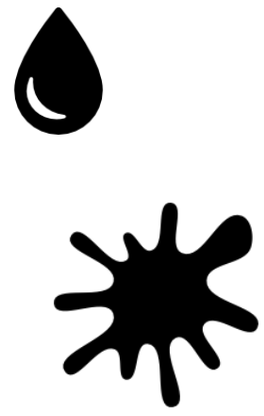
MEMORY LEAKS



Max Morlocke
Head of Engineering @ Mineralsoft

WHAT IS A MEMORY LEAK?

HOW DO I LEAK MEMORY IN PYTHON?

- Bad C Code
- Mutable default function parameters
- Storing data in a global or class scope
- Unhandled traceback keeping the stack frame alive
- Cyclic references

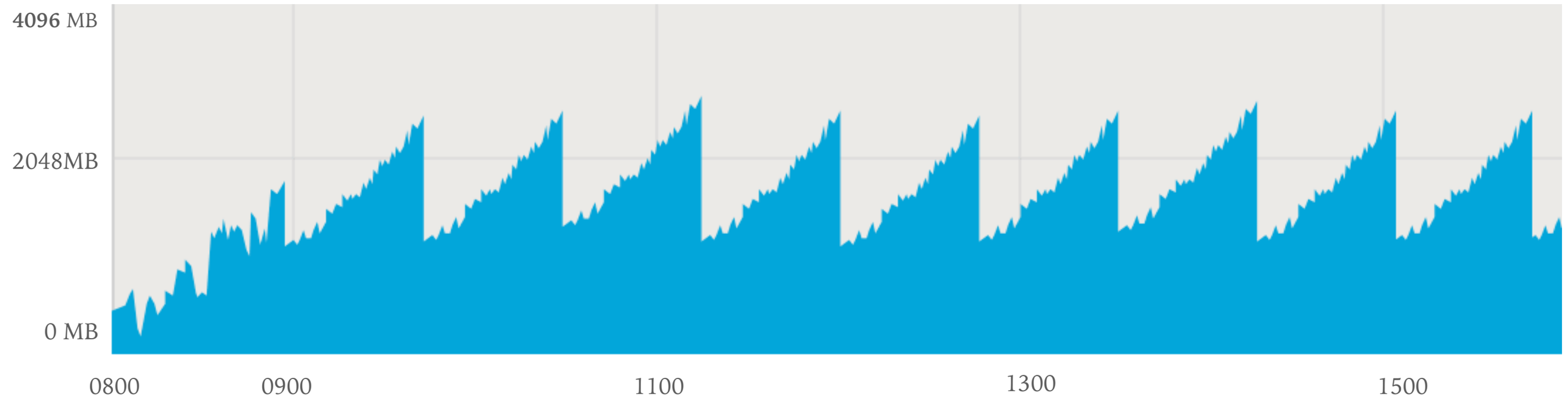
There are many more ways to generate a memory leak.

EXAMPLE

```
In [1]: def append_to(element, target=[]):
   ...:     target.append(element)
   ...:     return target
   ...:

In [2]: list = append_to(1)

In [3]: print(list)
[1]

In [4]: list = append_to(2)

In [5]: print(list)
[1, 2]
```

