



**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?

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

How do you read a flame graph?

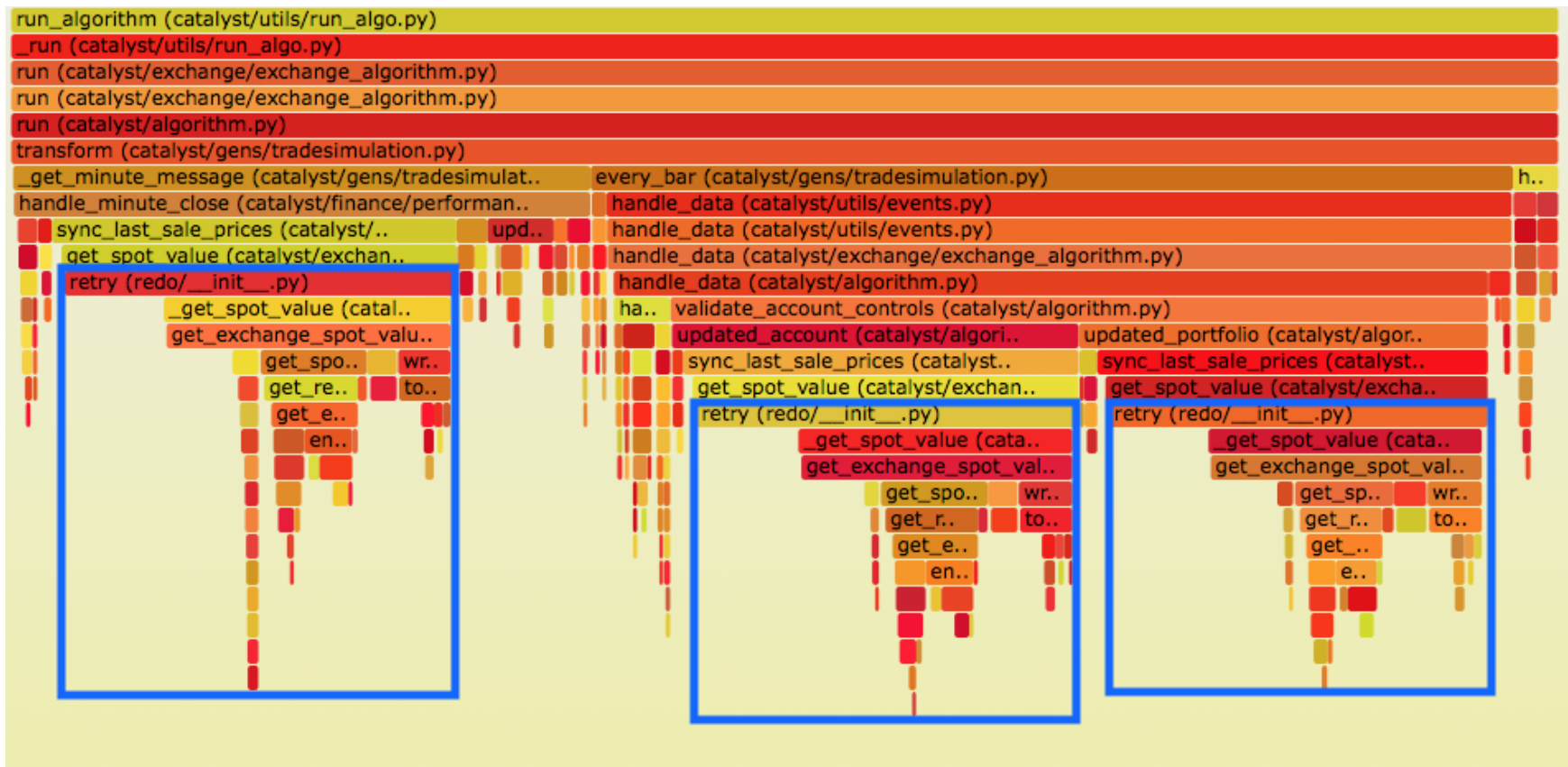
What to look for:

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

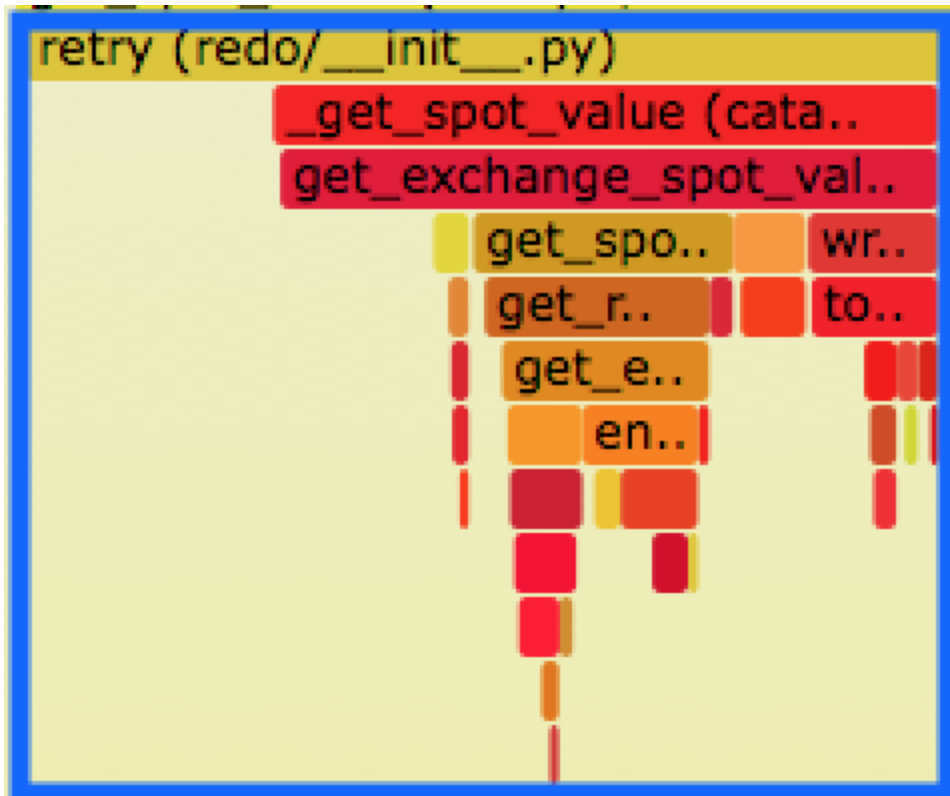
Performance anti-patterns + solutions

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)

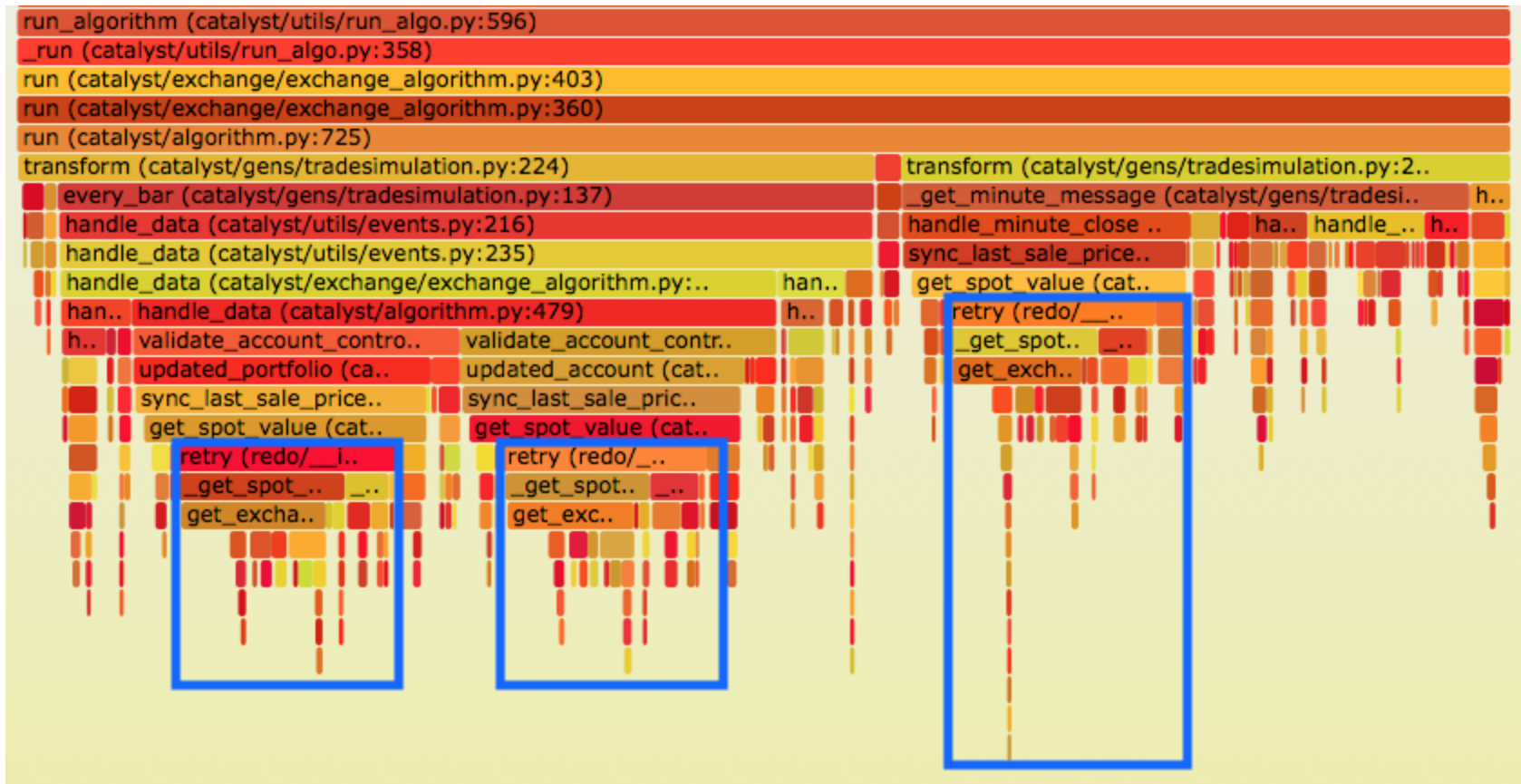
Logs a message with level `DEBUG` on this logger. The `msg` is the message format string, and the `args` are the arguments which are merged into `msg` using the string formatting operator.

