No time to idle about: Profiling import time in Python

Daniel Porteous

Production Engineer

dport.me/pycon.pdf

Me!

Me!!! Daniel Porteous!

Tweeter:

@banool1

Github:

github.com/banool

Website:

dport.me

Agenda

- How to acquire import time data.
- How to understand the data through visualisation.
- Suggestions for fixing the problem.

The problem

Imports are slow, and you don't know why

- Why are slow imports a problem?
  - Mainly, command line tools.
  - Worsens dev iteration time.
- How do we fix the problem?
  - First, understand it.

Looking at an example

Slooooooooooow imports zzz zzz zzz

main.py

```
1 import small
2 import large
3 print("Hi Pycon AU 2019!")
```

small.py

```
1 import time
2 time.sleep(1)
```

large.py

```
1 import time
2 time.sleep(10)
```

```
$ time python main.py
Hi Pycon AU 2019!

real 0m11.061s
```

Per module import times

A few different approaches

- Time each import directly in your own code.
  - Explicit time statements, context managers.
- Modify module Finders / Loaders.
- Probes???

Per module import times

A few different approaches

- `-X importtime` in 3.7.
  - Also the `PYTHONPROFILEIMPORTTIME` environment variable.
- Supported directly in `Python/import.c`.
- Thanks to the community and 3.7 contributors!
  - Special thanks to Victor Stinner and Inada Naoki!!!

Back to our example

Still slow, who's the culprit!

main.py

```python
import small
import large
print("Hi!")
```

small.py

```python
import time
time.sleep(1)
```

large.py

```python
import time
time.sleep(10)
```

```
$ python -X importtime main.py
import time: self [us] | cumulative | imported package
import time:       145 |        145 | zipimport
import time:       737 |        737 | _frozen_importlib_external
import time:        73 |         73 |     _codecs
import time:      6401 |       6474 |   codecs
$ python -X importtime main.py
import time:       512 |        512 | encodings.utf_8
import time:       115 |        115 | _signal
import time:       576 |        576 | encodings.latin_1
import time:        51 |         51 |     abc
$ PYTHONPROFILEIMPORTTIME="🤠"
  python main.py
import time:         7 |          7 |       _stat
import time:         0 |          0 |     stat
import time:      1073 |       1073 |       genericpath
import time:      2533 |       3605 |     posixpath
import time:      4793 |       4793 |     _collections_abc
import time:      4456 |      14072 |   os
import time:      1277 |       1277 |   __builtin__
import time:       190 |        190 |     _locale
import time:       639 |        829 |   _bootlocale
import time:      1761 |       1761 |   sitecustomize
import time:      4901 |      22838 | site
import time:       719 |        719 |   time
import time:   1003709 |    1004428 | small
import time:  10003076 |   10003076 | large
Hi Pycon AU 2019!
```

Individual import times

Knowledge is power 🤔⏱🐍

```
$ python -X importtime main.py
import time: self [us] | cumulative | imported package
import time:       719 |        719 |   time
import time:   1003709 |    1004428 | small
import time:  10003076 |   10003076 | large
Hi Pycon AU 2019!
```

dport.me/pycon.pdf

```
$ python -X importtime main.py
import time: self [us] | cumulative | imported package
import time:       145 |        145 | zipimport
import time:       737 |        737 | _frozen_importlib_external
import time:        73 |         73 |     _codecs
import time:      6401 |       6474 |   codecs
import time:      2292 |       2292 |   encodings.aliases
import time:      1780 |      10545 | encodings
import time:       512 |        512 | encodings.utf_8
import time:       115 |        115 | _signal
import time:       576 |        576 | encodings.latin_1
import time:        51 |         51 |     _abc
import time:      1233 |       1283 |   abc
import time:      1084 |       2367 | io
import time:        77 |         77 |       _stat
import time:      1144 |       1220 |     stat
import time:      1073 |       1073 |       genericpath
import time:      2533 |       3605 |     posixpath
import time:      4793 |       4793 |     _collections_abc
import time:      4456 |      14072 |   os
import time:      1277 |       1277 |   __builtin__
import time:       190 |        190 |     _locale
import time:       639 |        829 |   _bootlocale
import time:      1761 |       1761 |   sitecustomize
import time:      4901 |      22838 | site
import time:       719 |        719 |   time
import time:   1003709 |    1004428 | small
import time:  10003076 |   10003076 | large
Hi Pycon AU 2019!
```

Visualising the data

Discussion TODO

- Default output is great! 3.7 changed the game here.
  - The output still requires study.
- Visualisation is powerful because you can identify problems immediately.
- Flamegraphs are common visualisations of stack data.

Visualising the data

Structuring the data

```
$ python -X importtime main.py 2> main.stderr
Hi Pycon AU 2019!

$ cat main.stderr | python tree.py --basic > main.tree
everything 38523
 small 454  main.tree
  time 528
 large 1496528
large 1496

$ flamegraph.pl main.tree --flamechart tree > main.svg
```

Visualising the data

🐟 🐟 🐟

- Tuna: https://github.com/nschloe/tuna
  - Big thanks to Nico Schloemer!

```
$ python -X importtime main.py &> main.stderr

$ tuna main.stderr
# Starts up a Python webserver
```

Visualising the data

Is a tree 🌳 the right data structure?