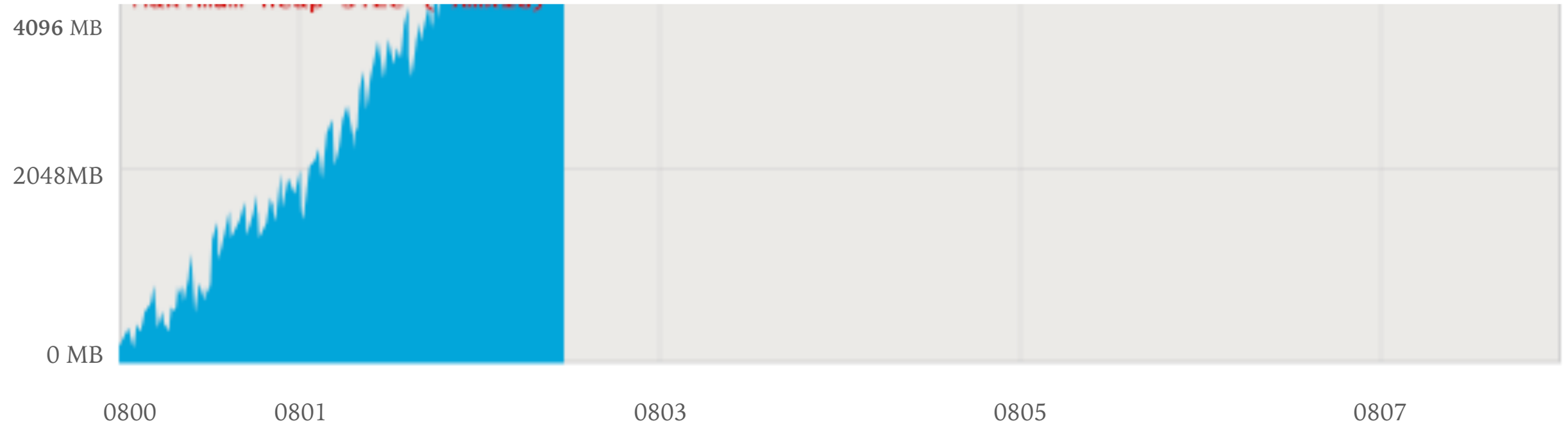
You may expect this to produce a new list each time the function is called. Instead, Python generates a new list ***once*** when the function is first initialized and maintains this list for each subsequent call. Thus, each time you call this function, the list grows.

MEMORY GRAPH 1: SAWTOOTH



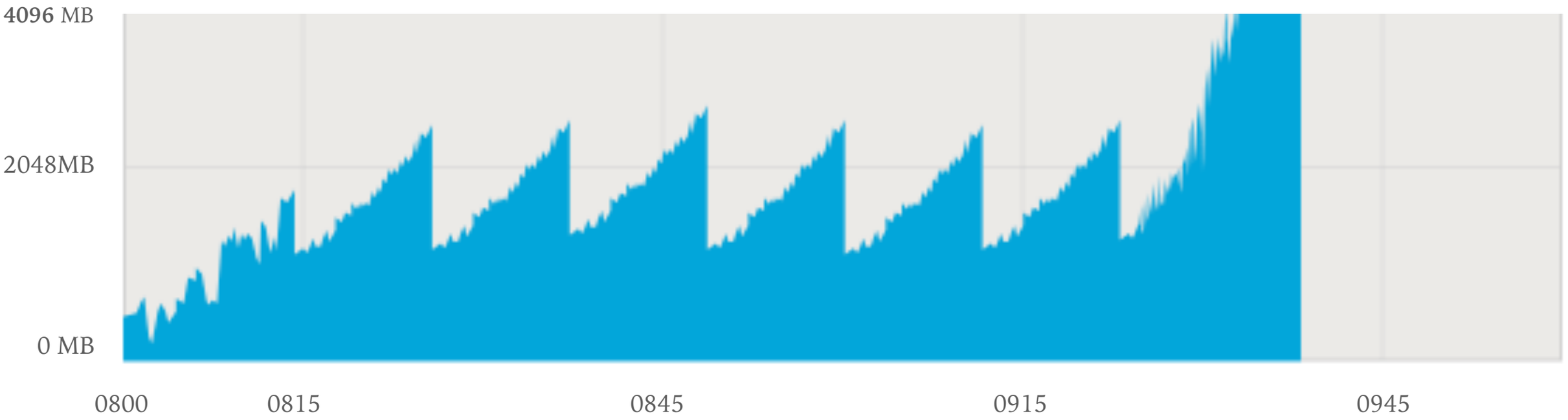
This is a fairly routine graph of memory usage for a single process (e.g. one gunicorn worker) of a web application under load. Note the Regular peaks and valleys in a sawtooth pattern. This occurs when longer lived objects (e.g. long running requests) consume memory for longer periods of time, shorter lived objects are collected more quickly. The most critical feature to note is that there is no growth trend in the long run, indicating that all objects in fact can and will be garbage collected, allowing us to claim that the application is leak-free.

