Advanced functions

Reuven M. Lerner, PhD
reuven@lerner.co.il

Functions are objects

```
def foo():
    return 5


>>> type(foo)
function
```

Aliasing functions

```
>>> x = foo


>>> x
<function __main__.foo>


>>> x()
5
```

Alias any function!

```
>>> import os

>>> x = os.listdir

>>> x('/tmp')

['.s.PGSQL.5432', '.s.PGSQL.5432.lock', '501',

 'launch-2HON6g','launchd-407.fNTzEI',
'mongodb-27017.sock', 'mysql.sock',
'wbxgpc.wbt']
```

Exploring functions

- If functions are objects, then:
  - Functions have a type
  - Functions have attributes
- We know the type… but what attributes do functions have?

Function attributes

```
>>> dir(foo)

['__call__', '__class__', '__closure__',
'__code__', '__defaults__', '__delattr__',
'__dict__', '__doc__', '__format__', '__get__',
'__getattribute__', '__globals__', '__hash__',
'__init__', '__module__', '__name__',
'__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', 'func_closure',
'func_code', 'func_defaults', 'func_dict',
'func_doc', 'func_globals', 'func_name']
```

What is special here?

- Let's look at some of the attributes that are special for functions
- These can give us insight into how Python works, and what it's doing in various places

__call__

- This is what Python invokes when you use parentheses!
- So you can say

```
foo()
```

- or

```
foo.__call__()
```

- and you'll get the same results

__defaults__

- A tuple (or None) containing default parameter values
- When a function is invoked without all of the parameter values assigned, these are used instead
- Also available via the "func_defaults" attribute

Mutable default gotcha

```
>>> def foo(mylist=[]):
      mylist.append('a')
>>> x = [1,2,3]
>>> foo(x)


>>> x
[1, 2, 3, 'a']
```

But...

```
>>> foo.__defaults__
([],)


>>> foo()
>>> foo.__defaults__
(['a'],)         # The default changed!


>>> foo()
>>> foo.__defaults__
(['a', 'a'],)    # The default changed again!
```

Oh, and don't do this

```
>>> foo.x = 5


>>> foo.x
5
```

func_code

- This is where most of the real goodies reside in a function
- Here are the attributes of a simple function's func_code:
- 'co_argcount', 'co_cellvars', 'co_code', 'co_consts', 'co_filename', 'co_firstlineno', 'co_flags', 'co_freevars', 'co_lnotab', 'co_name', 'co_names', 'co_nlocals', 'co_stacksize', 'co_varnames'

Greatest hits of func_code

- co_argcount — arity of the function (int)
- co_varnames — local variables (which is how LEGB works!)
- co_name — original name of the function when defined, and not affected by aliasing or removing the original name
- co_filename — name of the file on which it was defined
- co_firstlineno — line of the file in which the function definition started
- co_code — the bytecode of the function (more on this later)

Returning functions

- Functions are data
- We can return any type of data from a function
- Why not return a function?
- Yes, we can do this!

Inner functions

- We can define functions inside of functions
- The inner function, like all data in a function, exists only within the outer function's scope
- But of course, if you return the inner function, then you have a closure!

Inner functions

```
def foo():
    def bar():
        return 5
    return bar
x = foo()
x()    # 5
bar()  # error: not defined!
```

Scoping and inner functions

- Inner functions have access to variables in the outer functions
- That's the "E" ("enclosing") in the LEGB rule
- Variables local to the inner function have priority
- This allows a form of closure

Closure?

- An inner function has access to the outer function's variables, even after the outer function has finished executing
- There are certain techniques (especially functional ones) that benefit from closures

Inner functions

```
def multiplier(x):
    def mult_by(y):
        return x * y
    return mult_by
x = multiplier(5)
x(3)  # returns 15
```


20

Scoping and inner functions

- Python 2's scoping rules have an interesting quirk: An inner function cannot change the value of a variable in an outer function
- The inner function can read from that variable — but setting it merely creates a local variable

Example

```
def foo():
  x = 100
  def bar():
    x = 200
  bar()
  print(x)
foo()       # prints 100
```

global doesn't help

```
def foo():
  x = 100
  def bar():
    global x
    x = 200
  bar()
  print(x)
foo()       # still prints 100
```


23

Python 3: nonlocal

- Python 2 doesn't offer a solution to this
- Python 3, however, does: nonlocal
- The "nonlocal" keyword works like "global" in Python 2, but refers to variables in the enclosing function's scope

nonlocal use

```
def foo():
  x = 100
  def bar():
    nonlocal x
    x = 200
  bar()
  print(x)
foo()       # now prints 200
```

functools.partial

- We can accomplish much the same thing using functools.partial. We invoke it with the name of a function and the parameter(s) we want to pass.
- functools.partial then returns a function with the passed parameters already there.

Simple example

```
>>> from operator import mul

>>> p = partial(mul, 10)

>>> p(2)

20
```

```
>>> f = partial(int, base=2)

>>> f('100')

4
```

Function annotations

- In Python 3 (not 2!), you can add an "annotation" to one or more function parameters
- Each annotation is a value attached to a parameter name, and comes after :
- You can use annotations for type checking, but they are optional, not enforced by the language, and are never expected to be a part of the language

Example function annotations

- Here is an example use of annotations:

```
def foo(a:str='', b:int=0):

    print("Hello!")
```

- In the above example, we use types as the annotations. But we don't have to use types; we can use any Python object at all.

- We also don't need to set defaults

Storage of annotations

- They are stored in the __annotations__ attribute:

```
>>> foo.__annotations__
{'a': <class 'str'>, 'b': <class 'int'>}
```

- This is separate from the __defaults__ attribute:

```
>>> foo.__defaults__
('', 0)
```

Annotation of returns

- We can also annotate a function's return value!
- Again, this doesn't force us to return anything, but allows an external observer or program to know our stated intention.
- To do this, we follow our function's parameter list with an arrow (->), then the annotation (i.e., any object), and then the :
- Return annotations are available as "return"

Return annotation example

```
def foo(a:str='', b:int=0) -> int:

    print("Hello")



>>> foo.__annotations__

{'a': <class 'str'>, 'b': <class 'int'>,
'return': <class 'int'>}
```

Where to use them?

- Type checking
- Let IDEs show what types a function expects and returns
- Function overloading / generic functions
- Foreign-language bridges
- Adaptation
- Predicate logic functions
- Database query mapping
- RPC parameter marshaling
- Other information
- Documentation for parameters and return values