11 Decorators

Key terms: `@ args kwargs inner()`

Reading: Real Python's Primer on Python Decorators[26]

Reading: Akshar's Understanding '*', '*args', '' and '**kwargs'**[27]

Exercise: Write a decorator that produces degree-based versions of the radian-based trigonometic functions (`math.sin`, etc.).

Wrapper functions, e.g., `wrapper(original())`, accommodate additional, optional functionality, and functions can have different wrappers for different purposes. Decorators[28] provide an abstracted and reusable kind of wrapping. Their rationale is that various functions can benefit from the same decoration behavior. If we only wanted that functionality in a single method, it would just be built there, but if it's useful more generally, the same decorator can apply.

A decorator is a function which returns a "decorated" version of some other function. In more formal terms, decoration is a persistent functional closure — a context in which functions are decorated with some useful behavior. Of course, behavior can be controlled by parameters passed to the decorating function. This is a way to systematize object-oriented method overriding. Decorators can also be autogenerated for different uses.

A decorator returns a reference to a new inline function which calls the function being decorated. In this way decorators allow us to change the form of data returned by a method, or do additional setup and breakdown work around it.

```
# Decorator that adds logging:

def add_logging(action):
 def inner(base):
  print ( action, base, end='' )
  return action(base)
 return inner
```

The same decorator may be useful to log, validate, budget, monitor, or profile any number of functions, which may themselves be variously decorated. We decorate a function either with the `@` "pie" sign or by reassigning its name. Both approaches have the same effect.

```
# Decorate cube() by enabling either of the ## commented lines:

##@add_logging
def cube(base):
 print(base**3)

##cube=add_logging(cube)

# Demonstrate the decorated function:
cube(11)
```

Since decorators need to pass on arguments meant for a decorated and wrapped function, they are defined using generalities. A tuple, `args`, and a dict, `kwargs`, contain all the positional and named parameters for a function call. The "splat" operators again expand iterables in place: `f(*args,**kwargs)` — some positional arguments and some keyword arguments — is the most general form of an arbittrary function call.

Like a generator, if a decorator is stateful, `nonlocal` is needed to access a persistent value in the outer scope.

[26] `https://realpython.com/blog/python/primer-on-python-decorators/`
[27] `https://www.agiliq.com/blog/2012/06/understanding-args-and-kwargs/`
[28] `https://www.python.org/dev/peps/pep-0318/`