MEMORY GRAPH 2: EARLY EXPLOSION



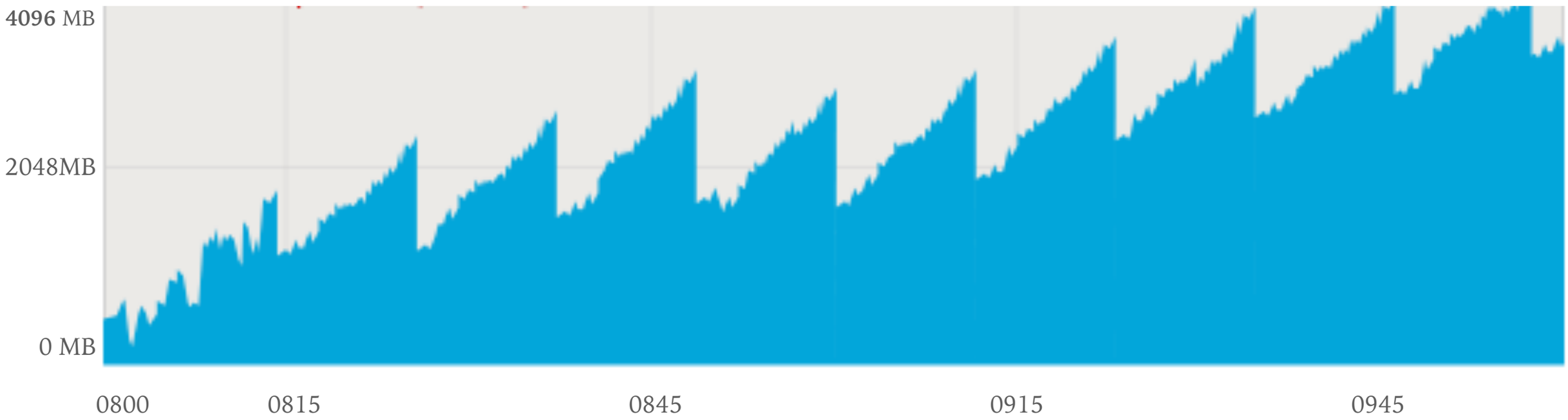
When memory quickly explodes at startup until a crash, this is probably not a memory leak. Instead, this is generally a symptom of either insufficient resources available *or* poorly optimized code keeping too many objects in memory (e.g. GranularLog.objects.all()).

MEMORY GRAPH 3: SURGE EXPLOSION



Similarly, when memory explodes after normal usage, this is probably not a memory leak. Instead, this is generally a symptom of either insufficient resources available *or* poorly optimized code keeping too many objects in memory (e.g. GranularLog.objects.all()). This is generally easier to correlate to a specific action with good logging and a few crashes.

MEMORY GRAPH 4: THE MEMORY LEAK



Here, while we see the expected sawtooth from garbage collection, there is a clear trend line of more and more memory being consumed over time. Typically, this does not correlate to requests. This is a clear case of a memory leak.

THE MEMORY DUMP

On Windows, MacOS, and Linux, you can get a memory dump using gdb. Even if your neckbeard makes Pat Rothfuss jealous, you probably still won't be able to produce anything useful from this. Why?