The import structure isn't strictly a tree, it's a graph

```
1 sys.modules = []
2
3 def import(module):
4     for m in module.imports:
5         if m in sys.modules:
6             continue
7         import(m)
8         sys.modules.append(m)
```

Visualising the data

Demonstrating import behaviour

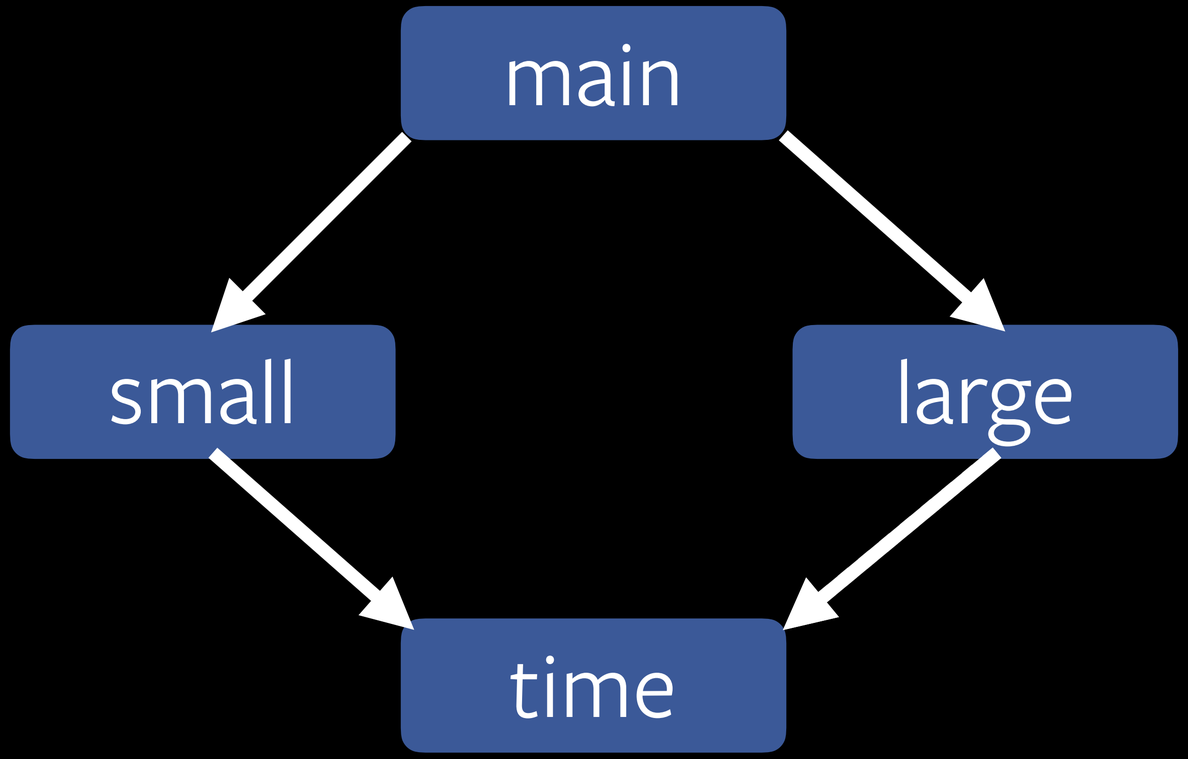
main.py

```
1 import small
2 import large
3 print("Hi!")
```

```
self   | cumulative | package
   528 |        528 |   time
   454 |        981 | small
  1496 |       1496 | large
```

large.py

```
1 import time
2 time.sleep(10)
```



Visualising the data

Showing imports as a graph

- Pydeps: https://github.com/thebjorn/pydeps
  - Open source project that produces import graphs
  - Big thanks to Bjørn Pettersen!

```
$ pydeps main.py --max-bacon 0 -o main.svg
```

Visualising the data

We've tweaked the numbers

main.py

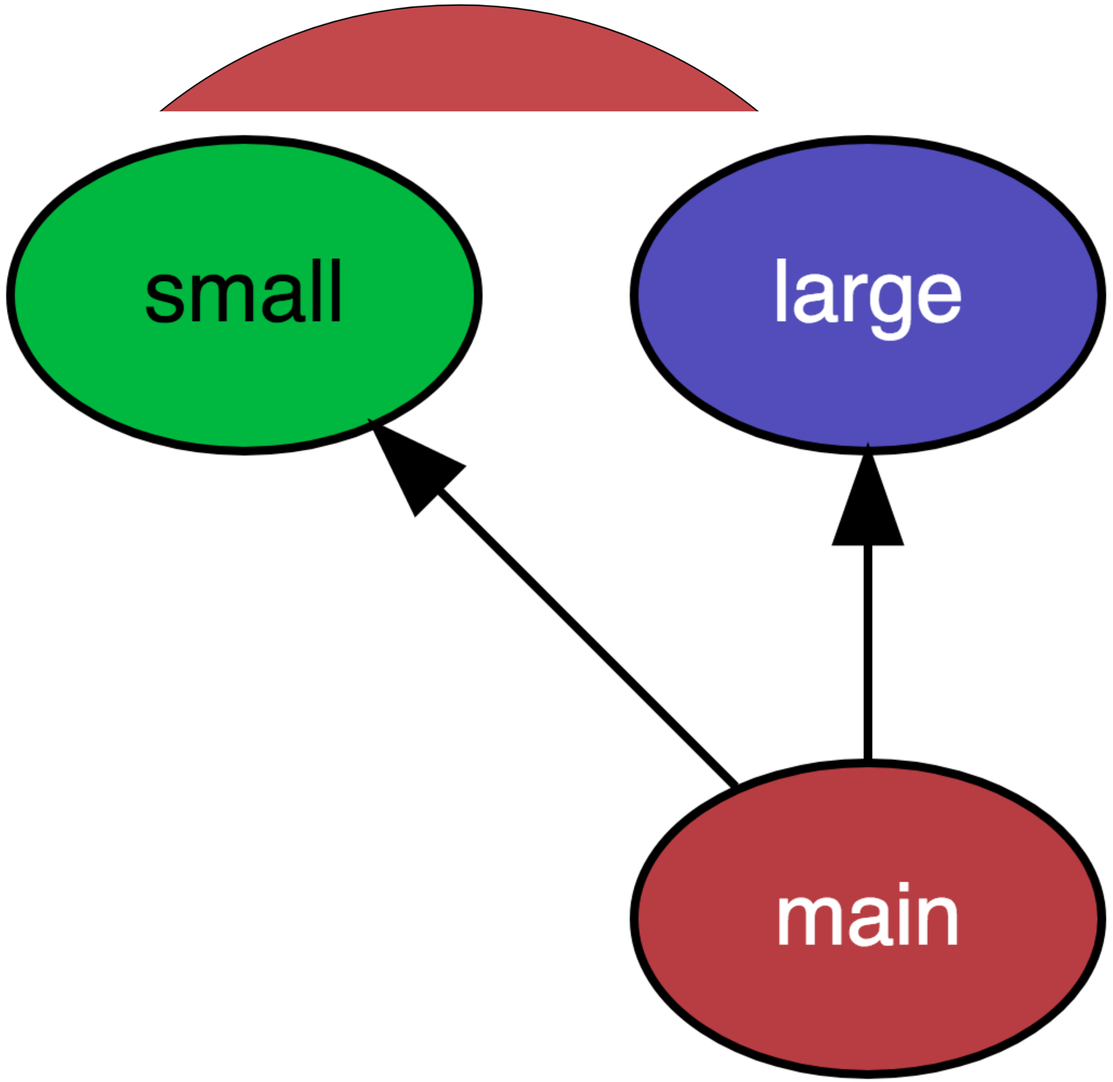
```
1 import small
2 import large
3 print("Hi!")
```

