Python Decorators

Gift or Poison?

Anastasiia Tymoshchuk
Cyren GmbH

Presentation Slides

https://atymo.me/presentations/GiftOrPoison/

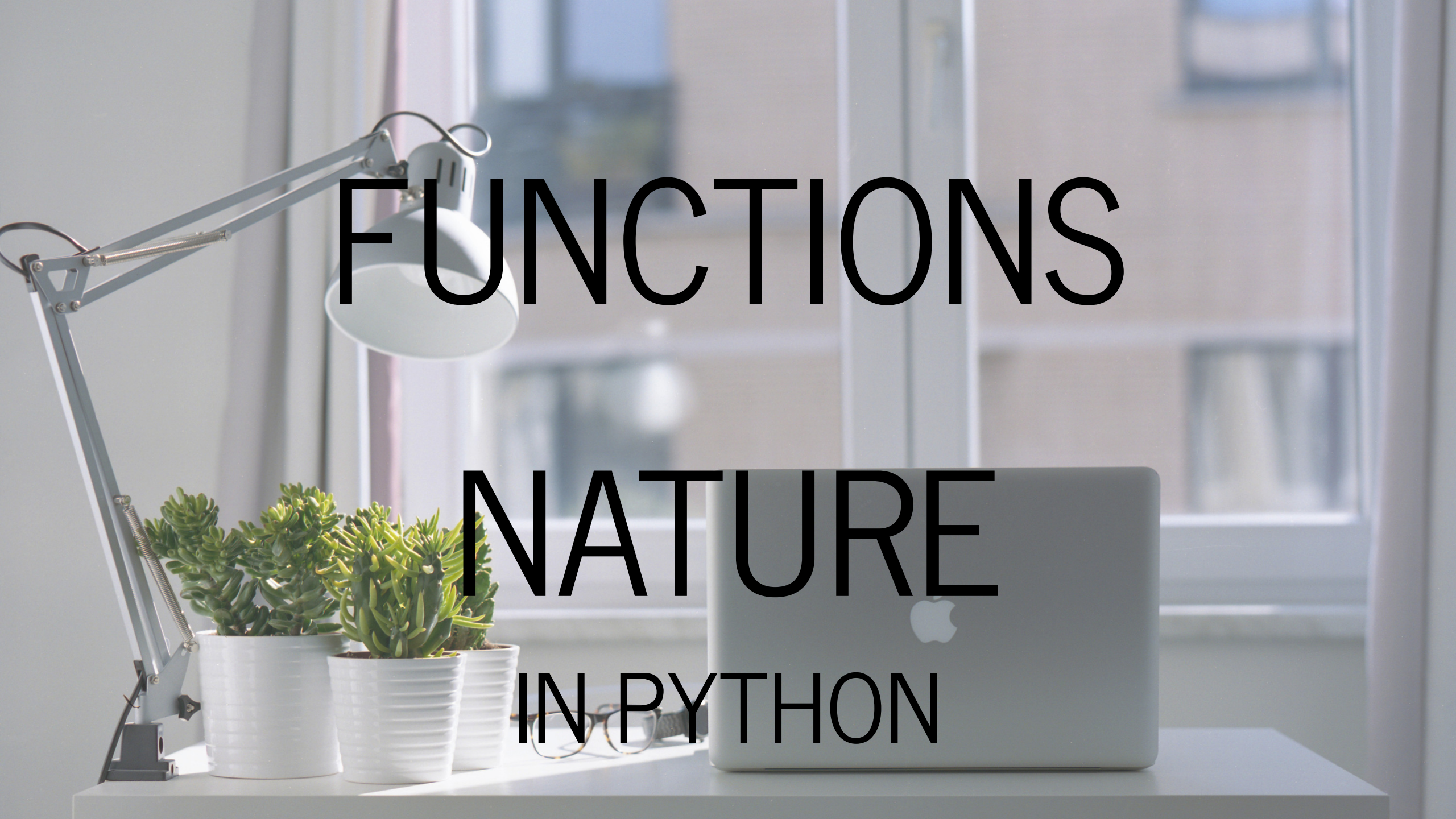
Code snippets

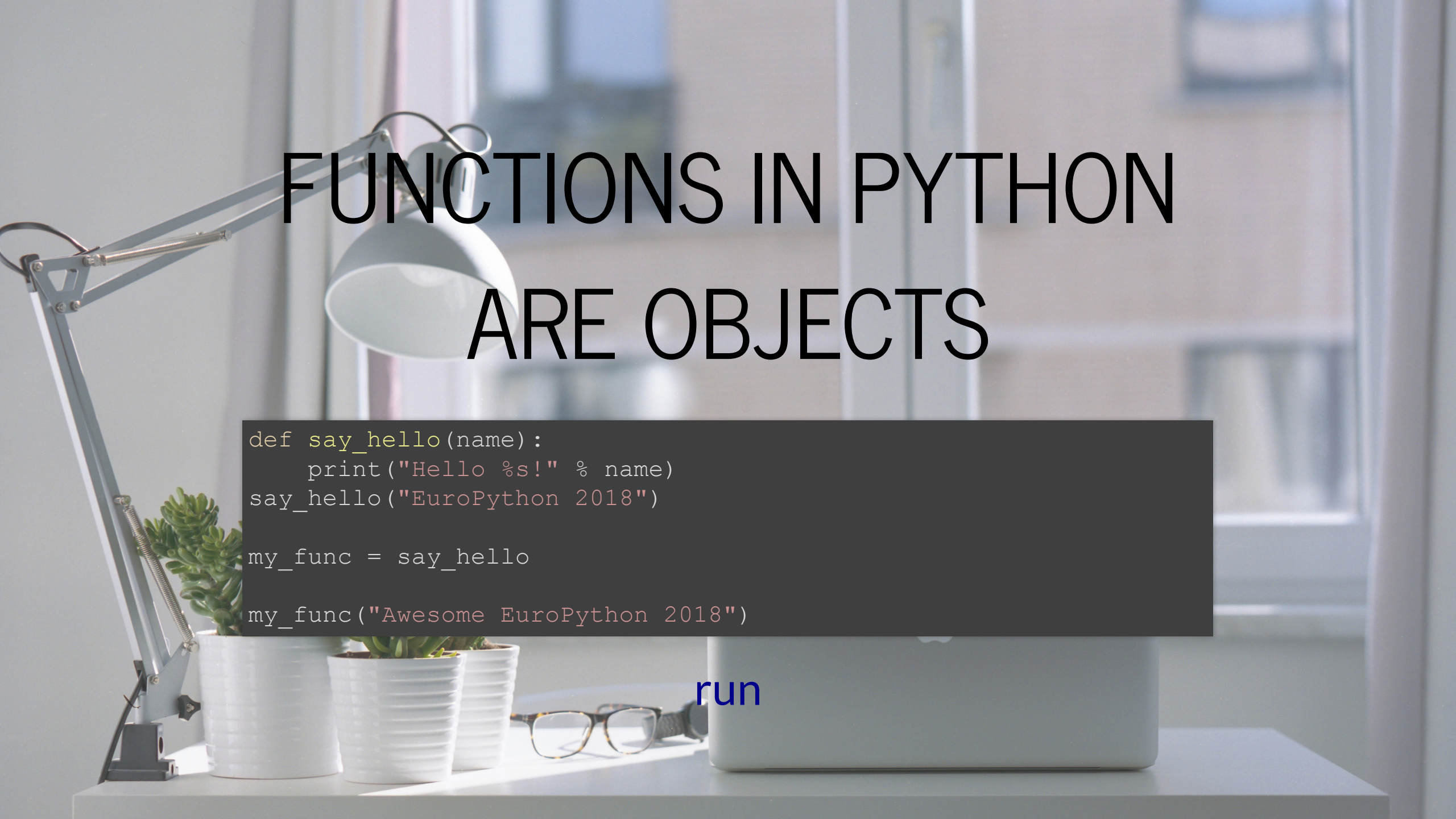
https://github.com/atymoshchuk/python_tutorials


CYREN
All about security
25B
Security Transactions Daily
1.3B
Users Protected
300M
Threats Blocked Daily

What's in the Talk

1. Functions nature in Python
2. Magic of a Decorator
3. Basics
4. When to use Decorators
5. Examples


FUNCTIONS
NATURE
IN PYTHON



FUNCTIONS IN PYTHON ARE OBJECTS

```
def say_hello(name):
    print("Hello %s!" % name)
say_hello("EuroPython 2018")

my_func = say_hello

my_func("Awesome EuroPython 2018")
```

run


Magic of
Decorators


There are two types of decorators
1. Function Decorators
Added in Python 2.4
2. Class Decorators
Added in Python 2.6