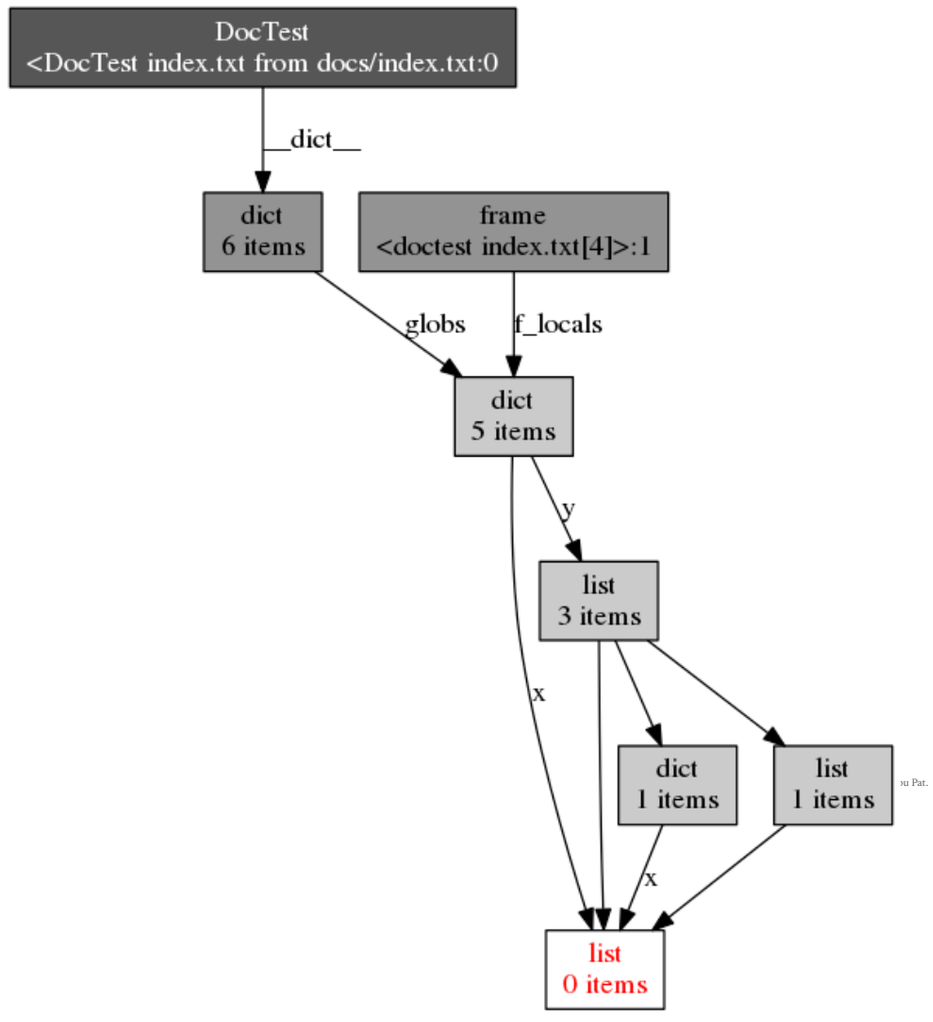
- GDB requires the source code and the environment used to run it. This makes it very difficult to troubleshoot cross-platform (e.g. linux server, mac dev environment).
- GDB graphical tools are negligible.
- No analyzer or hints exist.

I love you Pat.

INTRODUCING OBJGRAPH

objgraph (https://mg.pov.lt/objgraph/) is a module that lets you visually explore Python object graphs. You can:

- Show the most common types of objects.
- Examine the growth of objects between calls.
- Examine 'leaking' objects that have no objects pointing to them (you **must** filter out gc objects).
- Generate relationship graphs of any object tree using graphviz.



HOW DO I USE THIS?

Connecting via a remote debugger to production has some nasty security implications. You really shouldn't be able to do this… How else can this be done?

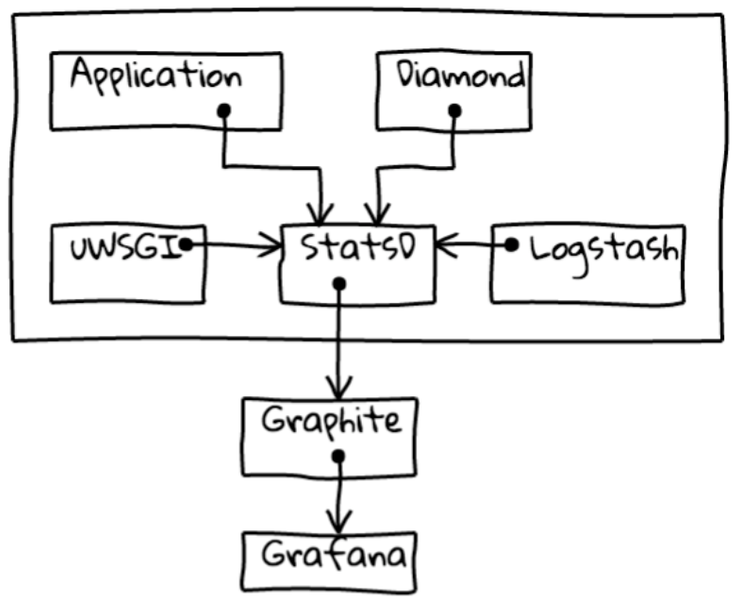
1. Log the most common objects in memory.
2. Do the unit of work.
3. Log the delta of objects in memory.

```
In [1]: import objgraph

In [2]: st = Statement.objects.filter(state='PENDING').last()

In [3]: objgraph.show_most_common_types()
function                   32778
dict                       23427
tuple                      18208
list                       10363
weakref                    7078
cell                       5736
type                       4777
getset_descriptor          3973
set                        2511
builtin_function_or_method 1915

In [4]: st.process()
2017-03-23 21:22:33,587 INFO  revenue.models [36038 140735203393536] [models.py-1280] Beginning processing of statement aeac5aab66

2017-03-23 21:22:38,657 INFO  revenue.models [36038 140735203393536] [models.py-1175] Setting state of statement aeac5aab66 to completed with status set to success
2017-03-23 21:22:38,657 DEBUG  revenue.models [36038 140735203393536] [models.py-1077] start creating accounting record for statement 955
2017-03-23 21:22:38,800 DEBUG  revenue.models [36038 140735203393536] [models.py-1092] finished creating accounting record 2872 for statement 955

In [5]: objgraph.show_growth()
function             33961    +33961
dict                 24902    +24902
tuple                18967    +18967
list                 10979    +10979
weakref               7491     +7491
cell                  5776     +5776
type                  5112     +5112
getset_descriptor     3996     +3996
set                   2515     +2515
ModuleSpec            2058     +2058
```