small.py

```
1 import time
2 time.sleep(1)
```

large.py

```
1 import time
2 time.sleep(10)
```



Connectedness metrics???

What can we do with this information?

Tuna: https://github.com/nschloe/tuna

- Big thanks to Nico Schloemer!

```
$ python -X importtime main.py &> main.stderr

$ tuna main.stderr
# Starts up a Python webserver
```

Propose node cost with just import time and connectedness

What can we do with this information?

Tuna: https://github.com/nschloe/tuna

- Big thanks to Nico Schloemer!

```
$ python -X importtime main.py &> main.stderr

$ tuna main.stderr
# Starts up a Python webserver
```

Get unique attribute accesses

What can we do with this information?

Tuna: https://github.com/nschloe/tuna

- Big thanks to Nico Schloemer!

```
$ python -X importtime main.py &> main.stderr

$ tuna main.stderr
# Starts up a Python webserver
```

Suggest node cost with import time, connectedness, and unique attribute access all together

What can we do with this information?

<import time>

<num connections> * <unique attributes accessed>

```
$ python -X importtime main.py &> main.stderr

$ tuna main.stderr
# Starts up a Python webserver
```

Improving import time

Finally!

- BLAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAHHH!!!!

BLAH

BLAAAAHHHHHHH

- BLAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAHHH!!!!

BLAH

BLAAAAHHHHHHH

- BLAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAHHH!!!!

Context managers in core Python

contextlib.suppress

```
1 def kill_process(pid):
2     try:
3         os.kill(pid, signal.SIGKILL)
4     except ProcessLookupError:
5         pass
```

Context managers in core Python

contextlib.suppress

```
1 from contextlib import suppress
```

Context managers in core Python

contextlib.suppress

```
1 from contextlib import suppress
2
3 def kill_process(pid):
4     with suppress(ProcessLookupError):
5         os.kill(pid, signal.SIGKILL)
```

Context managers in core Python

contextlib.suppress

```
1 from contextlib import suppress
2
3 def kill_process(pid):
4     with suppress(ProcessLookupError):
5         os.kill(pid, signal.SIGKILL)
```

```
1 def kill_process(pid):
2     try:
3         os.kill(pid, signal.SIGKILL)
4     except ProcessLookupError:
5         pass
```

Context managers in core Python

ThreadPoolExecutor - Bad version!

```
1 # Bad!!!
2 pool = ThreadPoolExecutor()
3 for k, v in data.items():
4     pool.submit(myfunc, k, v)
5 # Wait on the results and do something with them.
6 pool.shutdown()
```

Context managers in core Python

ThreadPoolExecutor - Good version!

```
1 # Bad!!!
2 pool = ThreadPoolExecutor()
3 for k, v in data.items():
4     pool.submit(myfunc, k, v)
5 # Wait on the results and do something with them.
6 pool.shutdown()
```

```
1 # Good, safe, context managed!
2 with ThreadPoolExecutor() as pool:
3     for k, v in data.items():
4         pool.submit(myfunc, k, v)
```

Context managers in core Python

ThreadPoolExecutor - Good version!

```
1 data = {
2     "Watermelon": "delicious",
3     "Fruit": "spectacular",
4     "Dairy": "scary",
5     "Chicken": "not cool",
6 }
```

Context managers in core Python

ThreadPoolExecutor - Good version!

```
1 data = {
2     "Watermelon": "delicious",
3     "Fruit": "spectacular",
4     "Dairy": "scary",
5     "Chicken": "not cool",
6 }
7
8 def myfunc(noun, adj):
9     return f"{noun} is {adj}!"
```

Context managers in core Python

ThreadPoolExecutor - Good version!

```
1 data = {
2     "Watermelon": "delicious",
3     "Fruit": "spectacular",
4     "Dairy": "scary",
5     "Chicken": "not cool",
6 }
7
8 def myfunc(noun, adj):
9     return f"{noun} is {adj}!"
```

```
$ python3 example.py
Meat is not cool!
Dairy is yucky!
Fruit is spectacular!
Watermelon is delicious!
```

Write your own context managers!

