Decorators

And why they're a bit awesome

@stestagg

Syntax

Example1.py

Example1.py | Example2.py

```python
# Simple Example:


def decorator(func):
    return decorated

@decorator
def func(args):
    ...

# Equivalent to:

def func(args):
    pass
func = decorator(func)
```

Syntax

Example2.py

Example1.py | Example2.py

```python
# Less Simple Example:

def decorator(args):
    def actual_decorator(func):
        return decorated
    return actual_decorator

@decorator(args)
def func(other_args):
    return

# Equivalent to:

def func(other_args):
    pass
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

Three.py

```python

# Functions are objects

def is_awesome(func):
    func.is_awesome = True
    return func

@is_awesome
def my_func():
    return "awesome"

...
if getattr(func, "is_awesome", False):
    ...
```

2. Registration

Three.py

```
API_METHODS = {}

def api(func):
    API_METHODS[func.__name__] = func
    return func

@api
def my_func(a, b):
    return a + b

...

print API_METHODS["my_func"](1, 2)
```

3. Modification

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

four.py

```
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!**

bad.py

```
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

lulwhat.py

```
1
2  class LulWhat?(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 LulWhat?(object):
3
4      def __init__(self):
5          self._connection = None
6
7      def connection(sel
8          if self._conne
9              self._conn
10         return self._c
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

notawesome.py

```python

def call_on_error(meth):
    def actual_decorator(func):
        @functools.wraps(func)
        def wrapped(self, *args, **kwargs):
            try:
                rv = func(self, *args, **kwargs)
            except MyException, e:
                return getattr(self, meth)(e, *args, **kwargs)
            finally:
                return rv
        return wrapped
    return actual_decorator

# Usage:

Class A(object):

    @call_on_error("error_handler_method")
    def my_method(self, a, b):
        ...
```

Why they're Not Awesome

notawesome.py

```
def call_on_error(meth):
    def actual_decorator(func):
        @functools.wraps(func)
        def wrapped(self, *args, **kwargs):
            try:
                rv = func(self, *args, **kwargs)
            exceptMyException, e:
                return getattr(self, meth)(e, *args, **kwargs)
            finally:
                return rv
        return wrapped
    return actual_decorator

# Usage:

Class A(object):

    @call_on_error("error_handler_method")
    def my_method(self, a, b):
        ...
```

First function just receives arguments

Second function receives/returns the actual function object

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

makes wrapped() 'look' like my_method in tracebacks etc..

A.my_method becomes this value

Original function gets called here

Aaaargh!