

**facebook**

# Context Managers: You Can Write Your Own!

**Daniel Porteous**

Production Engineer

[dport.me/pycon.pdf](https://dport.me/pycon.pdf)



# Me!

Me!!! Daniel Porteous!

**Tweeter:**

@banool1000

**Github:**

[github.com/banool](https://github.com/banool)

**Website:**

[dport.me](https://dport.me)





# Agenda

## TODO

- What is a context manager?
- Why use context managers?
- Core Python examples.
- Making our own context managers the hard way.
- The easy way!
- Best practices, gotchas, and more.

# What are context managers?

You've seen them, you just don't know it!

# What are context managers?

You've seen them, you just don't know it!

1 with

# What are context managers?

You've seen them, you just don't know it!

1 with

as

# What are context managers?

You've seen them, you just don't know it!

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



# Why use context managers?

They're pretty and safe is why!

- You can't forget to close resources.
- They can make code much prettier.
- They can make complex logic simpler.
- More!

# Why use context managers?

They're pretty and safe is why!

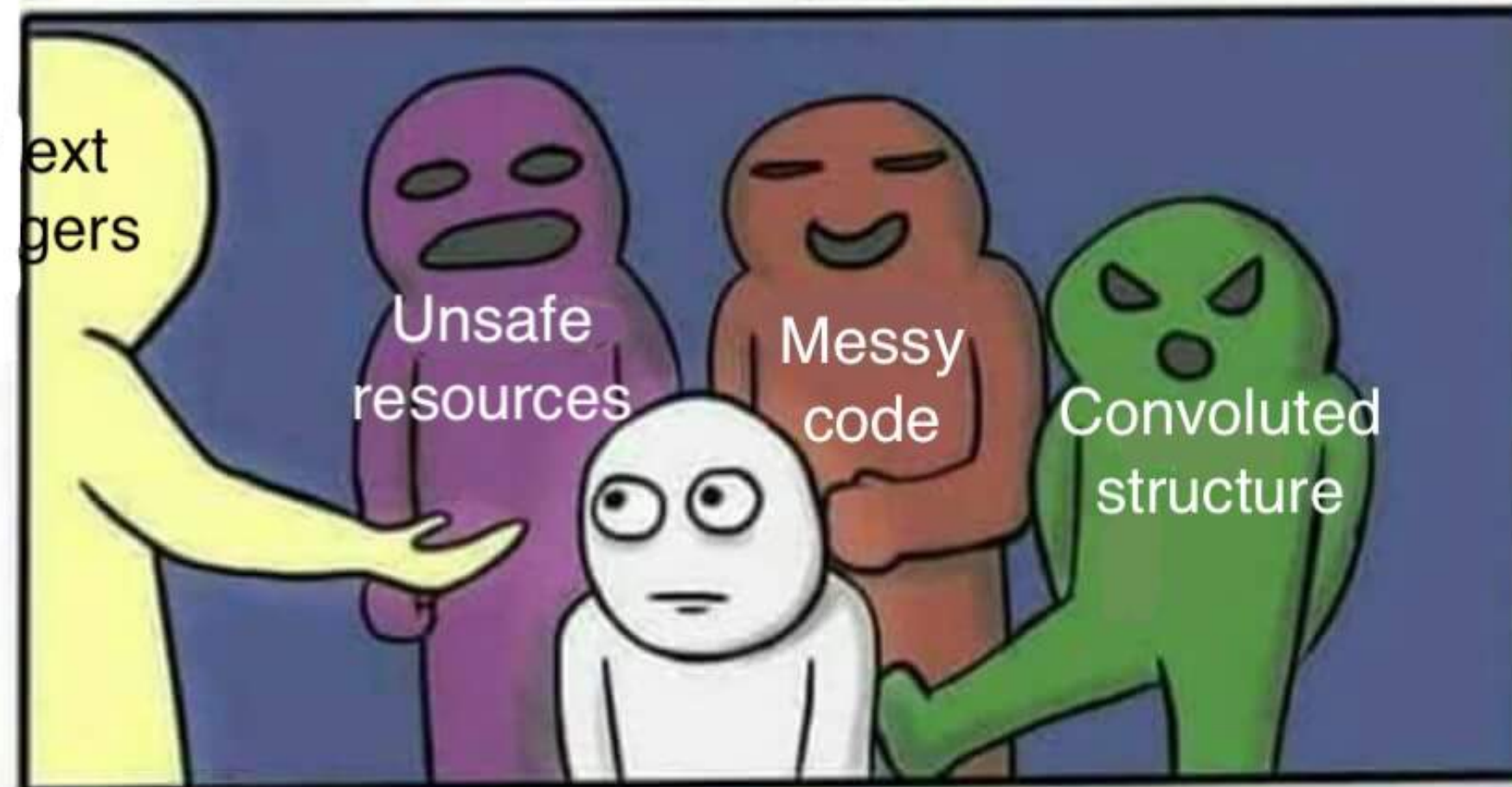
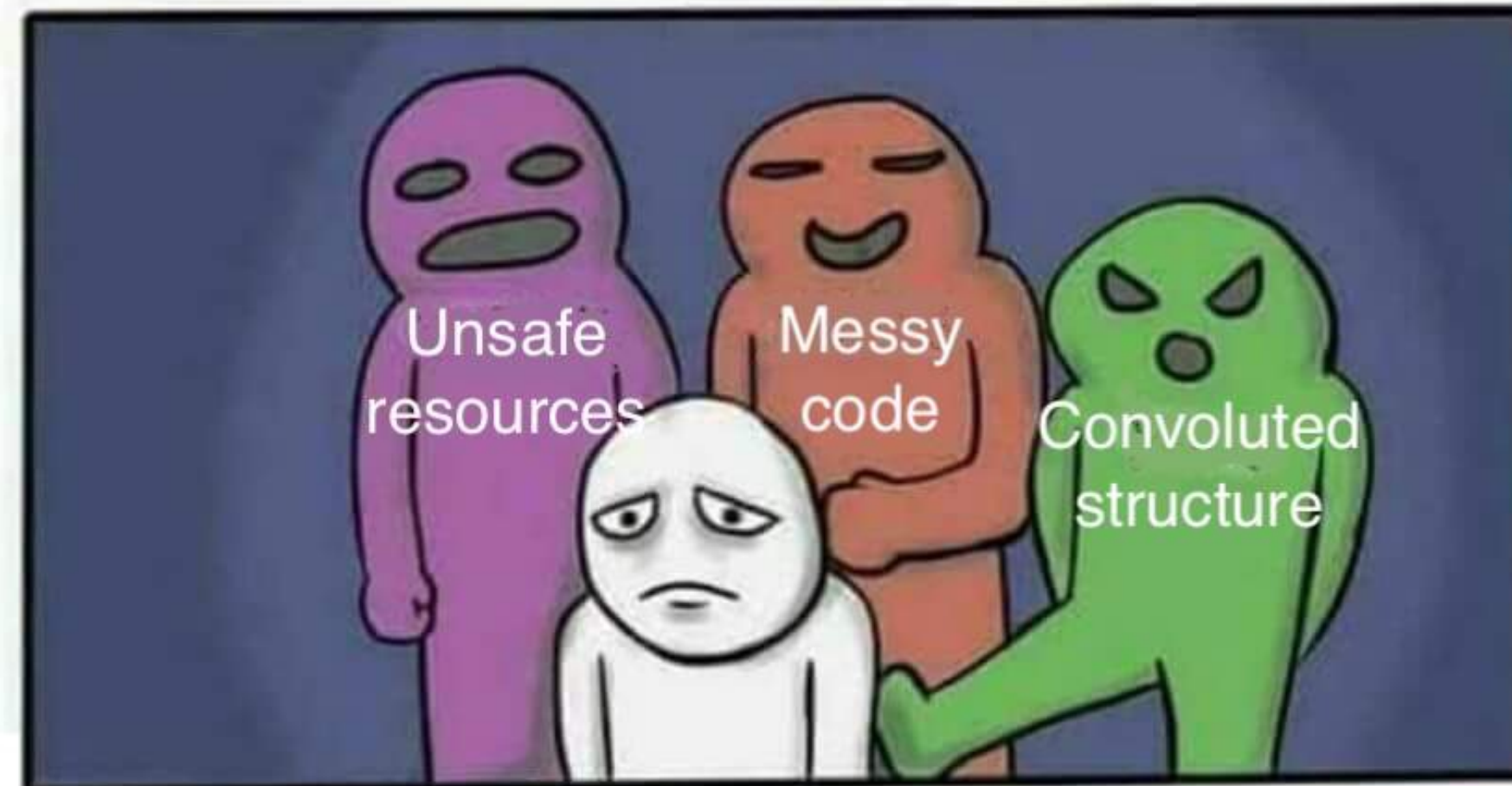
```
>>> import this
```