Our very own context manager!

Simple!

```
 1 class MyContextManager:
 2     def __enter__(self):
 3         print("Enter!")
 4
 5     def __exit__(self, *exc):
 6         print("Exit!")
 7
 8
 9 with MyContextManager():
10     print("Inside the block!")
```

Our very own context manager!

Simple!

```
 1 class MyContextManager:
 2     def __enter__(self):
 3         print("Enter!")
 4
 5     def __exit__(self, *exc
 6         print("Exit!")
 7
 8
 9 with MyContextManager():
10     print("Inside the block!")
```

Our very own context manager!

Super simple!

```
 1 class MyContextManager:
 2     def __enter__(self):
 3         print("Enter!")
 4
 5     def __exit__(self, *exc):
 6         print("Exit!")
 7
 8
 9 with MyContextManager():
10     print("Inside the block!")
```

```
$ python3 example.py
Enter!
Inside the block!
Exit!
```

As neat as it gets!

```
 1 class FoodContextManager:
 2     def __init__(self):
 3         self.data = {}
 4
 5     def __enter__(self):
 6         print(f"Enter: {self.data}")
 7         return self.data
 8
 9     def __exit__(self, *exc):
10         print(f"Exit: {self.data}")
11
12 with FoodContextManager() as data:
13     data["fruit"] = "delicious"
```

As neat as it gets!

```
 1 class FoodContextManager:
 2     def __init__(self):
 3         self.data = {}
 4
 5     def __enter__(self):
 6         print(f"Enter: {self.data}")
 7         return self.data
 8
 9     def __exit__(self, *exc):
10         print(f"Exit: {self.data}")
11
12 with FoodContextManager() as data:
13     data["fruit"] = "delicious"
```

A fully fledged context manager!

```
 1 class FoodContextManager:
 2     def __init__(self, data):
 3         self.data = data
 4
 5     def __enter__(self):
 6         print(f"Enter: {self.data}")
 7         return self.data
 8
 9     def __exit__(self, *exc):
10         print(f"Exit: {self.data}")
11
12 with FoodContextManager({"dairy":"yuck"}) as data:
13     data["fruit"] = "delicious"
```

A fully fledged context manager!

```
 1 class FoodContextManager:
 2     def __init__(self, data):
 3         self.data = data
 4
 5     def __enter__(self):
 6         print(f"Enter: {self.data}")
 7         return self.data
 8
 9     def __exit__(self, *exc):
10         print(f"Exit: {self.data}")
11
12 with FoodContextManager({"dairy": "yuck"}) as data:
13     data["fruit"] = "delicious"
```

dport.me/pycon.pdf

"Boy, that sure was a lot of work"

— Me when I first wrote a context manager this way

There is an easier way!

`contextlib.contextmanager`

`@contextlib.contextmanager`

> This function is a decorator that can be used to define a factory function for with statement context managers, without needing to create a class or separate `__enter__()` and `__exit__()` methods.

dport.me/pycon.pdf

Decorators

Decorators

🎁🎁🎁

```
1 def my_decorator(func):
2     def new_func():
3         return func() + "!!!!"
4     return new_func
```

Decorators

🎁🎁🎁

```
1 def my_decorator(func):
2     def new_func():
3         return func() + "!!!!"
4     return new_func
5
6
7 def hello_pycon():
8     return "Hello Pycon AU 2018!"
9
10
```

Decorators

🎁🎁🎁

```
1 def my_decorator(func):
2     def new_func():
3         return func() + "!!!!"
4     return new_func
5
6 @my_decorator
7 def hello_pycon():
8     return "Hello Pycon AU 2018!"
9
10
```

Decorators

```
1 def my_decorator(func):
2     def new_func():
3         return func() + "!!!!"
4     return new_func
5
6 @my_decorator
7 def hello_pycon():
8     return "Hello Pycon AU 2018!"
9
10 hello_pycon()
```

dport.me/pycon.pdf

Decorators

🎁🎁🎁

on AU 2018!!!!!

🐍 🐍 🐍 🐍 🐍

Decorators

```
1 @my_decorator
2 def hello_pycon():
3     return "Hello Pycon AU 2018!"
```

Decorators

```
1 @my_decorator
2 def hello_pycon():
3     return "Hello Pycon AU 2018!"
```

```
1 def hello_pycon():
2     return "Hello Pycon AU 2018!"
3 hello_pycon = my_decorator(hello_pycon)
```

dport.me/pycon.pdf

Generators

Generating nothing but good times 🎉

Generators

Generating nothing but good times 🎉

```
1 def first_n(n):
2     for i in range(n):
3         yield i
```

Generators

Generating nothing but good times 🎉

```
1 def first_n(n):
2     for i in range(n):
3         yield i
4
5 gen = first_n(5)
6 print(next(gen))
7 print(next(gen))
```

Generators

Generating nothing but good times 🎉

```
1 def first_n(n):
2     for i in range(n):
3         yield i
4
5 gen = first_n(5)
6 print(next(gen))  # 0
7 print(next(gen))
```

Generators

Generating nothing but good times 🎉

```
1 def first_n(n):
2     for i in range(n):
3         yield i
4
5 gen = first_n(5)
6 print(next(gen))  # 0
7 print(next(gen))  # 1
```

