Decorators: Advanced: Takeaways

Syntax

PRESEVE METADATA FOR DECORATED FUNCTIONS

- Use functools.wraps() to make sure your decorated functions maintain their metadata:

```
from functools import wraps
def timer(func):
    """A decorator that prints how long a function took to run."""
    @wraps(func)
    def wrapper(*args, **kwargs):
        t_start = time.time()
        result = func(*args, **kwargs)
        t_total = time.time() - t_start
        print('{} took {}s'.format(func.__name__, t_total))
        return result
    return wrapper
```

ADD ARGUMENTS TO DECORATORS

- To add arguments to a decorator, turn it into a function that returns a decorator:

```
def timeout(n_seconds):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            # Set an alarm for n seconds
            signal.alarm(n_seconds)
            try:
                # Call the decorated func
                return func(*args, **kwargs)
            finally:
                # Cancel alarm
                signal.alarm(0)
        return wrapper
    return decorator
```

Concepts

- One of the problems with decorators is that they obscure the decorated function's metadata. The `wraps()` function from the `functools` module is a decorator that you use when defining a decorator. If you use it to decorate the wrapper function that your decorator returns, it will modify `wrapper()`'s metadata to look like the function you are decorating.

- To add arguments to a decorator, we have to turn it into a function that returns a decorator, rather than a function that is a decorator.

Resources

- The `functools` module