Function
Decorators

Basic Function Decorator

```
def mydecorator(decorated_func):
    def wrapped(*args, **kwargs):
        print("Something happened in decorator!")
        return decorated_func(*args, **kwargs)
    return wrapped

@mydecorator

def myfunc(myarg):
    print("my function", myarg)

def mysecond_func(myarg):
    print("my second function", myarg)

myfunc('for the Talk')
mysecond_func = mydecorator(mysecond_func)
```

run

Stacked Function Decorator

```
def first_dec(func):
    def wrapped(*args, **kwargs):
        print("Something happened in First decorator!")
        return func(*args, **kwargs)
    return wrapped

def second_dec(func):

    def wrapped(*args, **kwargs):
        print("Something happened in Second decorator!")
        return func(*args, **kwargs)
    return wrapped
@first_dec
@second_dec
def myfunc(myarg):
    print("my function", myarg)
```

run


Class
Decorators

Basic Class Decorator

```
class my_decorator:
    ....
    # probably a lot of code here

@my_decorator
class MyClass:
    def do_something(self):
        ...
```

Class as a decorator

```
class entry_exit(object):

    def __init__(self, f):
        self.f = f

    def __call__(self):
        print("Entering", self.f.__name__)

        self.f()
        print("Exited", self.f.__name__)

@entry_exit
def func1():
    print("inside func1()")

@entry_exit
```

run


Dive into Basics

How decorator works?

```
def mydecorator(decorated_func):
    def wrapped(*args, **kwargs):
        print("Before decorated function")
        result = decorated_func(*args, **kwargs)
        print("After decorated function")
        return result
    return wrapped


@mydecorator
def myfunc(myarg):
    """prints some text combined with a string from argument"""
    print("my function", myarg)
    return "return value"

r = myfunc('for the Talk')
```

run

Why to use @wraps?

```
from functools import wraps

def mydecorator(f):
    @wraps(f)
    def wrapped(*args, **kwargs):
        print("Before decorated function")
        r = f(*args, **kwargs)

        print("After decorated function")
        return r
    return wrapped

@mydecorator
def myfunc(myarg):
    """prints some text combined with a string from argument"""
    print("my function", myarg)
```

run


When to use
Decorators?

Timing with Function Decorators

```
import time

def timeit(method):

    def timed(*args, **kw):
        ts = time.time()
        result = method(*args, **kw)

        te = time.time()

        print('%r (%r, %r) %2.2f sec' % (method.__name__, args, k
        return result

    return timed

class Foo(object):
```

run

Timing with Class Decorators

```
class ImportantStuff(object):
    @time_this
    def do_stuff_1(self):
        ...
    @time_this
    def do_stuff_2(self):
        ...
    @time_this
    def do_stuff_3(self):
        ...
```

```
@time_all_class_methods
class ImportantStuff:
    def do_stuff_1(self):
        ...
    def do_stuff_2(self):
        ...
    def do_stuff_3(self):
        ...
```

```
def time_this(original_function):
    print("decorating")
    def new_function(*args,**kwargs):
        print("starting timer")
        import datetime
        before = datetime.datetime.now()
        x = original_function(*args,**kwargs)
        after = datetime.datetime.now()
        print("Elapsed Time = {0}".format(after-before))
        return x
    return new_function

def time_all_class_methods(Cls):
    class NewCls(object):
        def __init__(self,*args,**kwargs):
            self.oInstance = Cls(*args,**kwargs)
```

