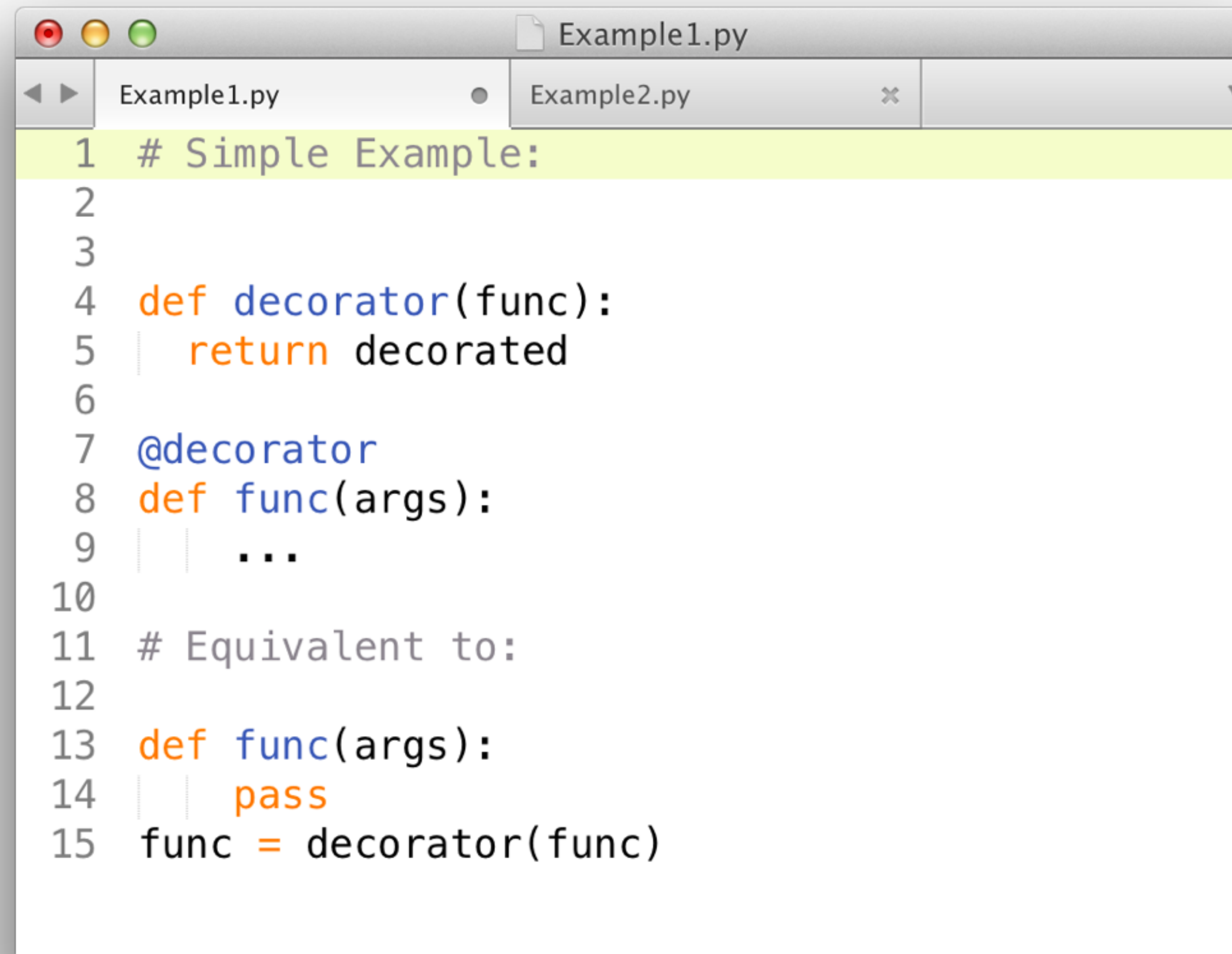


Decorators

And why they're a bit awesome

@stestagg

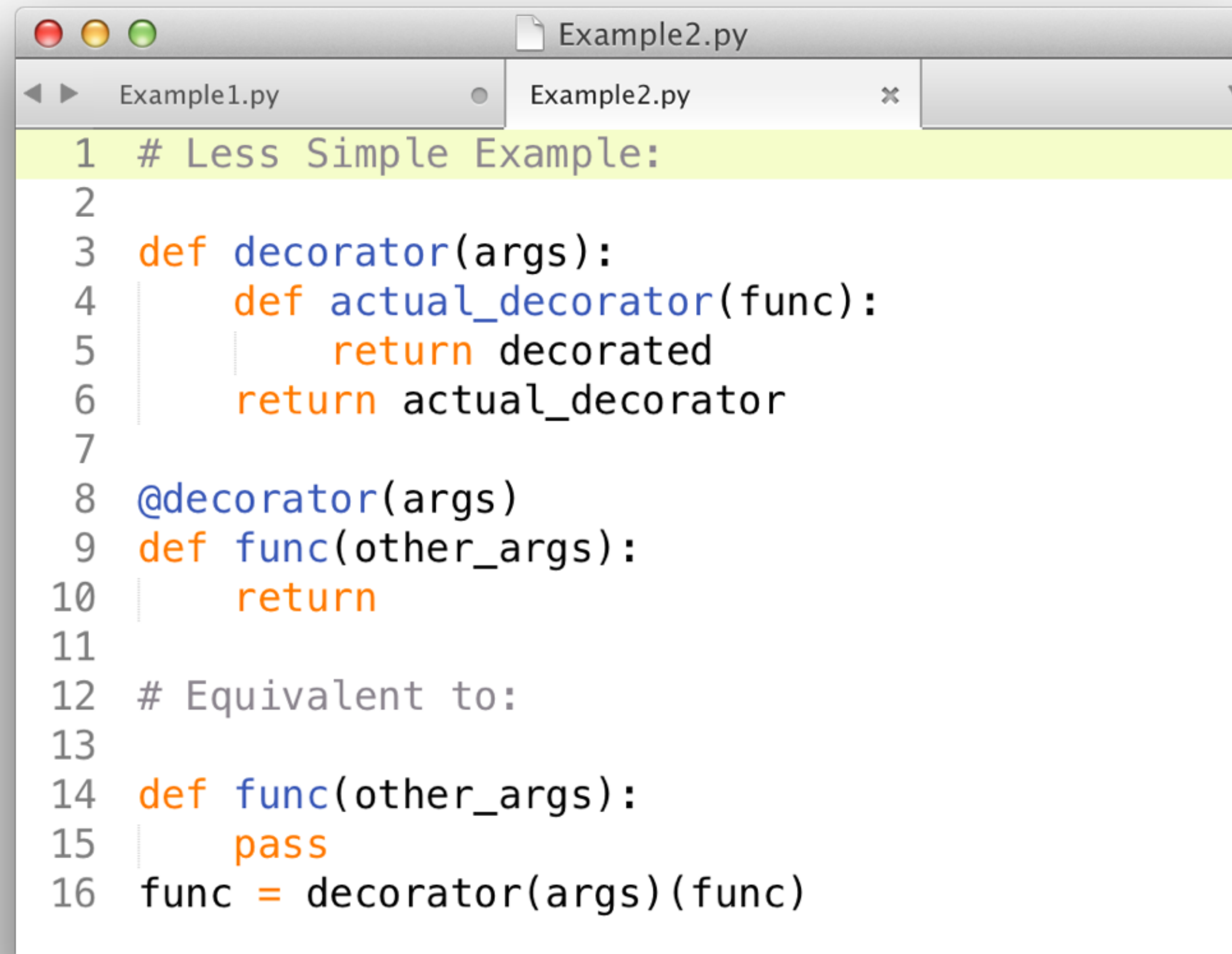
Syntax



The image shows a screenshot of a code editor window with a title bar containing three colored buttons (red, yellow, green) and a file name 'Example1.py'. Below the title bar is a tab bar with two tabs: 'Example1.py' (selected) and 'Example2.py'. The main editing area displays Python code with line numbers 1 through 15 on the left. The code defines a decorator function and applies it to another function. The first line is highlighted in yellow.

```
1 # Simple Example:
2
3
4 def decorator(func):
5     return decorated
6
7 @decorator
8 def func(args):
9     ...
10
11 # Equivalent to:
12
13 def func(args):
14     pass
15 func = decorator(func)
```

Syntax



```
1 # Less Simple Example:
2
3 def decorator(args):
4     def actual_decorator(func):
5         return decorated
6     return actual_decorator
7
8 @decorator(args)
9 def func(other_args):
10     return
11
12 # Equivalent to:
13
14 def func(other_args):
15     pass
16 func = decorator(args)(func)
```

What makes them Awesome?