Generators

Generating nothing but good times 🎉

```
1 def first_n(n):
2     for i in range(n):
3         yield i
4
5 for i in first_n(5):
6     print(i)
```

Generators

Generating nothing but good times 🎉

```
1 def first_n(n):
2     for i in range(n):
3         yield i
```

Generators

Generating nothing but good times 🎉

```
1 def first_n(n):
2     for i in range(n):
3         yield i
```

```
1 def first_n(n):
2     nums = []
3     for i in range(n):
4         nums.append(i)
5     return nums
```

Generators

Generating nothing but good times 🎉

```
1 def first_n(n):
2     for i in range(n):
3         yield i
```

```
1 def first_n(n):
2     nums = []
3     for i in range(n):
4         nums.append(i)
5     return nums
```

```
>>> print(sum(first_n(10)))
```

Generators

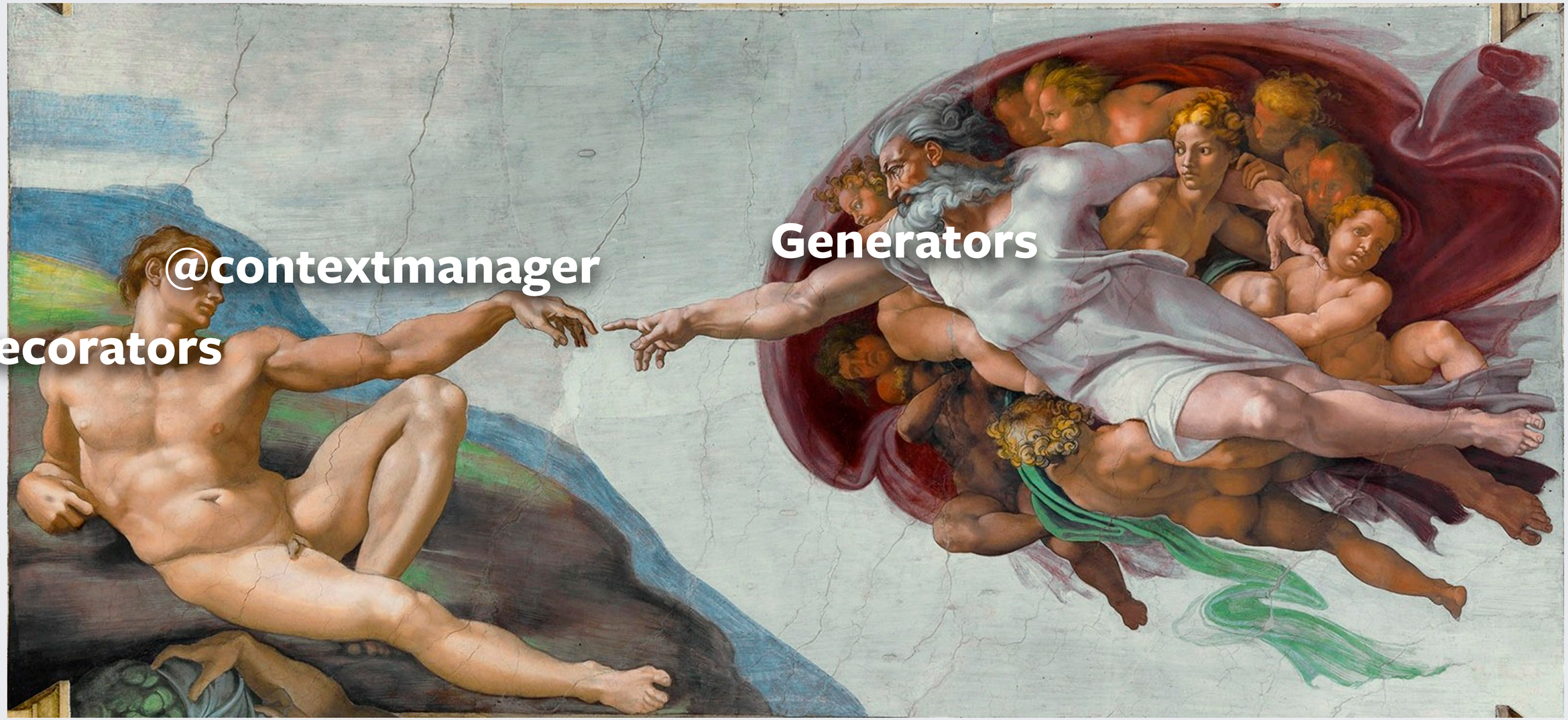
Generating nothing but good times 🎉

```
1 def first_n(n):
2     for i in range(n):
3         yield i
```

```
1 def first_n(n):
2     nums = []
3     for i in range(n):
4         nums.append(i)
5     return nums
```

```
>>> print(sum(first_n(10)))
45
```

Let's put them together



There is an easier way!

contextlib.contextmanager

```
1  class MyContextManager:
2      def __enter__(self):
3          print("Enter!")
4
5      def __exit__(self, *exc):
6          print("Exit!")
7
8
9  with MyContextManager():
10     print("Inside the block!")
```

There is an easier way!

contextlib.contextmanager

```
class MyContextManager:
    def __enter__(self):
        print("Enter!")

    def __exit__(self, *exc):
        print("Exit!")


with MyContextManager():
    print("Inside the block!")
```

```
@contextmanager
def my_context_manager():
    print("Enter!")
    yield
    print("Exit!")


with my_context_manager():
    print("Inside the block!")
```

There is an easier way!