run

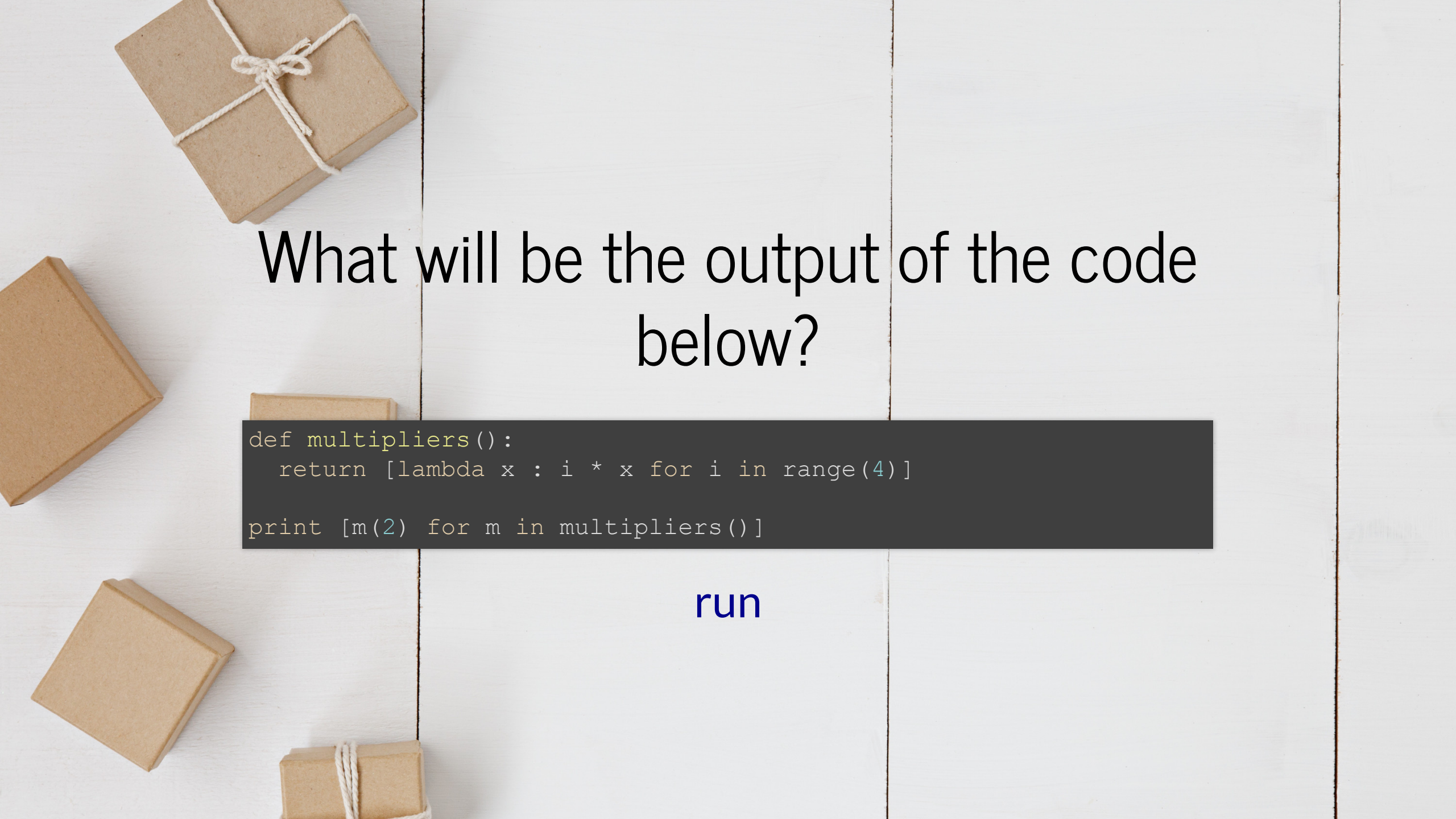
Examples

How to use decorators in unit testing

```
def add_tests(generator):
    def class_decorator(cls):
        """Add tests to `cls` generated by `generator()`."""
        for test_func, func, input, output in generator:
            test = lambda self, i=input, o=output, f=test_func, f
            test.__name__ = "test_%s(%r, %r)" % (func.__name__, i
            setattr(cls, test.__name__, test)

            # print("added:", cls, test.__name__, test)
        return cls

    return class_decorator


def test_cases(parameters):
    def t(self, func_to_test, data, result):
```

run


A bit more
Magic in Python



What will be the output of the code below?

```
def multipliers():
  return [lambda x : i * x for i in range(4)]

print [m(2) for m in multipliers()]
```

run

```
def multipliers():
  return [lambda x : i * x for i in range(4)]

print [m(2) for m in multipliers()]
```

Expectation

[0, 2, 4, 6]

Reality

[6, 6, 6, 6]

WHY?

How to fix?

```
def multipliers():
  for i in range(4): yield lambda x : i * x
```

or

```
def multipliers():
  return [lambda x, i=i : i * x for i in range(4)]
```

run

Are Decorators in Python

Gift or Poison?

Keep in touch

atymoshchuk@icloud.com

P.S. We are hiring!

www.cyren.com/about-cyren/careers