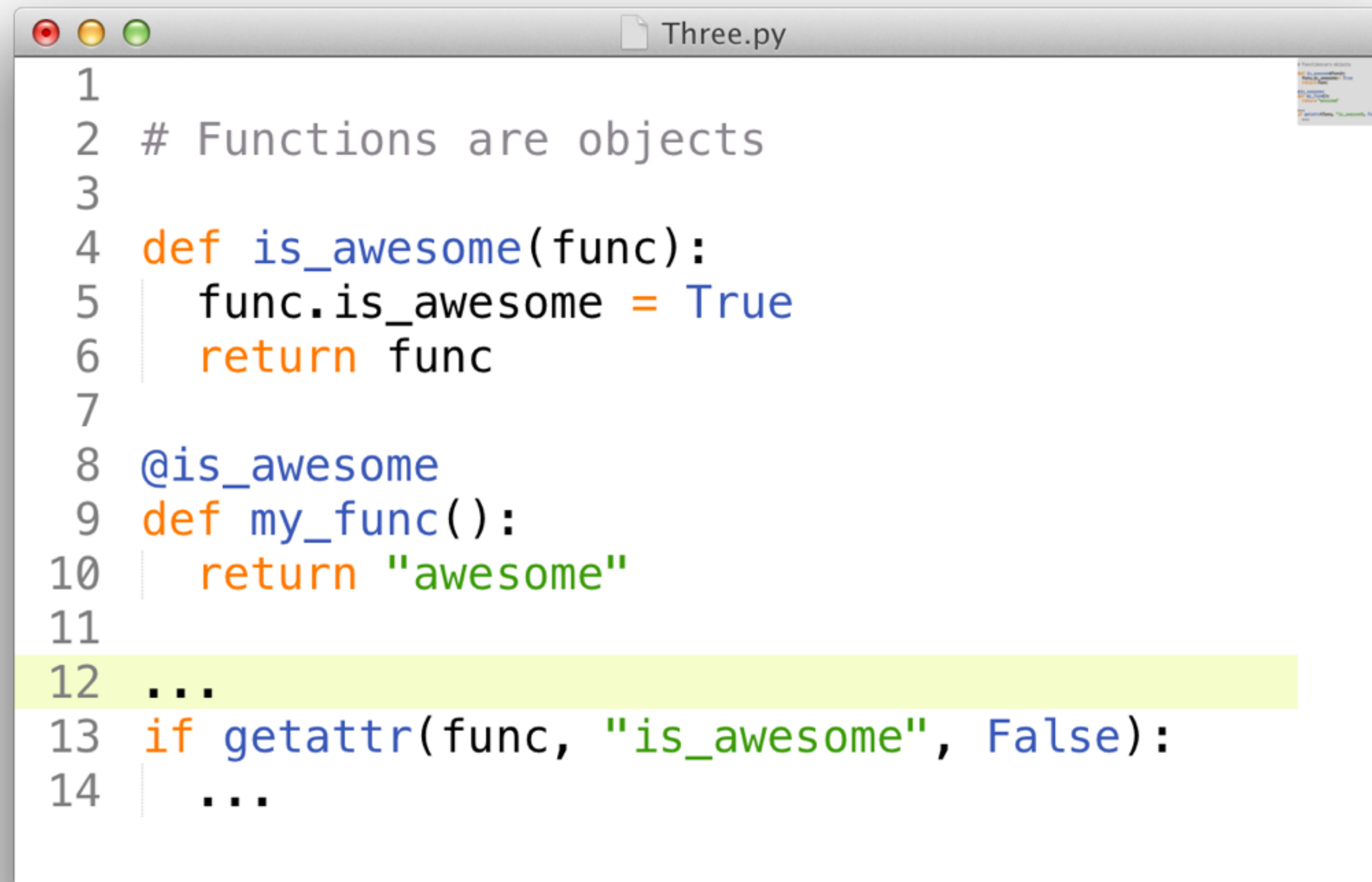
- `@property`
- Keep functions/methods clean and simple
- Document intent
- Hide boilerplate

Protip: Python functions are objects

Main uses:

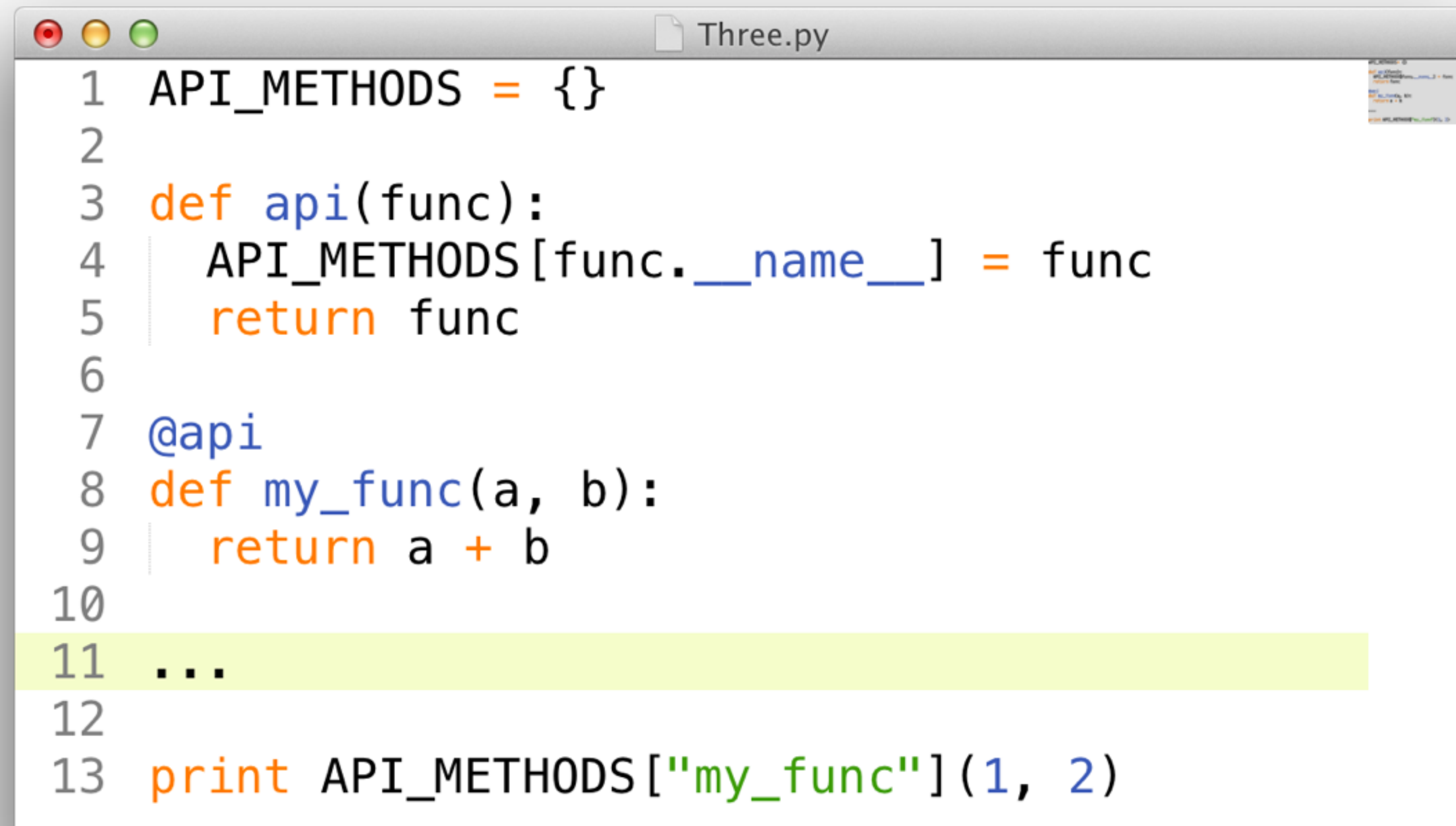
1. Annotation
2. Registration
3. Modification

1. Annotation



```
1
2 # Functions are objects
3
4 def is_awesome(func):
5     func.is_awesome = True
6     return func
7
8 @is_awesome
9 def my_func():
10     return "awesome"
11
12 ...
13 if getattr(func, "is_awesome", False):
14     ...
```