contextlib.contextmanager

```
class MyContextManager:
    def __enter__(self):
        print("Enter!")

    def __exit__(self, *exc):
        print("Exit!")


with MyContextManager():
    print("Inside the block!")
```

```
@contextmanager
def my_context_manager():
    print("Enter!")
    yield
    print("Exit!")


with my_context_manager():
    print("Inside the block!")
```

There is an easier way!

contextlib.contextmanager

```
class MyContextManager:
    def __enter__(self):
        print("Enter!")

    def __exit__(self, *exc):
        print("Exit!")


with MyContextManager():
    print("Inside the block!")
```

```
@contextmanager
def my_context_manager():
    print("Enter!")
    yield
    print("Exit!")


with my_context_manager():
    print("Inside the block!")
```

Old school context management

```
 1 class FoodContextManager:
 2     def __init__(self, data):
 3         self.data = data
 4
 5     def __enter__(self):
 6         print(f"Enter: {self.data}")
 7         return self.data
 8
 9     def __exit__(self, *exc):
10         print(f"Exit: {self.data}")
11
12 with FoodContextManager({"dairy":"yuck"}) as data:
13     data["fruit"] = "delicious"
```

New hotness

contextlib.contextmanager

```
 1 from contextlib import contextmanager
 2
 3 @contextmanager
 4 def FoodContextManager(data):
 5     print(f"Enter: {data}")
 6     yield data
 7     print(f"Exit: {data}")
 8
 9 with FoodContextManager({"dairy": "yuck"}) as data:
10     data["fruit"] = "delicious"
```

New hotness

contextlib.contextmanager

```
 1 from contextlib import contextmanager
 2
 3 @contextmanager
 4 def FoodContextManager(data):
 5     print(f"Enter: {data}")
 6     yield data
 7     print(f"Exit: {data}")
 8
 9 with FoodContextManager({"dairy": "yuck"}) as data:
10     data["fruit"] = "delicious"
```

New hotness

contextlib.contextmanager

```
 1 from contextlib import contextmanager
 2
 3 @contextmanager
 4 def FoodContextManager(data):
 5     print(f"Enter: {data}")
 6     yield data
 7     print(f"Exit: {data}")
 8
 9 with FoodContextManager({"dairy": "yuck"}) as data:
10     data["fruit"] = "delicious"
```

dport.me/pycon.pdf

Phew, that was a lot

Let's take a moment

Phew, that was a lot

Let's take a moment

- Context managers have:
  - `__enter__()`
  - `__exit__()`

Phew, that was a lot

Let's take a moment

- Context managers have:
  - `__enter__()`
  - `__exit__()`
- Generators and decorators are a thing that exist.

Phew, that was a lot

Let's take a moment

- Context managers have:
  - `__enter__()`
  - `__exit__()`
- Generators and decorators are a thing that exist.
- `@contextmanager` decorator is neat!

All the rest

Some things you should really know

- Scope
- Exceptions in `__exit__`
- try / finally with @contextmanager

Scope with Context Managers

Variables defined inside it still exist!

```
1 with open("myfile.txt") as f:
2     content = f.read()
3 print(content)
```

Scope with Context Managers

The thing yielded does too, but it'll be closed.

```
1 with open("myfile.txt") as f:
2     pass
3 content = f.read()
4 print(content)
5 # ValueError: I/O operation on closed file.
```

All the rest

Some things you should really know

- Scope
- **Exceptions in __exit__**
- try / finally with @contextmanager

Exceptions in __exit__

Making context managers even more powerful 💪

```
1  class MyContextManager:
2      def __enter__(self):
3          print("Enter!")
4
5      def __exit__(self, *exc):
6          print("Exit!")
7
8
9  with MyContextManager():
10     print("Inside the block!")
```

Exceptions in __exit__

Making context managers even more powerful 💪

```
1  class MyContextManager:
2      def __enter__(self):
3          print("Enter!")
4
5      def __exit__(self, *exc):
6          print("Exit!")
7
8
9  with MyContextManager():
10     print("Inside the block!")
```

```
def __exit__(
    self,
    exc_type,
    exc,
    exc_tb,
):
    print("Exit!")
```

Exception handling in __exit__

It's a little tricky

- Want to ignore exception?
  - Return `True`
- Want to raise exception?
  - Return `False` (or do nothing)
- Do **not** explicitly re-raise the exception.

Exceptions in __exit__

Making context managers even more powerful 💪

```
1 class MyContextManager:
2     ...
4
5     def __exit__(self, exc_type, exc, exc_tb):
6         if exc:
7             print("Oh no!")
8             call_for_help()
9             return False
```

Exceptions in __exit__

Making context managers even more powerful 💪

```
1 class MyContextManager:
2     ...
4
5     def __exit__(self, exc_type, exc, exc_tb):
6         if exc:
7             print("Oh no!")
8             call_for_help()
```

Exceptions in __exit__

Making context managers even more powerful 💪

```
 1 class suppress:
 2     def __init__(self, *exceptions):
 3         self.exceptions = exceptions
 4
 5     def __enter__(self):
 6         pass
 7
 8     def __exit__(self, exc_type, exc, exc_tb):
 9         return (
10             exc_type is not None and
11             issubclass(exc_type, self.exceptions)
12         )
```

Exceptions in __exit__

Making context managers even more powerful 💪

```
 1 class suppress:
 2     def __init__(self, *exceptions):
 3         self.exceptions = exceptions
 4
 5     def __enter__(self):
 6         pass
 7
 8     def __exit__(self, exc_type, exc, exc_tb):
 9         return (
10             exc_type is not None and
11             issubclass(exc_type, self.exceptions)
12         )
```

