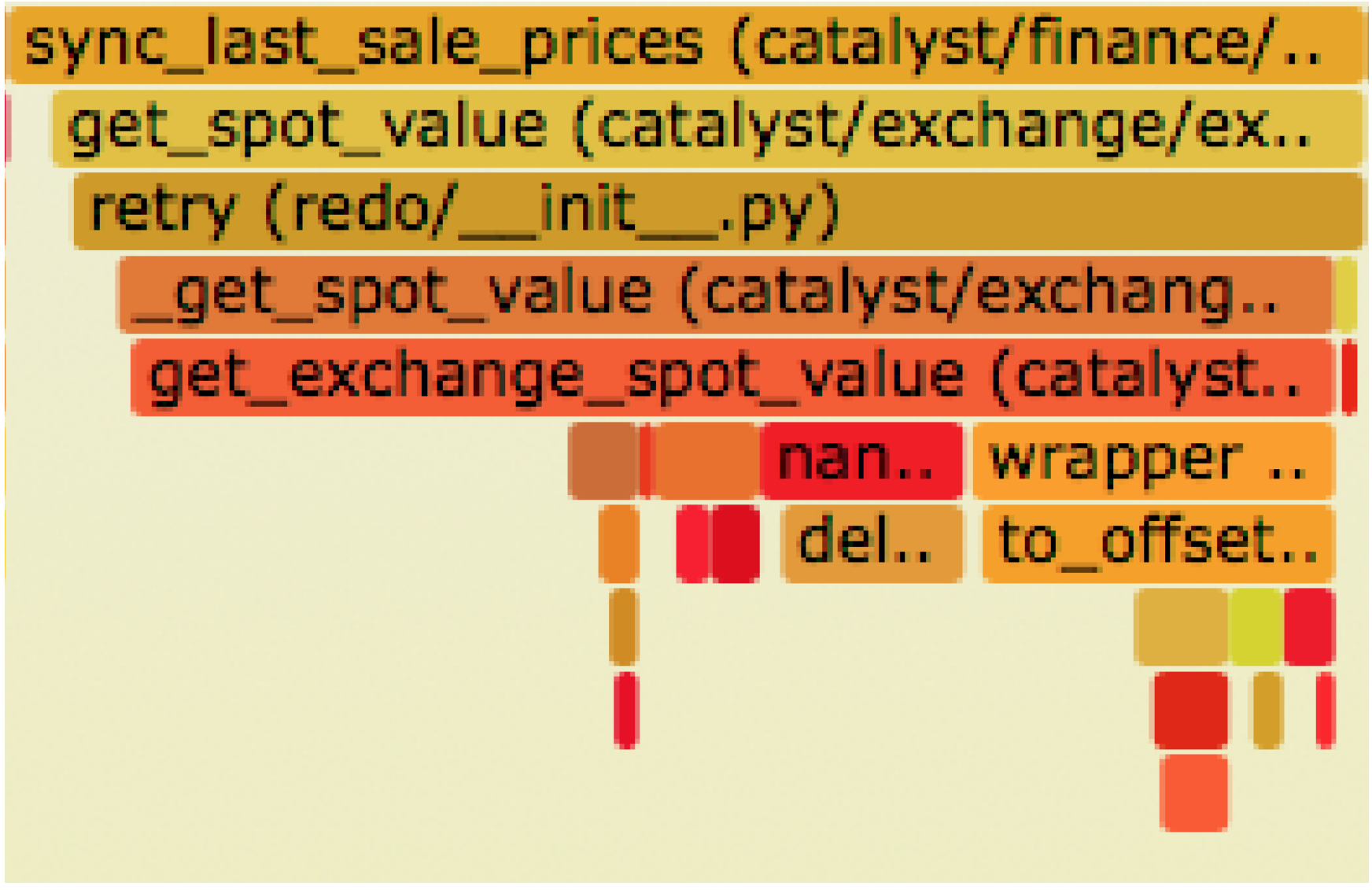
- Statistical, so approximate
- Not costly (can run in prod!)
    - Some CPU/cache overhead
- No code changes, inspect a running program
- E.g. `py-spy`

```
py-spy record \
       --pid 13337 \
       --function \
       --output flame_graph.svg \
       --duration 600 \
       --rate 1000
```

What are flame graphs?

- Visulization tools for profiler output.
- Gives a hierarchical representation of code paths and their relative sampling frequenecy.
- Can be interactive.

How do you read a flame graph?



- Vertical: Stack trace depth
- Horizontal axis: time spent, proportional to parent (⚠️ no order)
- "Gaps" between vertical levels: time spent directly in the function

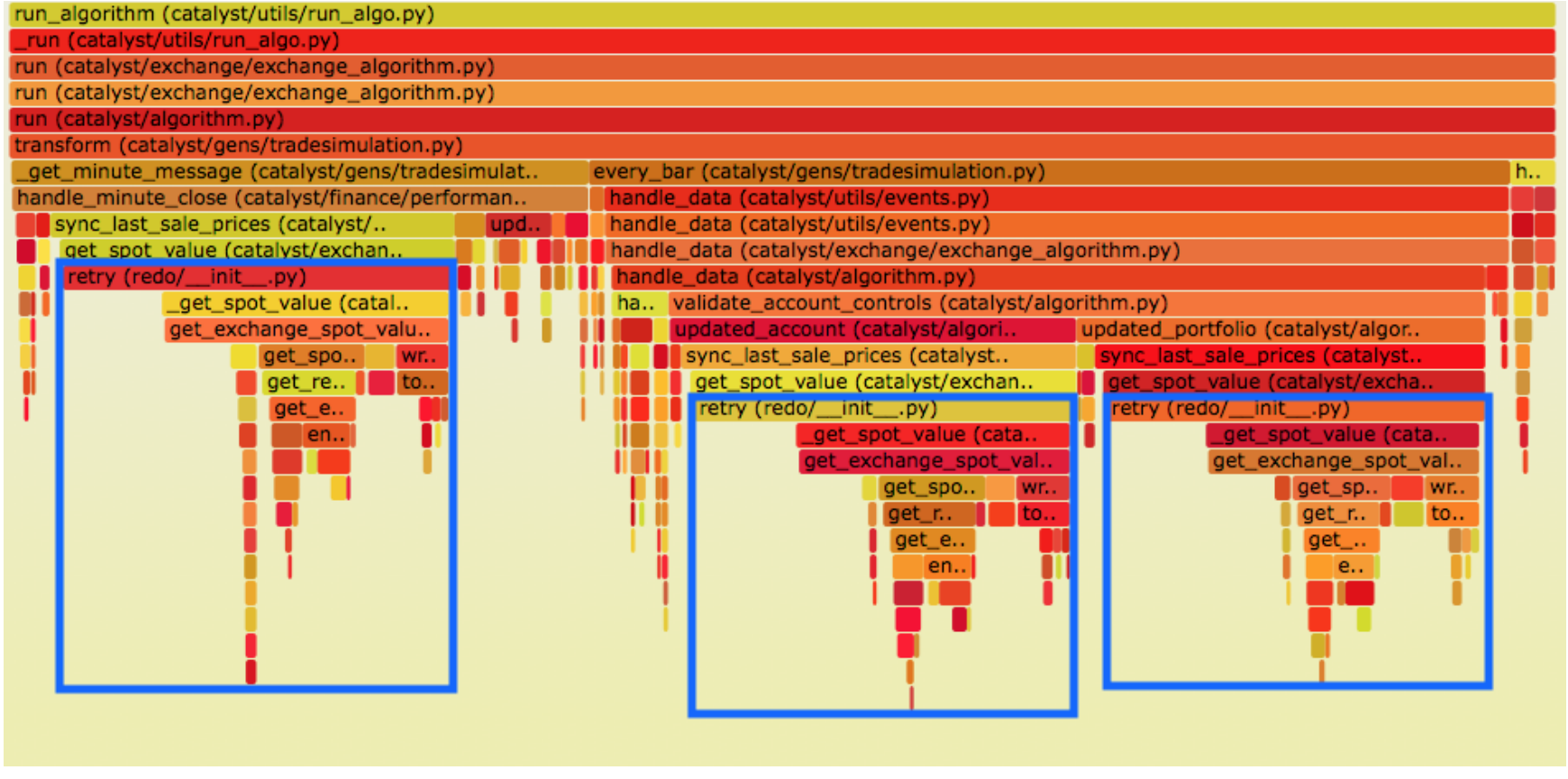
What to look for:

- Large horizontal portions
- Time spent in one function
- Recurring function calls