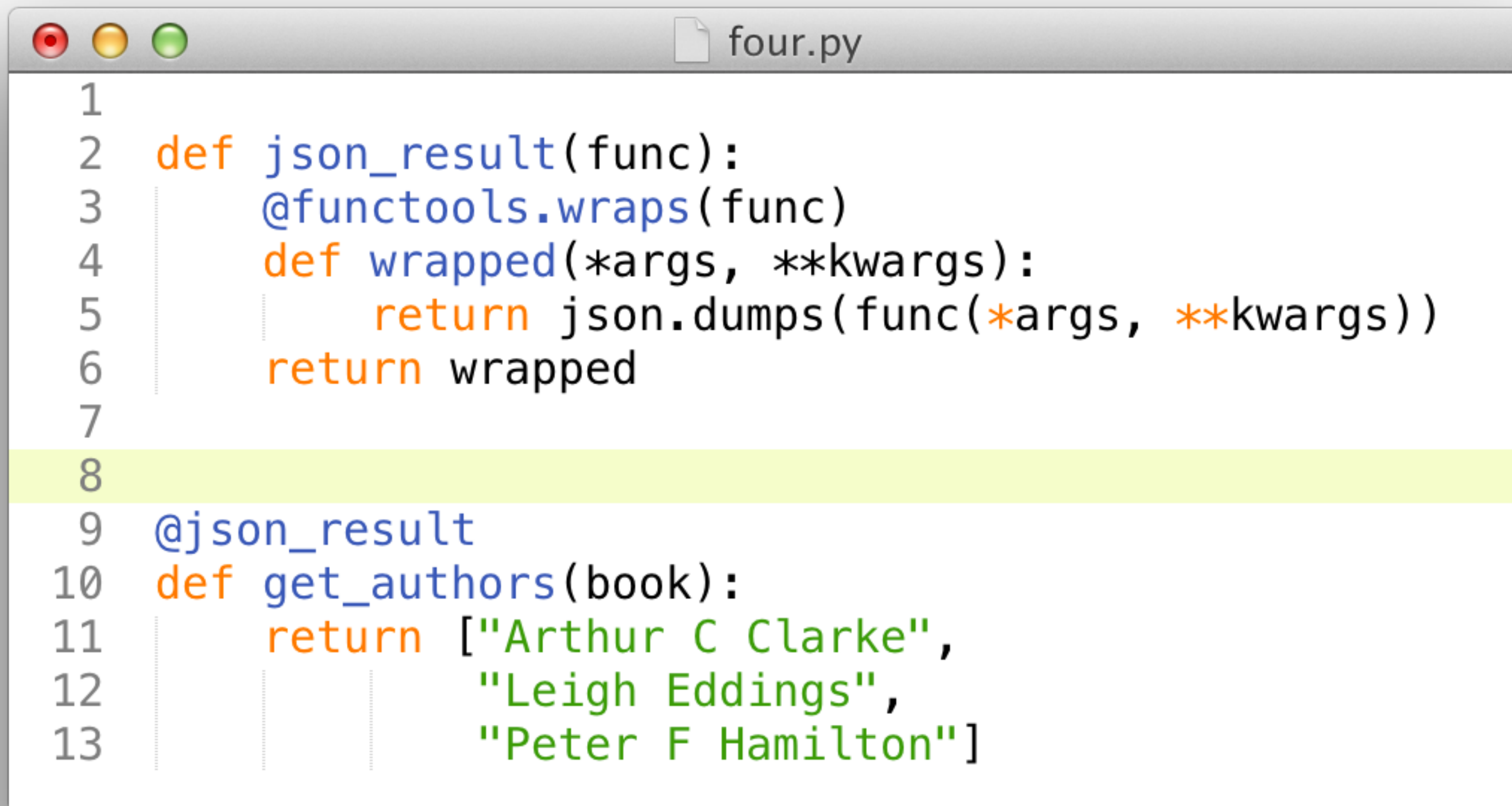
2. Registration



```
1 API_METHODS = {}
2
3 def api(func):
4     API_METHODS[func.__name__] = func
5     return func
6
7 @api
8 def my_func(a, b):
9     return a + b
10
11 ...
12
13 print API_METHODS["my_func"](1, 2)
```