The Zen of Python, by Tim Peters

Beautiful is better than ugly.

Simple is better than complex.

Readability counts.



# 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)
```

**Write your own context managers!**

# First, the hard way

But not necessarily the wrong way

write a context manager for something we've seen before. hmm open is a builtin and threadpoolexecutor is too complex. suppress would necessitate explanation of exception handling

# 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!")
```

```
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"
```

“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.







# 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
```

```
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)))
```



# 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()
```

# 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)
```

# Let's put them together

With our powers combined

Generators

Decorators



# 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`

```
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!")
```

```
@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`

```
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!")
```

```
@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`

```
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!")
```

```
@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"
```

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>
```

# 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` in generator context managers

# 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 in generator context managers



# 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`
- 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 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             isinstance(exc_type, self.exceptions)
12         )
```

# All the rest

Some things you should really know

- Scope
- Exceptions in `__exit__`
- **try / finally in generator context managers**

# try / finally

TODO



# Best practices

## TODO

- Do not explicitly re-raise exceptions in `__exit__` methods, return `False`.
- Know the roles of `__init__` and `__enter__`
  - No side effects in `__init__`
  - Don't make `__init__` too expensive.

# Other possible uses!

So many!

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

# Key take aways



- Use context managers!
- **Use context managers but bolded!**

Questions?

**facebook**