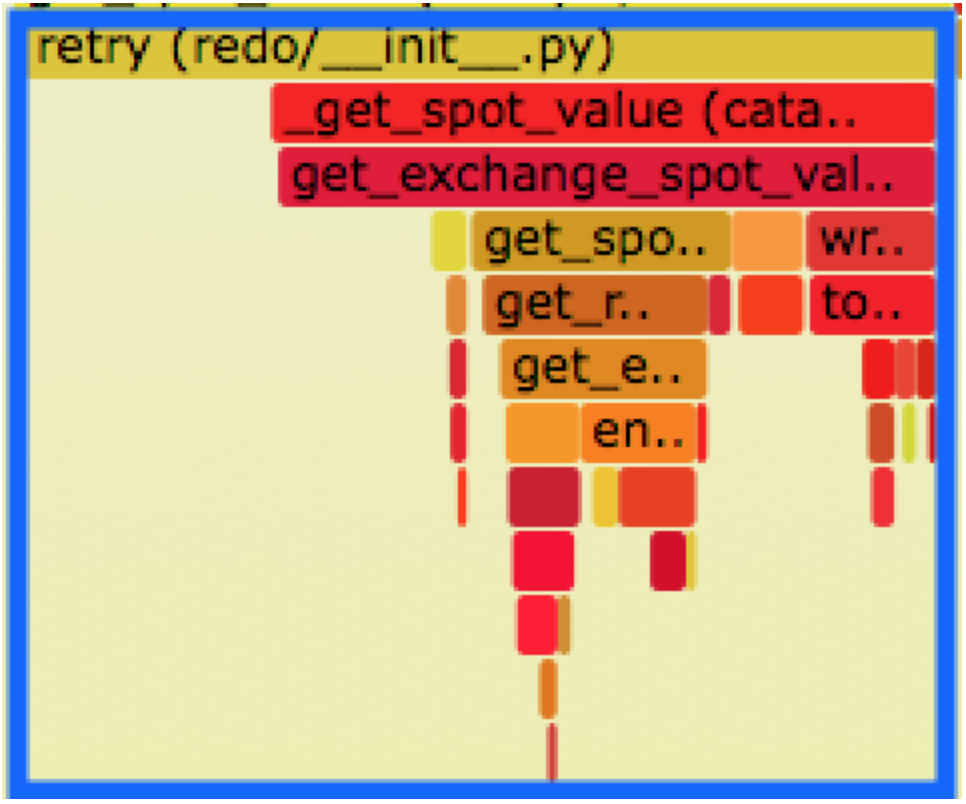
Performance anti-patterns + solutions

Back to case study (finally!)

Result of running `py-spy`:



Zoom, enhance:



`redo.retry()` (simplified)

```
def retry(action, args, kwargs, attempts=5):
    logging.debug(
        "calling %s with args: %s, kwargs: %s" % \
        (action.__name__, args, kwargs)
    )
    for i in attempts:
        try:
            action(*args, **kwargs)
        except:
            ... # Manage exceptions
        else:
            return
```

Anti-pattern: Costly eager string formatting

```
    logging.debug(
        "calling %s with args: %s, kwargs: %s" % \
        (action.__name__, args, kwargs)
    )
```

