In [10]:
```
from functools import wraps
import time
```

In [11]:
```
def show_args(function):
    @wraps(function)
    def wrapper(*args, **kwargs):
        print('hi from decorator - args:')
        print(args)
        result = function(*args, **kwargs)
        print('hi again from decorator - kwargs:')
        print(kwargs)
        return result
    # return wrapper as a decorated function
    return wrapper
```

In [12]:
```
@show_args
def get_profile(name, active=True, *sports, **awards):
    print('\n\thi from the get_profile function\n')
```

In [13]:
```
get_profile('bob', True, 'basketball', 'soccer',
            pythonista='special honor of the community', topcoder='2017 code camp')
```

```
hi from decorator - args:
('bob', True, 'basketball', 'soccer')

        hi from the get_profile function

hi again from decorator - kwargs:
{'pythonista': 'special honor of the community', 'topcoder': '2017 code camp'}
```

Using @wraps

In [14]:
```
def timeit(func):
    '''Decorator to time a function'''
    @wraps(func)
    def wrapper(*args, **kwargs):

        # before calling the decorated function
        print('== starting timer')
        start = time.time()

        # call the decorated function
        func(*args, **kwargs)

        # after calling the decorated function
        end = time.time()
        print(f'== {func.__name__} took {int(end-start)} seconds to complete')

    return wrapper
```

In [17]:
```
@timeit
def generate_report():
    '''Function to generate revenue report'''
    time.sleep(2)
    print('(actual function) Done, report links ...')

generate_report()
```

```
== starting timer
(actual function) Done, report links ...
== generate_report took 2 seconds to complete
```

stacking decorators

In [18]:
```
def print_args(func):
    '''Decorator to print function arguments'''
    @wraps(func)
    def wrapper(*args, **kwargs):

        # before
        print()
        print('*** args:')
        for arg in args:
            print(f'- {arg}')

        print('**** kwargs:')
        for k, v in kwargs.items():
            print(f'- {k}: {v}')
        print()

        # call func
        func(*args, **kwargs)
    return wrapper
```

In [19]:
```
def generate_report(*months, **parameters):
    time.sleep(2)
    print('(actual function) Done, report links ...')
```

In [20]:
```
@timeit
@print_args
def generate_report(*months, **parameters):
    time.sleep(2)
    print('(actual function) Done, report links ...')
```

In [21]:
```
parameters = dict(split_geos=True, include_suborgs=False, tax_rate=33)
```

In [22]:
```
generate_report('October', 'November', 'December', **parameters)
```

```
== starting timer

*** args:
- October
- November
- December
**** kwargs:
- split_geos: True
- include_suborgs: False
- tax_rate: 33

(actual function) Done, report links ...
== generate_report took 2 seconds to complete
```

Passing arguments to a decorator

Another powerful capability of decs is the ability to pass arguments to them like normal functions, afterall they're functions too. Let's write a simple decorator to return a noun in a format:

In [24]:
```
def noun(i):
    def tag(func):
        def wrapper(name):
            return "My {0} is {1}".format(i, func(name))
        return wrapper
    return tag

@noun("name")
def say_something(something):
    return something

print(say_something('Ant'))

@noun("age")
def say_something(something):
    return something

print(say_something(44))
```

```
My name is Ant
My age is 44
```

In [ ]: