Stateful Decorators, And More

@history

```
def history(func):
    return_vals = set()
    def wrapper(*args, **kwargs):
        return_val = func(*args, **kwargs)
        return_vals.add(return_val)
        print('Return values: ' + str(sorted(return_vals)))
        return return_val
    return wrapper

@history
def foo(x):
    return x + 2

# Remember, same as:
def foo(x):
    return x + 2
foo = history(foo)
```

History

```
>>> print(foo(3))
Return values: [5]
5
>>> print(foo(2))
Return values: [4, 5]
4
>>> print(foo(3))
Return values: [4, 5]
5
>>> print(foo(7))
Return values: [4, 5, 9]
9
```

State in Decorators

```
def history(func):
    return_vals = set()
    def wrapper(*args, **kwargs):
        return_val = func(*args, **kwargs)
        return_vals.add(return_val)
        print('Return values: ' + str(sorted(return_vals)))
        return return_val
    return wrapper

@history
def foo(x):
    return x + 2

# Remember, same as:
def foo(x):
    return x + 2
foo = history(foo)
```

Revisiting

How can you automatically transform the first into the second?

```
# Original:
def get_items():
    return requests.get(API_URL + '/items')
# Automatically retrying version:
def get_items():
    MAX_TRIES = 3
    tries = 0
    resp = None
    while True:
        resp = requests.get(API_URL + '/items')
        if resp.status_code == 500 and tries < MAX_TRIES:
            tries += 1
            continue
        break
    return resp
```

@retry

```
def retry(func):
    def wrapper(*args, **kwargs):
        MAX_TRIES = 3
        tries = 0
        while True:
            resp = func(*args, **kwargs)
            if resp.status_code == 500 and tries < MAX_TRIES:
                tries += 1
                continue
            break
        return resp
    return wrapper

@retry
def get_items():
    return requests.get(API_URL + '/items')
```

Memoization

A function design pattern.

Given an expensive function `f`, you can cache its value.

```
def f(x, y, z):
    # do something expensive

cache = {}
def cached_f(x, y, z):
    # tuples can be dictionary keys.
    key = (x, y, z)
    if key not in cache:
        cache[key] = f(x, y, z)
    return cache[key]
```

This has been around for decades. It's still useful.

Lab: memoize

```
# Turn this:
cache = {}
def cached_f(x, y, z):
    # tuples can be dictionary keys.
    key = (x, y, z)
    if key not in cache:
        cache[key] = f(x, y, z)
    return cache[key]
# ... into this:
@memoize
def f(x, y, z):
   # ...
```

Lab file: `decorators/memoize.py`

- In `labs` folder
- When you are done, study the solution - compare to what you wrote.
- ... and then optionally do `decorators/memoize_extra.py`

HINT: In `memoize.py`, `wrapper` takes just `*args`, not `**kwargs`.