Functions as arguments and return values, decorators I

- Functions in Python can be passed as arguments and be returned as values from a function since they are value objects like other values.
- This can be used to create many programming idioms.
- *args, **kwargs allow variable number of arguments and keyword based arguments. The '*' is important, the names can be any legal Python name. args and kwargs are the names chosen by convention.
- The order is `function(required-arguments, *args, **kwargs)`.
- This can be used for defining useful special functions e.g. addN(b) - adds N to b.
- More generally it allows one to 'decorate' a function. (see code below) which allow functions to be modified using an annotation syntax but basically uses the fact that we can return functions as values.

Functions as arguments and return values, decorators II

```
def decoratorFn(fn):
    def wrap(<args>):
        'extension/modification'
        original function fn can be called with <args>
        at any stage
        'extension/modification'
    return wrap #the modified function is returned

@decoratorFn #this is the annotation
def actualFn(<args>): # function being decorated
    Actual function body
```

Python will do `actualFn=decoratorFn(actualFn)` at compile time. So, at run time any call to `actualFn` becomes a call to `wrap` with the same arguments as `actualFn`.
Note the order in which the definitions are done.

Functions as arguments and return values, decorators III

- A decorator_object is any **callable** object. It is returned when `decoratorFn(fn)` is executed. So after
  `actualFn=decorator_object`
  a call to `actualFn` becomes a call to the callable `decorator_object` which is a wrapper around the original `actualFn(<args>)`. In particular the `decoratorFn` need not be a function but can be a class which has the `__call__(self[,...])` attribute making it a callable object.

Property class and decorators

- In Python attributes are public so encapsulation is achieved by convention.
- But sometimes implementation or policy changes and can require code changes which will break the existing interface. One example is when a data attribute has restrictions placed on it so that any change in value must be checked for correctness. In this case a setter method is required. This will break all code that sets the attribute directly.
- The property class (it is often called a function) which uses the decorator idiom allows one to retain the old interface but get/set/delete is delegated to get/set/delete methods. Most often used for changing the interface to get/set.
- The property object is created by:
  `prop = property(getfn, setfn, delfn, docstring)`
  This will set the get, set and del operations for property/attribute `prop` to the respective functions in the argument list.

Property syntax

Both are equivalent.

```
class Cls(object):
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x

    def setx(self, v):
        self._x = v

    def delx(self):
        del self._x

    x = property(getx, setx,
        delx, "The 'x' property.")
```

```
class Cls(object):
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """The 'x' property."""
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```