Exceptions in __exit__

Making context managers even more powerful 💪

```
 1 class suppress:
 2     def __init__(self, *exceptions):
 3         self.exceptions = exceptions
 4
 5     def __enter__(self):
 6         pass
 7
 8     def __exit__(self, exc_type, exc, exc_tb):
 9         return (
10             exc_type is not None and
11             issubclass(exc_type, self.exceptions)
12         )
```

Exceptions in __exit__

Making context managers even more powerful 💪

```
 1 class suppress:
 2     def __init__(self, *exceptions):
 3         self.exceptions = exceptions
 4
 5     def __enter__(self):
 6         pass
 7
 8     def __exit__(self, exc_type, exc, exc_tb):
 9         return (
10             exc_type is not None and
11             issubclass(exc_type, self.exceptions)
12         )
```

All the rest

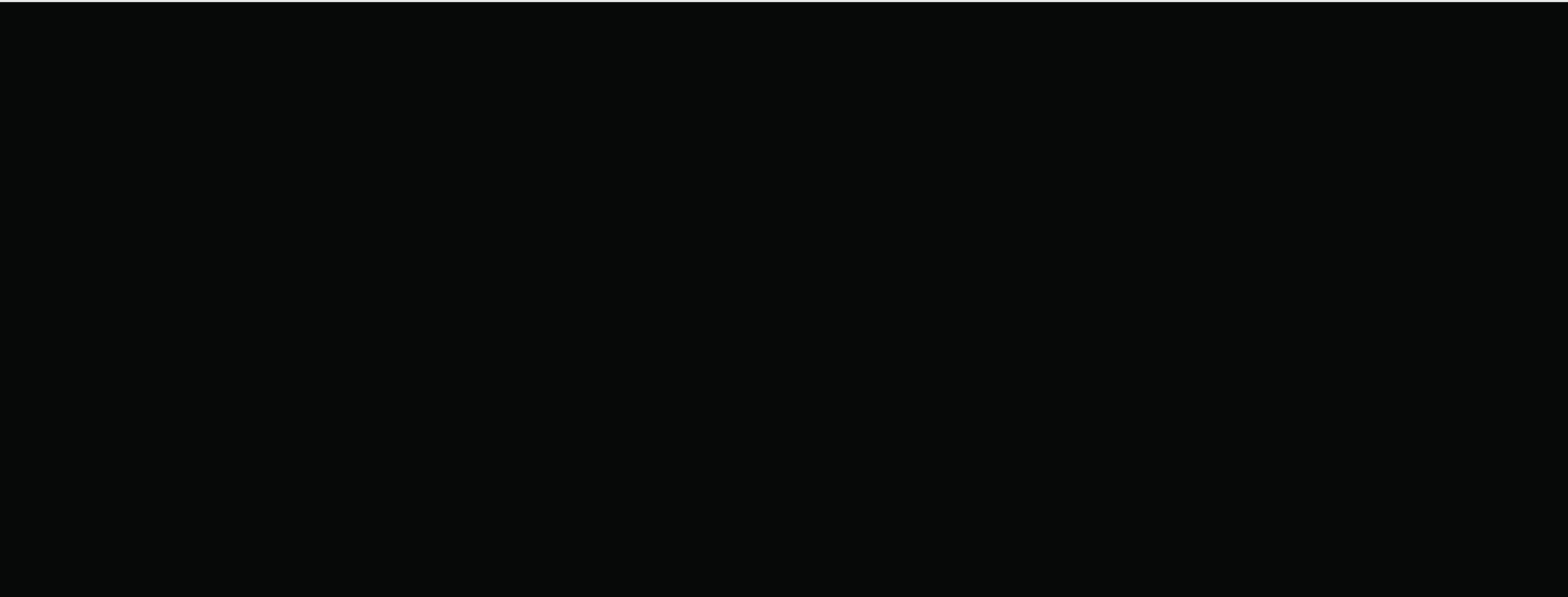
Some things you should really know

- Scope
- Exceptions in `__exit__`
- **Exception handling in @contextmanagers**

dport.me/pycon.pdf

try / except / finally

Exception handling in @contextmanagers



try / except / finally

Exception handling in @contextmanagers

```
1 @contextmanager
2 def my_context_manager():
3     print("Before")
4     yield
5     print("After")
6
7 with my_context_manager():
8     print(f"Neat: {1/0}")
```

try / except / finally

Exception handling in @contextmanagers

```
1 @contextmanager
2 def my_context_manager():
3     print("Before")
4     yield
5     print("After")
6
7 with my_context_manager():
8     print(f"Neat: {1/0}")
```

try / except / finally

Exception handling in @contextmanagers

```
$ python3 16_contextmanager_exceptions.py
Before
Traceback (most recent call last):
  File "example.py", line 10, in <module>
    print(f"I love this number: {1/0}")
ZeroDivisionError: division by zero
```

try / except / finally

Exception handling in @contextmanagers

```
1 @contextmanager
2 def my_context_manager():
3     print("Before")
4     yield
5     print("After")
6
7 with my_context_manager():
8     print(f"Neat: {1/0}")
```

try / except / finally

Exception handling in @contextmanagers

```
1 @contextmanager
2 def my_context_manager():
3     print("Before")
4     yield
5     print("After")
6
7 with my_context_manager():
8     print(f"Neat: {1/0}")
```

```
1 @contextmanager
2 def my_context_manager():
3     print("Before")
4     try:
5         yield
6     except Exception as e:
7         print(f"Oh no: {e}")
8     finally:
9         print("After")
```

try / except / finally

Exception handling in @contextmanagers

```
1 @contextmanager
2 def my_context_manager():
3     print("Before")
4     yield
5     print("After")
6
7 with my_context_manager():
8     print(f"Neat: {1/0}")
```

```
1 @contextmanager
2 def my_context_manager():
3     print("Before")
4     try:
5         yield
6     except Exception as e:
7         print(f"Oh no: {e}")
8     finally:
9         print("After")
```

try / except / finally

Exception handling in @contextmanagers

```
$ python3 16_contextmanager_exceptions.py
Before
Oh no: division by zero
After
```

We made it!

dport.me/pycon.pdf

Best practices

Best practices

- Do not explicitly re-raise exceptions in `__exit__` methods.

