DECORATING GENERATOR FUNCTIONS

So far...

we saw how to create a context manager using a class and a generator function

```
def gen_function(args):
    …
    try:
        yield obj
    finally:
        …
```

single yield — the return value of `__enter__`

cleanup phase — `__exit__`

```
class GenContextManager:
    def __init__(gen_func):
        self.gen = gen_func()

    def __enter__(self):
        return next(self.gen)

    def __exit__(self, …):
        next(self.gen)
```

returns what was yielded

runs the finally block

Usage

```
with GenContextManager(gen_func):
    …
```

We can tweak this a bit to also allow passing in arguments to `gen_func`

And usage now becomes:

```
gen = gen_func(args)

with GenContextManager(gen):
    …
```

This works, but we have to create the generator object first, and use the `GenContextManager` class

→ lose clarity of what the context manager is

```
class GenContextManager:
    def __init__(gen_obj):
        self.gen = gen_obj

    def __enter__(self):
        return next(self.gen)

    def __exit__(self, …):
        next(self.gen)
```

Using a decorator to encapsulate these steps

```
gen = gen_func(args)
with GenContextManager(gen):
    …
```

```
class GenContextManager:
    def __init__(gen_obj):
        self.gen = gen_obj

    def __enter__(self):
        return next(self.gen)

    def __exit__(self, …):
        next(self.gen)
```

```
def contextmanager_dec(gen_fn):
    def helper(*args, **kwargs):
        gen = gen_fn(*args, **kwargs)
        return GenContextManager(gen)
    return helper
```

Usage Example

```
@contextmanager_dec
def open_file(f_name):
    f = open(f_name)
    try:
        yield f

    finally:
        f.close()
```

```
def contextmanager_dec(gen_fn):
    def helper(*args, **kwargs):
        gen = gen_fn(*args, **kwargs)
        return GenContextManager(gen)
    return helper
```

→ `open_file = contextmanager_dec(open_file)`

→ `open_file` is now actually the `helper` closure

calling `open_file(f_name)`

→ calls `helper(f_name)` [free variable `gen_fn = open_file`]

→ creates the generator object

→ returns `GenContextManager` instance

→ `with open_file(f_name)`

The `contextlib` Module

One of the goals when context managers were introduced to Python

PEP 343

was to ensure generator functions could be used to easily create them

Technique is basically what we came up with

→ more complex → exception handling

→ if an exception occurs in `with` block, needs to be propagated back to generator function

```
__exit__(self, exc_type, exc_value, exc_tb)
```

→ enhanced generators as coroutines → later

This is implemented for us in the standard library:

```
contextlib.contextmanager
```

→ decorator which turns a generator function into a context manager

Coding Exercises