Well, those numbers don't look too ridiculous for processing a single PDF document that had 200 pages of data, normalizing it, auditing the arithmetic on the statement, and persisting all the data into the database.

WHAT NEXT?

Look at data at scale. Grab the objgraph output, parse it, and output it into statsd so it can be shipped to Graphite.



Send these hooks to your production environment and collect the data -OR- replay activities in a production-like environment. While a carefully limited objgraph calculation can be completed in less than a hundred ms on large memory allocations, this may break latency sensitive applications. Pick what suits you best!

WHAT MIGHT A REAL LEAK LOOK LIKE?

When developing a set of PDF parsers using a python wrapper for C code, I wrote some basic init functionality…

```python
class PDF2Json:
    '''
    Helper class for converting PDF's into a standard set of dicts
    '''

    def __init__(self, searchpath=None):
        license_key = self._get_license_key()

        self.tet = TET()
        self.tet.set_option('license={%s}' % license_key)
        self.tet.set_option('searchpath={./parsers/resources/pdflib}')
        self.tmpdir = utils.gen_tempdir()
        self.CHARACTER_DETAIL = False
```

Later on, I called some basic open and close file functionality I **ass**umed would open and close a file, properly clean up the file handles, etc. See where I'm going here?

```python
def _open(self):
    '''
    Opens a PDF file and returns a handle
    @param pdf: FileField: A django FileField object containing a pdf
    @return: int: the integer code associated with the file handle
    '''
    self.pdf_file_field.file.seek(0)
    self.data = self.pdf_file_field.file.read()
    doc_handle = self.tet.open_document(self.pdf_file_field.name, optlist='tetml={} shrug')
    if doc_handle == -1:
        raise TETException('Unable to open document {}: {}'.format(self.pdf_file_field.file, self.tet.get_errmsg()))

    return doc_handle

def _close(self, handle):
    '''
    Closes the tet file handle
    @param handle:
    @return:
    '''
    self.tet.close_document(handle)
```