Best practices

- Do not explicitly re-raise exceptions in `__exit__` methods.
- In @contextmanagers however you must re-raise.

Best practices

- Do not explicitly re-raise exceptions in `__exit__` methods.
- In @contextmanagers however you must re-raise.
- Know the roles of `__init__` and `__enter__`
  - No side effects in `__init__`
  - Don’t make `__init__` too computationally expensive.

Other possible uses!

So many!

- Enclose an event and log based on what happens.

Other possible uses!

So many!

- Enclose an event and log based on what happens.
- Remote integration tests.

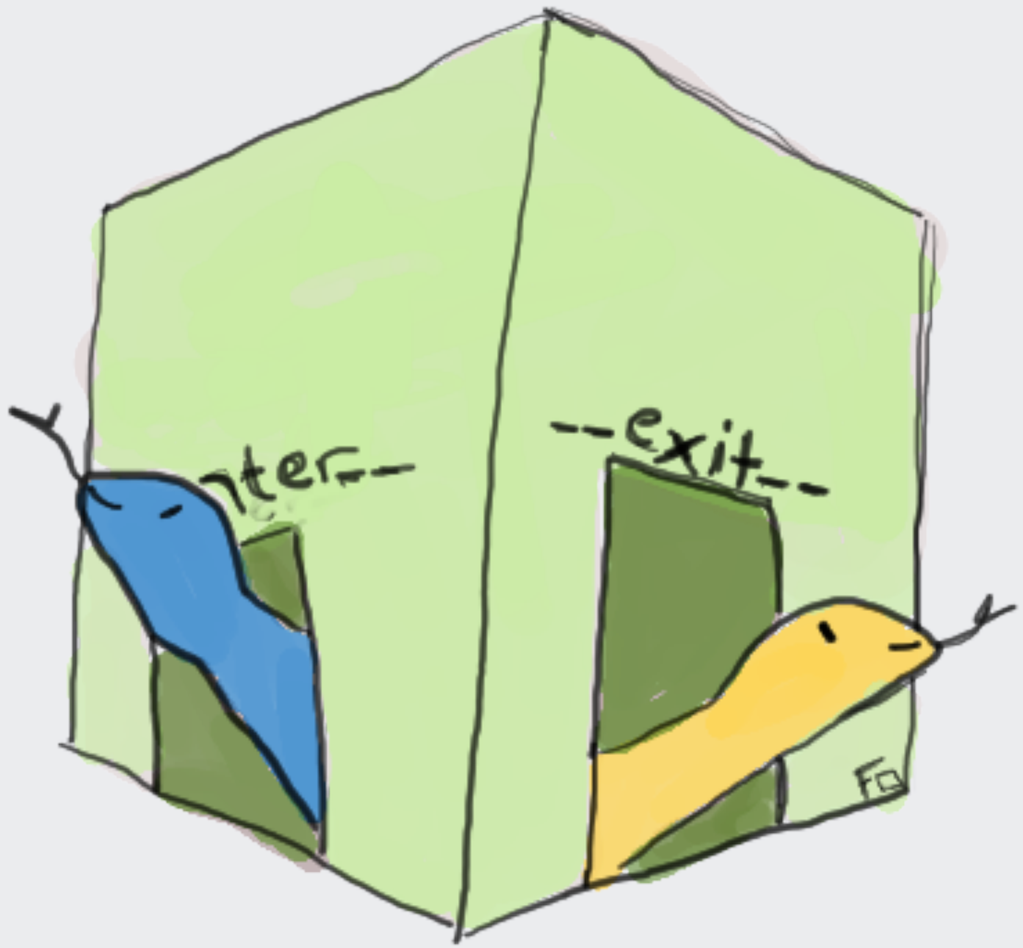
Other possible uses!

So many!

- Enclose an event and log based on what happens.
- Remote integration tests.
- More!

"Context managers are one honking great idea -- let's use more of those!"

—Tim Peters (almost)



- @fatty_box

Thanks to these great folks

😍😍😍😍😍

- Lachlan, John, Luka, and Sam
  - For watching all my dry-runs
- Fatema and Kelly
  - For drawings and moral support
- The Pycon AU team!!
- All of you lovely people 😊

Questions?

Tweeter

@banool1

Website

dport.me

Github

github.com/banool

facebook

Key take aways

🔑🔑🔑🔑🔑

- Use context managers!
- Context managers are one honking great idea -- let's use more of those!
- **Use context managers but bolded!**

dport.me/pycon.pdf

First, the hard way

But not necessarily the wrong way

e. hmm open is a builtin and threadpoolexecutor is too complex.
suppress would necessitate explanation of exception handling

Typical usage:

```
@contextmanager
def some_generator(<arguments>):
    <setup>
    try:
        yield <value>
    finally:
        <cleanup>
```

equivalent to this:

```
<setup>
try:
    <variable> = <value>
    <body>
finally:
    <cleanup>
```

This makes this:

```
with some_generator(<arguments>) as <variable>:
    <body>
```