LECTURE 35

ANONYMOUS FUNCTIONS AND DECORATORS

MCS 275 Spring 2023

Emily Dumas

LECTURE 35: VARIADIC FUNCTIONS AND DECORATORS

Reminders and announcements:

- Today is the deadline to receive approval on Project 4 custom (non-SQLite) topics.
- Homework 13 available.

FUNCTIONS

As you know, functions can be defined with the `def` keyword:

```
def f(x):
    "Compute the square of `x`"
    return x*x
```

After this definition, name `f` refers to a function object. Thus we've created a **function with a name**.

LAMBDA

In Python, you can create a function with no name—an **anonymous function**—using the syntax:

```
lambda x: x*x    # takes x, returns x*x
lambda x,y: x-y  # takes x and y, returns value x-y
```

`lambda` then evaluates to a function object, so the expression

```
lambda x,y: x-y
```

behaves just like the name

```
diff
```

if you previously defined

```
def diff(x,y):
    return x-y
```

WHEN TO USE LAMBDA

Functions definitely deserve names if they are used in several places, or if they are complicated.

But `lambda` is **good for simple functions used once**. Then, the definition and the only place of use are not separated.

COMMON USE FOR LAMBDA

The built-in functions `max`, `min`, and `list.sort` accept a keyword argument `key` that is a function which is applied to elements before making comparisons.

e.g. if `L` is a list of words, then `max(L, key=len)` is the longest word.

FUNCTION ARGUMENTS

Functions in Python can accept functions as arguments.

```
def dotwice(f):
    """Call function f twice"""
    f()
    f()
```

A better version works with functions that accept arguments:

```
def dotwice(f,*args,**kwargs):
    """Call function f twice (allowing arguments)"""
    f(*args,**kwargs)
    f(*args,**kwargs)
```

Here, `*args` means any number of positional arguments, and `**kwargs` means any number of keyword arguments.

RETURNING FUNCTIONS

Functions in Python can return functions. Often this is used to make "function factories".

```
def power_function(n):
    def inner(x): # function inside a function!
        """Raise x to a power"""
        return x**n
    return inner
```

MODIFYING FUNCTIONS

```
def return_twice_doer(f):
    """Return a new function which calls f twice"""
    def inner(*args,**kwargs):
        """Call a certain function twice"""
        f(*args,**kwargs)
        f(*args,**kwargs)
    return inner
```

REPLACING FUNCTIONS

In some cases we might want to replace an existing function with a modified version of it (e.g. as returned by some other function).

```
def g(x):
    """Print the argument with a message"""
    print("Function got value",x)

# actually, I wanted to always print that message twice!
g = return_twice_doer(g)
```

DECORATOR SYNTAX

There is a shorter syntax to replace a function with a modified version.

```
@modifier
def fn(x,y):
    """Function body goes here"""
```

is equivalent to

```
def fn(x,y):
    """Function body goes here"""

fn = modifier(fn)
```

The symbol `@modifier` (or any `@name`) before a function definition is called a **decorator**.

RETURNING VALUES

Usually, the inner function of a decorator should return the value of the (last) call to the argument function.

```
def return_twice_doer(f):
    """Return a new function which calls f twice"""
    def inner(*args,**kwargs):
        """Call a certain function twice"""
        f(*args,**kwargs)
        return f(*args,**kwargs)
    return inner
```

DECORATOR ARGUMENTS

Python allows `@name(arg1,arg2,...)`.

In that case, `name` should be a **decorator factory**.

E.g.

```
@dec(2)
def printsq(x):
    print(x*x)
```

is equivalent to

```
thisdec = dec(2)

@thisdec
def printsq(x):
    print(x*x)
```

Note a decorator factory is a function that returns a function that return a function!

A FEW BUILT-IN DECORATORS

- **`@functools.lru_cache(100)`** — Memoize up to 100 recent calls to a function.*
- **`@classmethod`** — Make a method a class method (callable from the class itself, gets class as first argument). E.g. for alternate constructors.
- **`@atexit.register`** — Make sure this function is called just before the program exits.

* In Python 3.9+ there is also the simpler `functools.cache` decorator which stores an unlimited number of past function calls..

MULTIPLE DECORATORS

Allowed. Each must be on its own line.

```
@dec1
@dec2
@dec3
def f(x):
    """Function body goes here"""
```

replaces `f` with `dec1(dec2(dec3(f)))`.

So the decorator closest to the function name acts first.

REFERENCES

- See Lutz, Chapter 18 for more about `*args` and `**kwargs`.
- See Lutz, Chapter 39 for a detailed discussion of Python decorators.
- See Beazley & Jones, Chapter 9 for several examples of decorators.

ACKNOWLEDGMENT

When preparing an earlier version of this lecture, I reviewed course materials by Danko Adrovic and Jan Verschelde (other MCS instructors).

REVISION HISTORY

- 2022-01-26 Last year's lecture on this topic finalized
- 2023-04-13 Updated for 2023