`debug(msg, *args, **kwargs)`

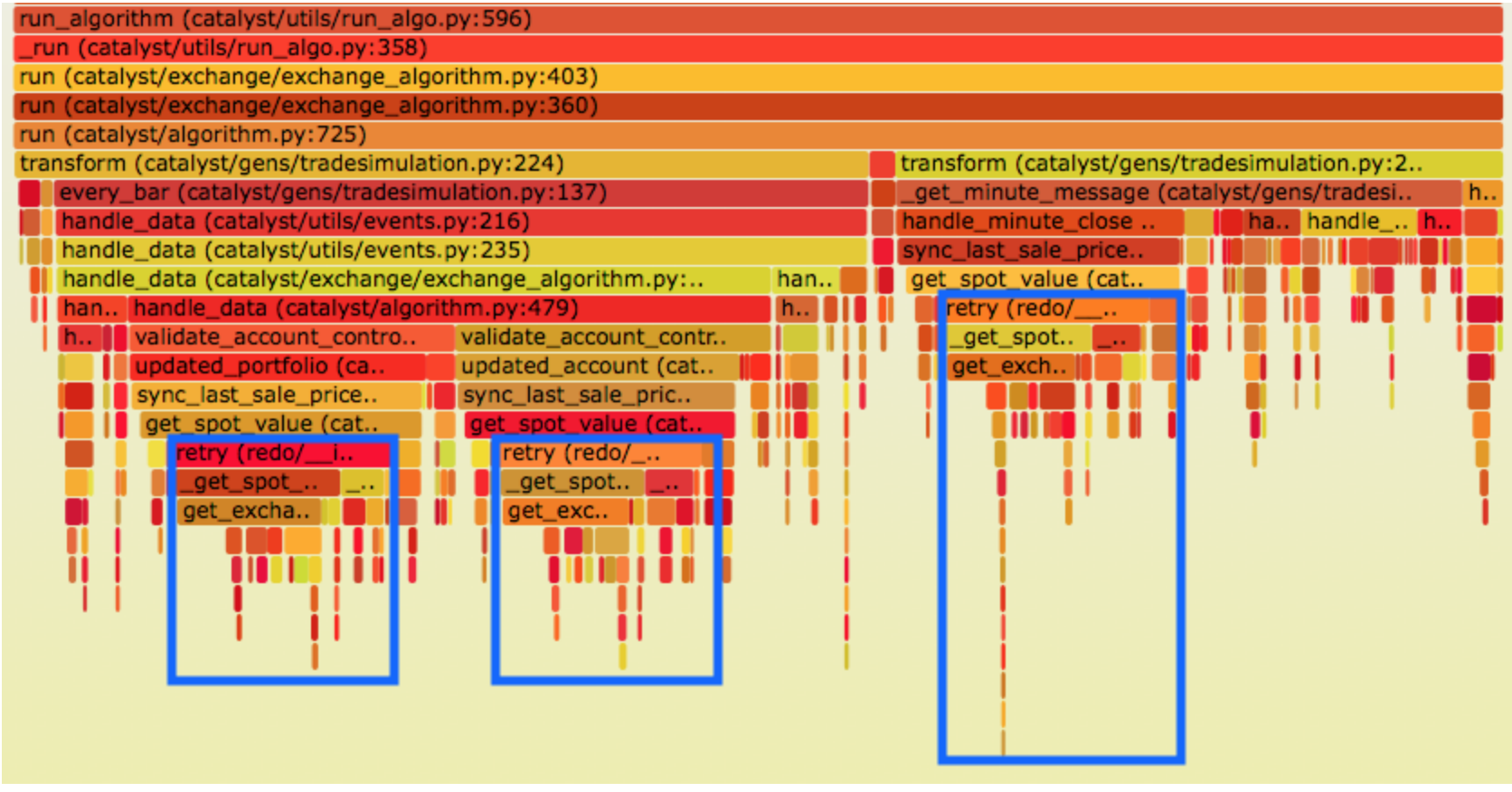
> Logs a message with level `DEBUG` on this logger. The `msg` is the message format string, and the `args` are the arguments which are merged into msg using the string formatting operator.