Solution: Lazy formatting in the `logging` standard library

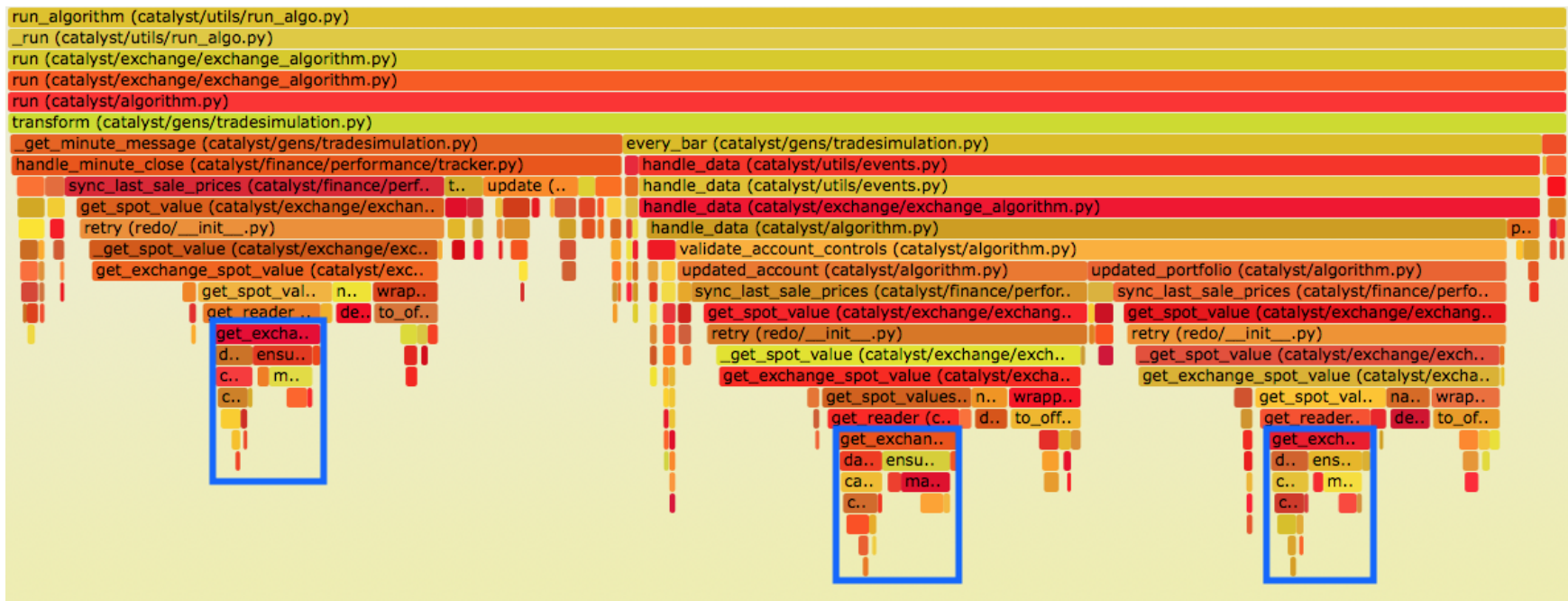
```
logging.debug(  
-     "calling %s with args: %s, kwargs: %s" % \  
-     (action.__name__, args, kwargs)  
+     "calling %s with args: %s, kwargs: %s", \  
+     action.__name__, args, kwargs,  
)
```

Saved ~10% of execution time!

