Does this look familiar ?

```
>>> from functools import wraps
>>> def log_errors(func):
...     @wraps(func)
...     def log_errors_wrapper(*args, **kwargs):
...         try:
...             return func(*args, **kwargs)
...         except Exception as exc:
...             print("Raised %r for %s/%s" % (exc, args, kwargs))
...             raise
...     return log_errors_wrapper

>>> @log_errors
... def broken_function():
...     raise RuntimeError()
>>> from pytest import raises
>>> t = raises(RuntimeError, broken_function)
Raised RuntimeError() for ()/{}
```

Not very nice

- Having to write the wrapper boiler plate
    - Tendious
    - Same stuff every time
    - Confusing if you don't understand closures
    - Ever forgot to return the wrapper ?
- What if you use it on a generator ?

What if you use it on a generator ?

```
>>> @log_errors
... def broken_generator():
...     yield 1
...     raise RuntimeError()

>>> t = raises(RuntimeError, lambda: list(broken_generator()))
```

Dooh ! No output.

How to fix it ?

```
>>> from inspect import isgeneratorfunction
>>> def log_errors(func):
...     if isgeneratorfunction(func): # because you can't both return and yield in the same function
...         @wraps(func)
...         def log_errors_wrapper(*args, **kwargs):
...             try:
...                 for item in func(*args, **kwargs):
...                     yield item
...             except Exception as exc:
...                 print("Raised %r for %s/%s" % (exc, args, kwargs))
...                 raise
...     else:
...         @wraps(func)
...         def log_errors_wrapper(*args, **kwargs):
...             try:
...                 return func(*args, **kwargs)
...             except Exception as exc:
...                 print("Raised %r for %s/%s" % (exc, args, kwargs))
...                 raise
...     return log_errors_wrapper
```

Now it works:

```
>>> @log_errors
... def broken_generator():
...     yield 1
...     raise RuntimeError()

>>> t = raises(RuntimeError, list, broken_generator())
Raised RuntimeError() for ()/{}
```

The alternative, use `aspectlib`

```
>>> from aspectlib import Aspect
>>> @Aspect
... def log_errors(*args, **kwargs):
...     try:
...         yield
...     except Exception as exc:
...         print("Raised %r for %s/%s" % (exc, args, kwargs))
...         raise
```

Works as expected with generators:

```
>>> @log_errors
... def broken_generator():
...     yield 1
...     raise RuntimeError()
>>> t = raises(RuntimeError, lambda: list(broken_generator()))
Raised RuntimeError() for ()/{}

>>> @log_errors
... def broken_function():
...     raise RuntimeError()
>>> t = raises(RuntimeError, broken_function)
Raised RuntimeError() for ()/{}
```

`aspectlib`

- **This presentation**:

  https://github.com/ionelmc/python-aspectlib/tree/master/docs/presentations

  Generated using restview and converted to pdf using Google Chome (pagination by css)

- `aspectlib` **does many more things, check it out**:

  http://python-aspectlib.readthedocs.org/en/latest/