Solution: Lazy formatting in the `logging` standard library

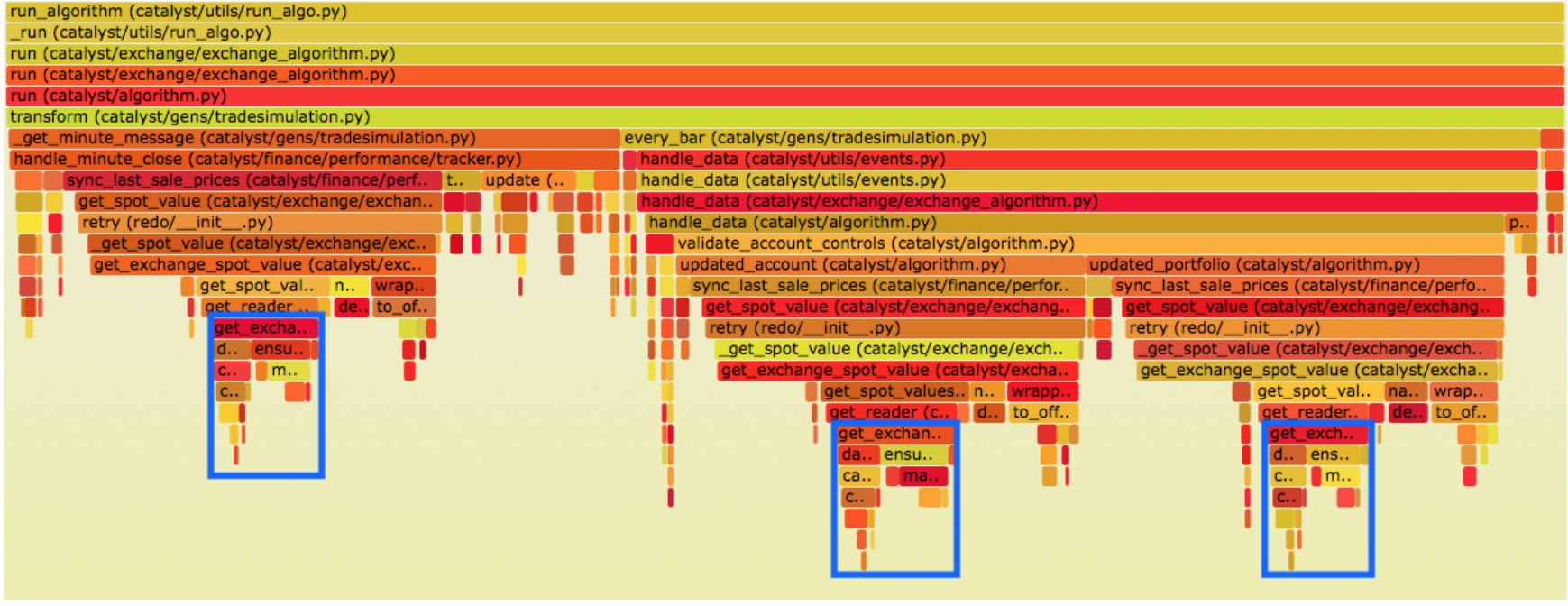
```
     logging.debug(
-        "calling %s with args: %s, kwargs: %s" % \
-        (action.__name__, args, kwargs)
+        "calling %s with args: %s, kwargs: %s", \
+        action.__name__, args, kwargs,
     )
```

Saved ~10% of execution time!

Result of running `py-spy` after that fix:



What next?:


run_algorithm (catalyst/utils/run_algo.py)
_run (catalyst/utils/run_algo.py)
run (catalyst/exchange/exchange_algorithm.py)
run (catalyst/exchange/exchange_algorithm.py)
run (catalyst/algorithm.py)
transform (catalyst/gens/tradesimulation.py)
_get_minute_message (catalyst/gens/tradesimulation.py)
every_bar (catalyst/gens/tradesimulation.py)
handle_minute_close (catalyst/finance/performance/tracker.py)
handle_data (catalyst/utils/events.py)
sync_last_sale_prices (catalyst/finance/perf..
t..
update (..
handle_data (catalyst/utils/events.py)
get_spot_value (catalyst/exchange/exchan..
handle_data (catalyst/exchange/exchange_algorithm.py)
retry (redo/__init__.py)
handle_data (catalyst/algorithm.py)
p..
_get_spot_value (catalyst/exchange/exc..
validate_account_controls (catalyst/algorithm.py)
get_exchange_spot_value (catalyst/exc..
updated_account (catalyst/algorithm.py)
updated_portfolio (catalyst/algorithm.py)
get_spot_val..
n..
wrap..
sync_last_sale_prices (catalyst/finance/perfor..
sync_last_sale_prices (catalyst/finance/perfo..
get_reader ...
de..
to_of..
get_spot_value (catalyst/exchange/exchang..
get_spot_value (catalyst/exchange/exchang..
get_excha..
retry (redo/__init__.py)
retry (redo/__init__.py)
d..
ensu..
_get_spot_value (catalyst/exchange/exch..
_get_spot_value (catalyst/exchange/exch..
c..
m..
get_exchange_spot_value (catalyst/excha..
get_exchange_spot_value (catalyst/excha..
c..
get_spot_values..
n..
wrapp..
get_spot_val..
na..
wrap..
get_reader (c..
d..
to_off..
get_reader..
de..
to_of..
get_exchan..
get_exch..
da..
ensu..
d..
ens..
ca..
ma..
c..
m..
c..
c..

- `get_exchange_folder`, you say?
- `ensure_folder_exists`? Really?

`catalyst.get_exchange_folder()` (simplified)

```
def get_exchange_folder(exchange_name):
    exchange_folder = os.path.join(
        DATA_ROOT, 'exchanges', exchange_name
    )
    ensure_directory(exchange_folder)
    return exchange_folder
```

- Called every hour in the simulation!

