Function dissection lab

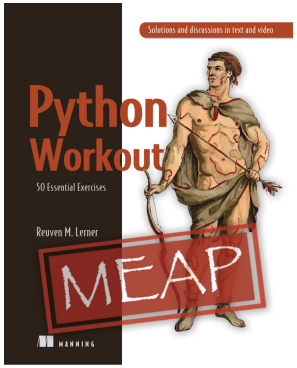
Reuven M. Lerner • PyCon 2020

reuven@lerner.co.il • @reuvenmlerner

I teach Python

- Corporate training
- Video courses about Python + Git
- Weekly Python Exercise
  - More info at https://lerner.co.il/

- "Python Workout" — published by Manning



- "Better developers" — free, weekly newsletter about Python
  - https://BetterDevelopersWeekly.com/

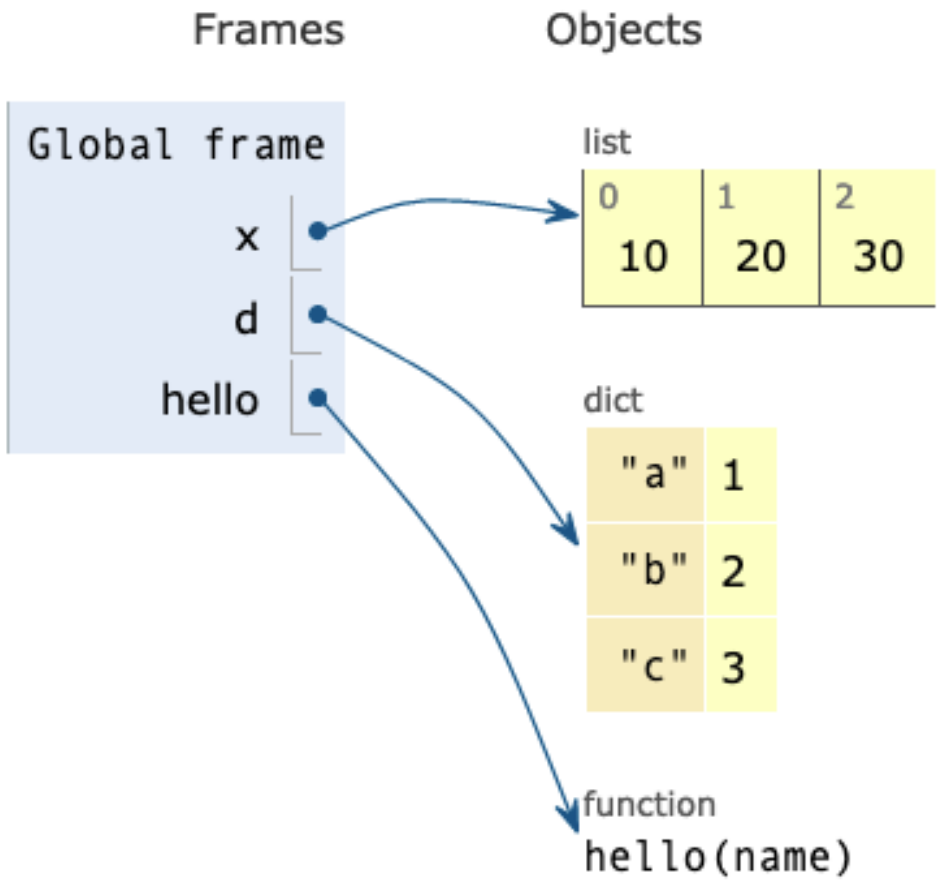
Let's write some code!

(And then we'll take it apart...)

Consider this code

```
x = [10, 20, 30]
d = {'a': 1, 'b': 2, 'c': 3}


def hello(name):
    return f'Hello, {name}!'
```


Frames
Objects
Global frame
x
d
hello
list
0
1
2
10
20
30
dict
"a"
1
"b"
2
"c"
3
function
hello(name)

Function objects?

- When you use “def”, you create a function object...
- ...and then you assign it to a variable

- Functions are nouns, not just verbs!
  - Functions (like all objects) can be assigned

```
hello2 = hello
```

  - Functions (like all objects) can be passed as arguments

```
hello(hello)
```

  - Functions (like all objects) have attributes

```
dir(hello)
```

How does Python use a function's attributes?

Example

```
>>> def hello(name):
        return f'Hello, {name}!'

>>> hello('world')
'Hello, world!'

>>> hello()
TypeError: hello() missing 1 required
positional argument: 'name'
```

How did Python know?

__code__

- The most important attribute in a function is __code__.
- Its attributes contain:
  - Python byte code
  - Hints to the Python interpreter about our function

__code__.co_argcount

- How many arguments does the function take?

```
>>> hello.__code__.co_argcount
1
```

So when we run this:

```
>>> def hello(name):
        return f'Hello, {name}!'

>>> hello('world')
'Hello, world!'
```

- Python says:
  - co_argcount say that we need 1 argument
  - The user passed 1 argument

But when we do this:

```
>>> def hello(name):
        return f'Hello, {name}!'

>>> hello()
TypeError: hello() missing 1 required
positional argument: 'name'
```

- Python says:
  - co_argcount say that we need 1 argument
  - We didn't pass any arguments — error!