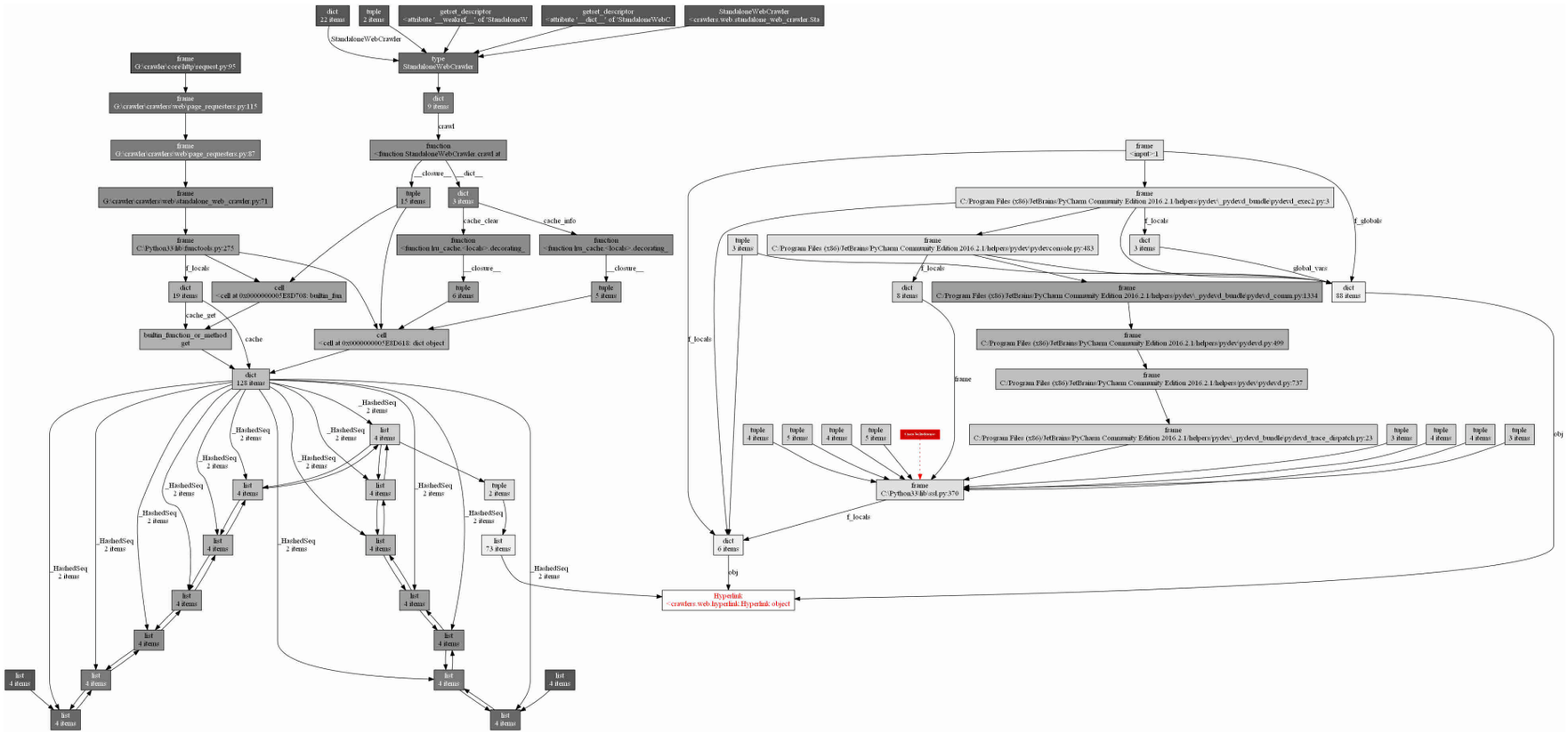
WHAT MIGHT A REAL LEAK LOOK LIKE?

I didn't have statsd, graphite, etc. setup at the time. So, I parsed through a whole lot of logged JSON dumps of objgraph output, pushed it into a relational database, and saw a lot of these objects from my celery workers:

- Dict: yay, dictionaries. I use these a *lot*.
- List: yay, lists. I use these a *lot*.
- lxml.element.Etree: We use pdflib for PDF processing. It produces XML. I need to parse that XML to generate a JSON data structure to store, as well as python objects to interact with. So, somewhere, I'm hanging onto things.
- ….. lots of other custom objects in low volumes

WHAT MIGHT A REAL LEAK LOOK LIKE?

The next thing I did was pull up an actual graph to see what the highest parent of an object I created was. It turned out to be the PDF2Json class.



This graph has literally nothing to do with my code and was pulled from Ben Bernard's blog (https://benbernardblog.com/tracking-down-a-freaky-python-memory-leak/). Alas, I do not have the objgraphs from my code showing the leak.

WHAT MIGHT A REAL LEAK LOOK LIKE?

I spent a lot of time looking at my code. Thankfully, I had a really big clue; I had recently upgraded to a major new revision of PDFLib. So, I had a hunch this was related. Unfortunately, I was on a new release where all my regressions had passed. Guess what I missed? There are create and delete virtual filesystem function that were created. Lo and behold, this manages memory space for documents that are passed in memory. If you don't use that, memory isn't cleaned up. No other functionality changes. Notice the use of __enter__ and __exit__ to guarantee the calling of the delete function(!!!!!):

```
class PDF2Json:
    '''
    Helper class for converting PDF's into a standard set of dicts
    '''

    def __enter__(self, searchpath=None):
        license_key = self._get_license_key()

        self.tet = TET()
        self.tet.set_option('license={%s}' % license_key)
        self.tet.set_option('searchpath={./parsers/resources/pdflib}')
        self.tmpdir = utils.gen_tempdir()
        self.CHARACTER_DETAIL = False
        return self

    def __exit__(self, exception_type, exception_value, traceback):
        if self.tet.info_pvf(self.pdf_file_field.name, 'exists') != 0:
            self.tet.delete_pvf(self.pdf_file_field.name)
        self.tet.delete()
```