Anti-pattern: Repeated file system reads

- File system reads are expensive!
- Real life equivalent: `ensure_keyboard()`

Solution: `lru_cache` in the `functools` standard library

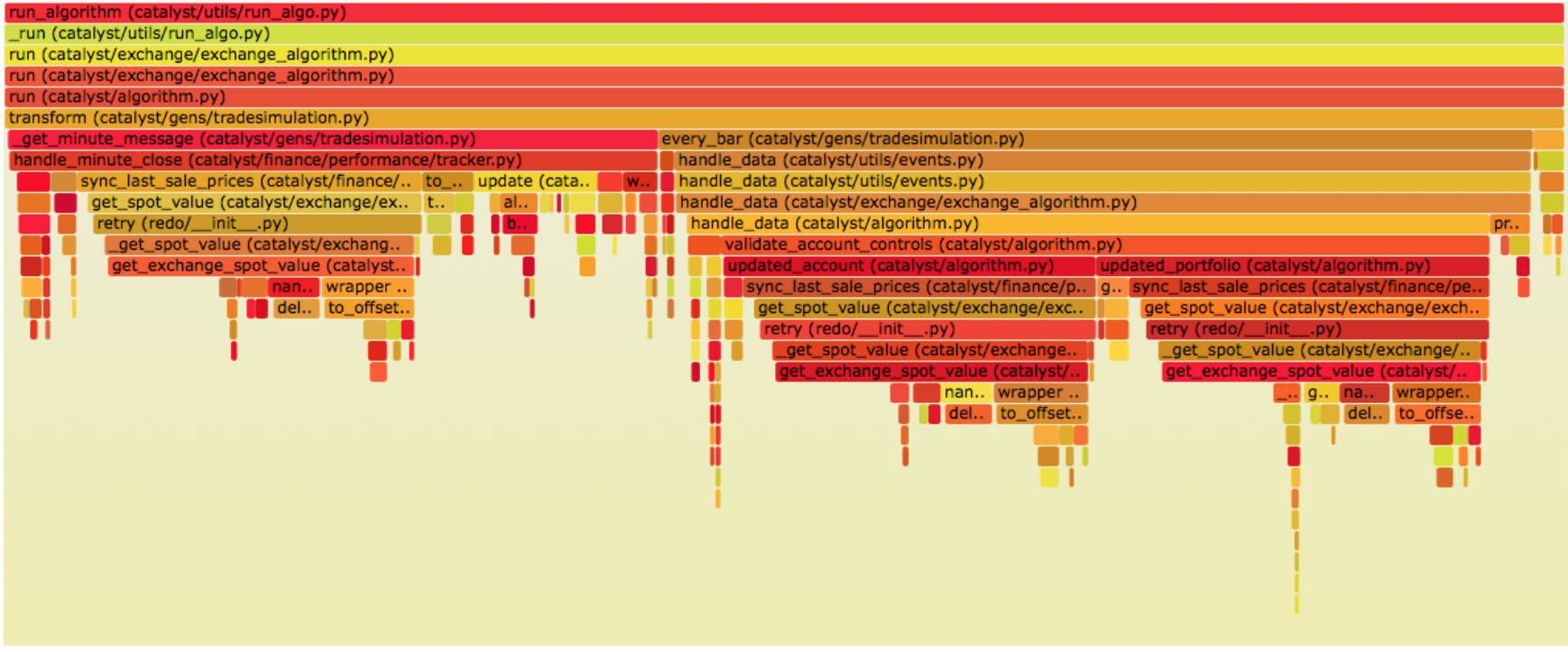
```
+from functools import lru_cache

+@lru_cache(maxsize=None)
 def get_exchange_folder(exchange_name):
     exchange_folder = os.path.join(
        DATA_ROOT, 'exchanges', exchange_name
     )
     ensure_directory(exchange_folder)
     return exchange_folder
```