3. Modification

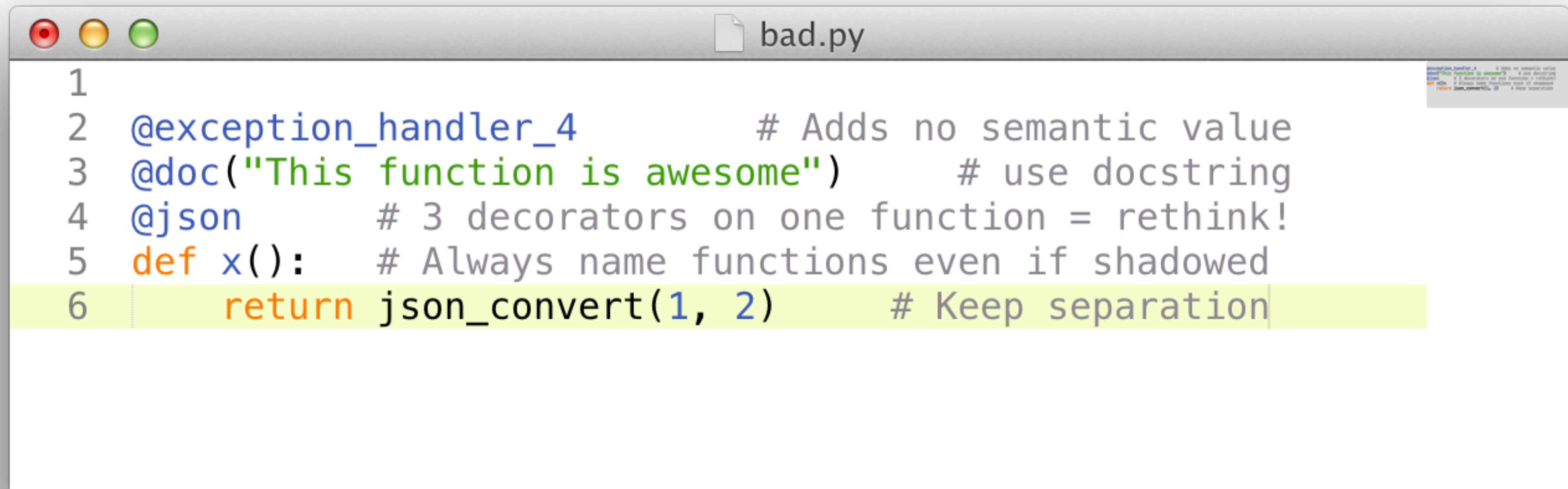
- More dangerous! - leads to surprising code
- Good for Serialisation / Exception handling etc



```
1
2 def json_result(func):
3     @functools.wraps(func)
4     def wrapped(*args, **kwargs):
5         return json.dumps(func(*args, **kwargs))
6     return wrapped
7
8
9 @json_result
10 def get_authors(book):
11     return ["Arthur C Clarke",
12            "Leigh Eddings",
13            "Peter F Hamilton"]
```