OTHER THINGS THAT MAY BE HELPFUL

Taking thread dumps of activity may show tasks running and called dependencies that you may not expect. While GDB is a really hard tool to use for memory dumps, thread dumps are readable to most senior developers (https://wiki.python.org/moin/DebuggingWithGdb):

```
(gdb) thread apply all py-list
...
 200
 201        def accept(self):
>202            sock, addr = self._sock.accept()
 203            return _socketobject(_sock=sock), addr
 204        accept.__doc__ = _realsocket.accept.__doc__
 205
 206        def dup(self):
 207            """dup() -> socket object

Thread 35 (Thread 0xa0bfdb40 (LWP 17911)):
Unable to locate python frame

Thread 34 (Thread 0xa13feb40 (LWP 17910)):
 197            for method in _delegate_methods:
 198                setattr(self, method, dummy)
 199        close.__doc__ = _realsocket.close.__doc__
 200
 201        def accept(self):
>202            sock, addr = self._sock.accept()
 203            return _socketobject(_sock=sock), addr
...
```

BEST PRACTICES: C LIBRARIES

There is only so much you can control here, so do your best to control what you can.

- Use well known, regularly updated libraries and wrappers!
- Don’t depend solely on application logic regression tests, run sustained load tests when upgrading major libraries.
- Register for distribution lists for patch notifications so you know when a memory leak has already been fixed.
- Keep yourself up to date.

BEST PRACTICES: MUTABLE FUNCTION PARAMETERS

Don’t use mutable default arguments, set to None and test if None at the beginning of the function:

```
def append_to(element, to=None):
    if to is None:
        to = []
    to.append(element)
    return to
```

Thanks to the Hitchhiker’s Guide to Python’s Common Gotcha’s section: http://docs.python-guide.org/en/latest/writing/gotchas/

BEST PRACTICES: USE WEAK REFERENCES APPROPRIATELY

Weak references are a dangerous and powerful tool. When an object is referred to only by weak references, it can be destroyed. Classic examples of using weak references is in mappings referring to large objects or cache and observer pattern implementations.

http://stackoverflow.com/questions/2436302/when-to-use-weak-references-in-python

```python
import weakref

class Emitter(object):
    def __init__(self):
        self.listeners = weakref.WeakSet()

    def emit(self):
        for listener in self.listeners:
            # Notify
            listener('hello')


class Receiver(object):
    def __init__(self, emitter):
        # Create the bound method object
        cb = self.callback
        # Register it
        emitter.listeners.add(cb)
        # But also create an own strong reference to keep it alive
        self._callbacks = set([cb])

    def callback(self, msg):
        print('Message received: {}'.format(msg))


e = Emitter()
l = Receiver(e)
assert len(e.listeners) == 1

del l
import gc; gc.collect()
assert len(e.listeners) == 0
```

```python
class Dict(dict):
    pass

obj = Dict(red=1, green=2, blue=3)   # this object is weak referenceable
```

https://docs.python.org/3/library/weakref.html

BEST PRACTICES: USE DESTRUCTORS CAREFULLY

Cyclic references are typically collected by python's garbage collector. The most common case where this fails is when someone overrides the __del__ function. In almost all cases, you don't want to do this. Instead, use the __enter__ and __exit__ functions and call your class using 'with':

```
class PDF2Json:
    '''
    Helper class for converting PDF's into a standard set of dicts
    '''

    def __enter__(self, searchpath=None):
        license_key = self._get_license_key()

        self.tet = TET()
        self.tet.set_option('license={%s}' % license_key)
        self.tet.set_option('searchpath={./parsers/resources/pdflib}')
        self.tmpdir = utils.gen_tempdir()
        self.CHARACTER_DETAIL = False
        return self

    def __exit__(self, exception_type, exception_value, traceback):
        if self.tet.info_pvf(self.pdf_file_field.name, 'exists') != 0:
            self.tet.delete_pvf(self.pdf_file_field.name)
        self.tet.delete()
```

```
from parsers.pdf_to_json import PDF2Json

with PDF2Json() as converter:
    json_data = converter.get_data(cls.st.report_file)
```

```
with open('output.txt', 'w') as f:
    f.write('Hi there!')
```