Python decorators are used to apply the same process to multiple functions.
This can be useful for developing a logging framework to help developing and debugging code.

Consider the following two kinds of logging:

- Printing out the name of a function and its arguments each time it runs.
- Printing out the amount of time a function takes to run.

In order to do this, you need to be able to do the following (in addition to creating closures and decorators)

- Get time information, using Python's time module
  - After importing you can call its time() method:
    - time.time() returns epoch time
- Pass arguments from a function variable (f) of an outer function (wrapper) to an inner function (inner)
  - ***arg** represents arguments passed to a function. Exempli gratia:

```
def wrapper( f ):
   def inner( *arg ):
       return f( *arg )
   return inner
closure = wrapper(foo)
closure( -2, 3, 'hello' )
```

  Output: The closure(-2,3,'hello') call will run foo with the arguments -2, 3, and 'hello' through wrapper. (Test drive this!)
- Get the name of a function
  - If f is a variable that represents a function, **f.func_name** will return the name of the function.

What would this all look like?
If you had a standard recursive fibonacci function, putting a name and argument logger (by creating the appropriate closure apparatus and using it as a decorator) on it would result in output like the following:

```
$ python fibtest.py
fib: (2,)
fib: (1,)
fib: (3,)
fib: (2,)
fib: (4,)
fib: (2,)
fib: (1,)
fib: (3,)
fib: (5,)
5
```

If you had a different function, and created an execution time logger, you might see output like this:

```
$ python timetest.py
execution time: 4.87653708458
```

Your mission: Write a closure/decorator apparatus that will facilitate timing, and another for output of function name and arguments, and use decorators to apply one or both of these to functions of your choosing. Store your work in **hw09** subdir of your hw repo.

For the last portion, you may find the following code snippet helpful. Predict the output of this code, then run…

```
#a simple example of applying multiple decorators
def make_bold(fn):
    return lambda : "<b>" + fn() + "</b>"

def make_italic(fn):
    return lambda : "<i>" + fn() + "</i>"

@make_bold
@make_italic
def hello():
    return "hello world"

helloHTML = hello()

print helloHTML
```