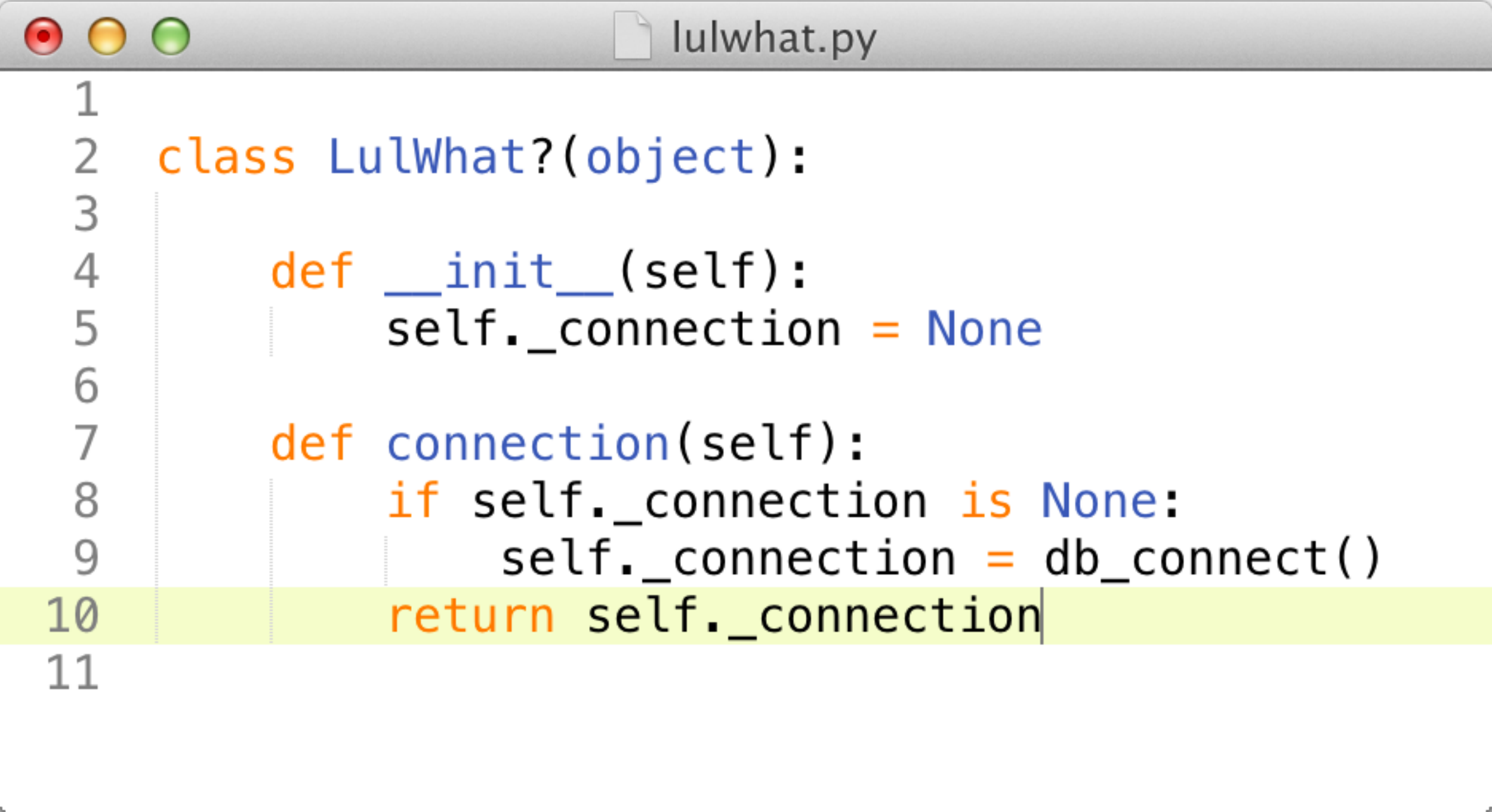

Decorators: Bad bits

- **Decorators tell a story**
- **Be sensible!**



```
1
2 @exception_handler_4      # Adds no semantic value
3 @doc("This function is awesome")  # use docstring
4 @json                     # 3 decorators on one function = rethink!
5 def x():                  # Always name functions even if shadowed
6     return json_convert(1, 2)    # Keep separation
```

Example - fin.cache



```
1
2 class LuWhat?(object):
3
4     def __init__(self):
5         self._connection = None
6
7     def connection(self):
8         if self._connection is None:
9             self._connection = db_connect()
10        return self._connection
11
```

**Caching 'expensive'
values on objects**

**Actual code swamped
by infrastructure**

Look familiar?

Example - fin.cache

```
lulwhat.py
1
2 class LuWhat?(object):
3
4     def __init__(self):
5         self._connection = None
6
7     def connection(self):
8         if self._connection is None:
9             self._connection = db_connect()
10        return self._connection
11
```

```
clearer.py
1
2 class Awesome!(object):
3
4     @fin.cache.property
5     def connection(self):
6         return db_connect()
7
```

@fin.cache.property
documents intent

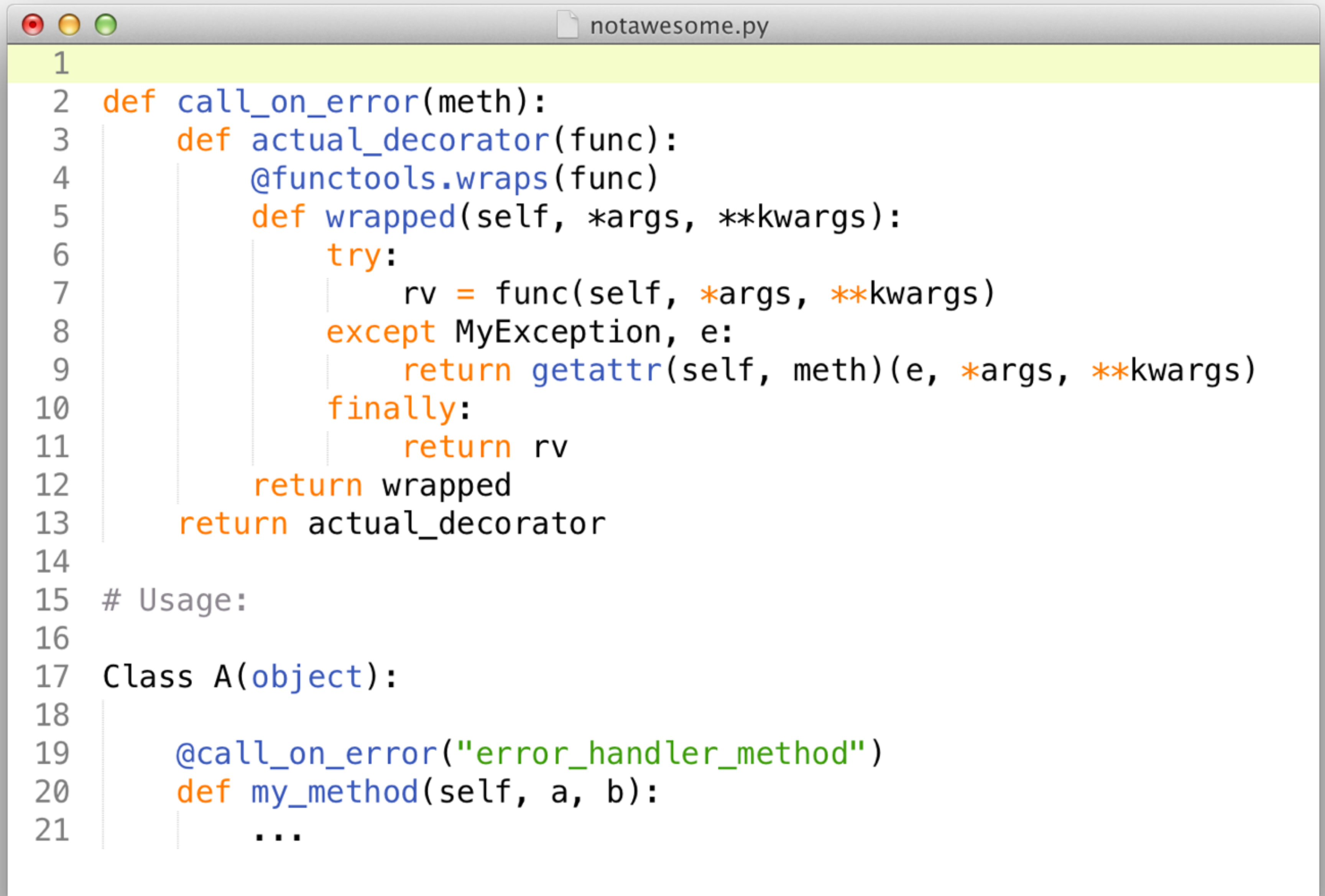
Decorators

Thanks for listening



@stestagg

Why they're Not Awesome



```
1
2 def call_on_error(meth):
3     def actual_decorator(func):
4         @functools.wraps(func)
5         def wrapped(self, *args, **kwargs):
6             try:
7                 rv = func(self, *args, **kwargs)
8             except MyException, e:
9                 return getattr(self, meth)(e, *args, **kwargs)
10            finally:
11                return rv
12        return wrapped
13    return actual_decorator
14
15 # Usage:
16
17 class A(object):
18
19     @call_on_error("error_handler_method")
20     def my_method(self, a, b):
21         ...
```

Why they're Not Awesome

First function just receives arguments

Second function receives/returns the actual function object

This function is what gets called with a.my_method()

A.my_method becomes this value

Original function gets called here

Aaaargh!

```
notawesome.py
1
2 def call_on_error(meth):
3     def actual_decorator(func):
4         @functools.wraps(func)  makes wrapped() 'look' like my_method
5         def wrapped(self, *args, **kwargs) in tracebacks etc..
6             try:
7                 rv = func(self, *args, **kwargs)
8             except MyException, e:
9                 return getattr(self, meth)(e, *args, **kwargs)
10            finally:
11                return rv
12        return wrapped
13    return actual_decorator
14
15 # Usage:
16
17 class A(object):
18
19     @call_on_error("error_handler_method")
20     def my_method(self, a, b):
21         ...
```