__code__.co_varnames

- A tuple of strings
- Lists all of a function's local variables

```
>>> hello.__code__.co_varnames
('name',)
```

- So Python knows:
  - The function requires one argument
  - That argument will be assigned to "name"
  - If we get no arguments, then "name" is missing a value

Sure enough:

```
>>> def hello(name):
        return f'Hello, {name}!'

>>> hello()
TypeError: hello() missing 1 required
positional argument: ‘name'
```

Two parameters

```
>>> def hello(first, last):
        return f'Hello, {first} {last}!'


>>> hello.__code__.co_argcount
2

>>> hello.__code__.co_varnames
('first', ‘last')
```

Error messages use this info

```
>>> hello('Reuven')

TypeError: hello() missing 1 required positional
argument: 'last'


>>> hello(last='Lerner')

TypeError: hello() missing 1 required positional
argument: ‘first'


>>> hello('a', 'b', 'c')

TypeError: hello() takes 2 positional arguments but 3
were given
```

Additional variables

- What if we define a local variable, as well?

```
>>> def hello(first, last):
        s = f'Hello, {first} {last}!'
        return s

>>> hello.__code__.co_argcount
2

>>> hello.__code__.co_varnames
('first', 'last', 's')
```

- The first co_argcount elements of co_varnames are parameters

*args

```
>>> def hello(first, last, *args):
        return f'Hello {first} {last}, args = {args}'

>>> hello('a', 'b', 'c', 'd', 'e')
"Hello a b, args = ('c', 'd', 'e')"

>>> hello.__code__.co_argcount
2

>>> hello.__code__.co_varnames
('first', 'last', 'args')
```

So, how does Python know?

- This information is kept in co_flags, a int
- This int is the bitwise "and" of several bit flags
- If the bit is 1, then the flag is "on." Otherwise, it's off.

2^5 Generator	2^4 Nested	2^3 **kwargs	2^2 *args	2^1 New locals	2^0 Optimized

Always on

It's easier in hex, you know

```
co_optimized    0x01  # use fast locals
co_newlocals    0x02  # new dict for code block
co_varargs      0x04  # function has *args
co_varkeywords  0x08  # function has **kwargs
co_nested       0x10  # nested scopes
co_generator    0x20  # it's a generator function
```

Sure enough...

```
>>> hello.__code__.co_flags & 0x04    # Yes *args
4

>>> hello.__code__.co_flags & 0x08    # No **kwargs
0
```

Same goes for **kwargs

```
>>> def hello(**kwargs):
        return f'Hello, {kwargs}!'

>>> hello.__code__.co_flags & 0x04    # No *args
0

>>> hello.__code__.co_flags & 0x08    # Yes **kwargs
8
```

dis.show_code

```
>>> dis.show_code(hello)
Name:              hello
Filename:          <ipython-input-56-823d1147b5b6>
Argument count:    0
Kw-only arguments: 0
Number of locals:  1
Stack size:        3
Flags:             OPTIMIZED, NEWLOCALS, VARKEYWORDS, NOFREE
Constants:
   0: None
   1: 'Hello, '
   2: '!'
Variable names:
   0: kwargs
```

Constants?

- Literal values are stored in __code__.co_consts
- The first (zero-index) item in co_consts is always None
- Other constants (e.g., ints and strings) are also stored
  - Notice that f-strings are broken up into parts!
- The byte codes then refer to constants by index number

```
>>> hello.__code__.co_consts
(None, 'Hello, ', '!')
```

Bytecodes

- The function's byte codes are stored in co_code, as a bytestring:

```
>>> hello.__code__.co_code
b'd\x01|\x00\x9b\x00d\x02\x9d\x03S\x00'
```

- It's probably easier to understand with "dis.dis":

```
>>> dis.dis(hello)
  2           0 LOAD_CONST               1 ('Hello, ')
              2 LOAD_FAST                0 (kwargs)
              4 FORMAT_VALUE             0
              6 LOAD_CONST               2 ('!')
              8 BUILD_STRING             3
             10 RETURN_VALUE
```

What about defaults?

```
def hello(name='world'):
    return f'Hello, {name}'
```

```
>>> hello.__code__.co_argcount
 1
```

- It seems like our function works just like before
- But we know that we can call it with no arguments
- How does this work?

__defaults__

- A function's defaults are stored in __defaults__
- (Note: This is a function attribute, not a __code__ attribute!)

```
>>> hello.__defaults__
('world',)
```

- __defaults__ is always a tuple
- No defaults? Then it's an empty tuple

When Python calls a function...