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



What next?:



- `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

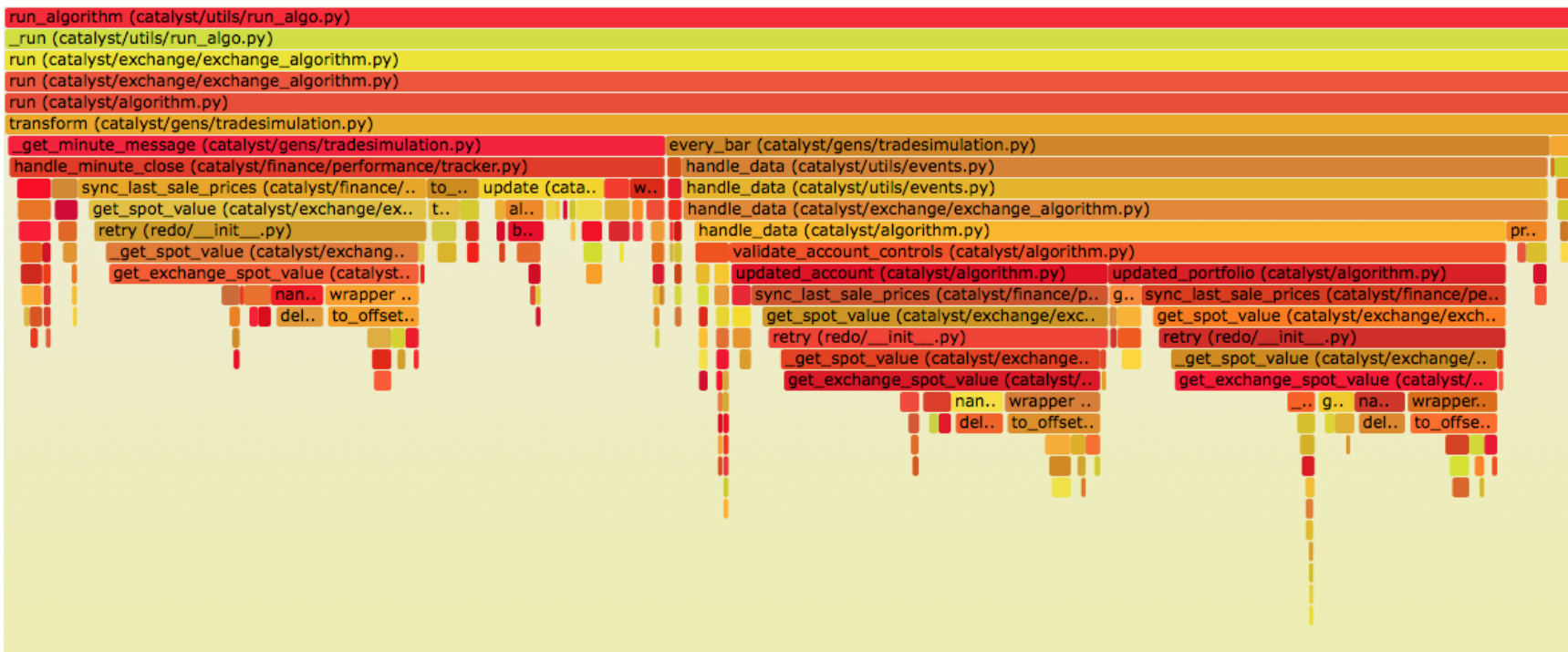
+@lru_cache(maxsize=None)
def get_exchange_folder(exchange_name):
    exchange_folder = os.path.join(
        DATA_ROOT, 'exchanges', exchange_name
    )
    ensure_directory(exchange_folder)
    return exchange_folder
```

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:

sync_last_sale_prices (catalyst/finance/..

```
get_spot_value (catalyst/exchange/ex..
```

```
retry (redo/___init__.py)
```

```
_get_spot_value (catalyst/exchang..
```

```
get_exchange_spot_value (catalyst..
```

nan.. wrapper ..

del.. to_offset..

- `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 Standard Library: logging](#)
- [Python Standard Library: functools.lru_cache](#)
- [benfred/py-spy](#)
- [enigmampc/catalyst](#)

Issues/PRs referenced:

- [mozilla-releng/redo#51](#)
- [enigmampc/catalyst#500](#)

Slides:

- [samueldg/talks](#)