Saved ~20% of execution time

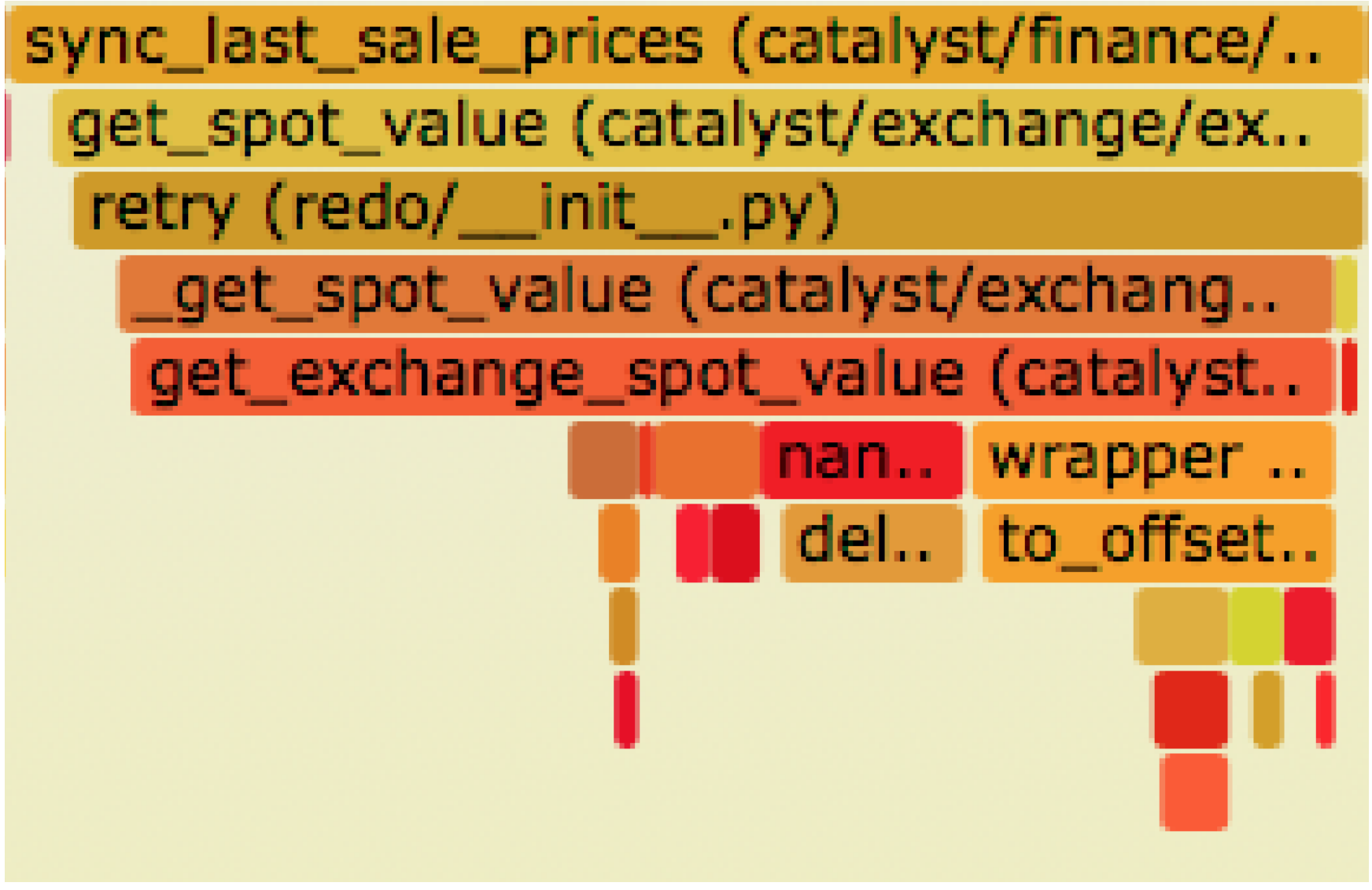
```
@functools.lru_cache(maxsize=128, typed=False)
```

> Decorator to wrap a function with a memoizing callable that saves up to the `maxsize` most recent calls. It can save time when an expensive or I/O bound function is periodically called with the same arguments.

Result of running `py-spy` after both fixes:



Zoom, enhance:



- `retry` : Almost no overhead
- No more expensive folder check!

Conclusion: key takeaways

- Profilers are not scary
- Flame graphs are lit 🔥
- 30-35% improvements with two simple fixes!
- Led to improvements in catalyst and redo
- Read the py-spy docs, there is much more you can do with it, we barely scratched the surface!

Q & A

Links

- Flame Graphs
- Python Standarb Library: `logging`
- Python Standarb Library: `functools.lru_cache`
- benfred/py-spy
- enigmampc/catalyst

Issues/PRs referenced:

- mozilla-releng/redo#51
- enigmampc/catalyst#500

Slides:

- samueldg/talks