- It compares the arguments with co_argcount
- Does the number match?
  - Pass arguments and call the function
- Not enough arguments?
  - Checks if __defaults__ can close the gap
  - If so, use enough from __defaults__ to get to co_argcount
- Too many arguments?
  - Check co_flags to see if *args is defined
  - If so, assign remaining arguments to *args
  - Or whatever variable is named in co_varnames[co_argcount]

Consider this function:

```
def add_one(x):
    x.append(1)


mylist = [10, 20, 30]

add_one(mylist)
print(mylist)          [10, 20, 30, 1]

add_one(mylist)
print(mylist)          [10, 20, 30, 1, 1]
```

Let's add a default

```
def add_one(x=[]):
    x.append(1)
    return x

print(add_one())          [1]
print(add_one())          [1, 1]
print(add_one())          [1, 1, 1]
```

The problem?

- __defaults__ is populated when you define the function

```
def add_one(x=[]):
    x.append(1)
    return x
```

- **Conclusion: Never use mutable defaults!**

Don't ignore this warning!

```
$ pylint add_one.py

************* Module add_one

add_one.py:4:0: W0102: Dangerous default value [] as
argument (dangerous-default-value)
```

Keyword-only arguments

```
def hello(*args, sep=' '):
    return f'Hello, {sep.join(args)}!'

>>> hello('a', 'b', 'c')
'Hello, a b c!'

>>> hello('a', 'b', 'c', sep='*')
'Hello, a*b*c!'
```

Where does Python put that?

- It isn't counted with the other arguments:

```
>>> hello.__code__.co_argcount
0
```

- Rather, it's listed here:

```
>>> hello.__code__.co_kwonlyargcount
1
```

Python checks in many places!

- co_argcount — number of mandatory, positional arguments
- __defaults__ — values that make co_argcount flexible
- co_flags
  - Do we assign extra positional args to *args?
  - Do we assign extra keyword args to **kwargs?
- co_kwonlyargcount — number of keyword-only args

Let's talk about scoping

```
x = 100

def func():
    print(f'In func, x = {x}')

print(f'Before, x = {x}')     # 100
func()                        # 100
print(f'After, x = {x}')      # 100
```

- L — Local
- E — Enclosing
- G — Global
- B — Builtins

How does Python know x isn't local?

```
def func():
    print(f'In func, x = {x}')
```

- It checks in the attributes, of course:

```
>>> func.__code__.co_varnames
()
```

- Since "x" isn't in co_varnames, it isn't a local variable.

Let's make things more complex

```
x = 100

def func():
    x = 200
    print(f'In func, x = {x}')

print(f'Before, x = {x}')     # 100
func()                        # 200
print(f'After, x = {x}')      # 100
```

- L — Local
- E — Enclosing
- G — Global
- B — Builtins

How does Python know x is local?

- Because it's in co_varnames

```
>>> func.__code__.co_varnames
('x',)
```

- Notice: co_varnames is populated at compile time, not runtime!

Let's make a slight change...

```
x = 100

def func():
    print(f'In func, x = {x}')
    x = 200


print(f'Before, x = {x}')
func()
print(f'After, x = {x}')
```

Swapped these lines

What happens now?

```
Before, x = 100
Traceback (most recent call last):
  File "./func12.py", line 12, in <module>
    func()
  File "./func12.py", line 7, in func
    print(f'In func, x = {x}')
UnboundLocalError: local variable 'x' referenced before
assignment
```

Huh?

- Consider our function:

```
def func():
    print(f'In func, x = {x}')
    x = 200
```

- Because we assign to x in the function, x is local
  - x is thus in __code__.co_varnames
- When we run the function, we need x's value for the "print"
- Python knows that x is local, but has no local value...

UnboundLocalError

A more common version of this problem

```
x = 100

def func():
    x += 1
    print(f'In func, x = {x}')


print(f'Before, x = {x}')
func()
print(f'After, x = {x}')
```

Same as

$x = x + 1$

The "global" declaration

```
x = 100


def func():
    global x
    x = 200
    print(f'In func, x = {x}')


print(f'Before, x = {x}')      # 100
func()                         # 200
print(f'After, x = {x}')       # 200
```

What does "global" do?

- It removes a variable from co_varnames:

```
>>> func.__code__.co_varnames
()
```

- Python uses LEGB to look for "x"
- It cannot find "x" in the tuple of local variable names
- So it assigns to the global variable x!

Bytecodes with a local "x"

```
>>> dis.dis(func)
  3           0 LOAD_CONST               1 (200)
              2 STORE_FAST               0 (x)

  4           4 LOAD_GLOBAL              0 (print)
              6 LOAD_CONST               2 ('In func, x = ')
              8 LOAD_FAST                0 (x)
             10 FORMAT_VALUE             0
             12 BUILD_STRING             2
             14 CALL_FUNCTION            1
             16 POP_TOP
             18 LOAD_CONST               0 (None)
             20 RETURN_VALUE
```

Bytecodes with a global "x"

```
>>> dis.dis(func)
  5           0 LOAD_CONST               1 (200)
              2 STORE_GLOBAL             0 (x)

  6           4 LOAD_GLOBAL              1 (print)
              6 LOAD_CONST               2 ('In func, x = ')
              8 LOAD_GLOBAL              0 (x)
             10 FORMAT_VALUE             0
             12 BUILD_STRING             2
             14 CALL_FUNCTION            1
             16 POP_TOP
             18 LOAD_CONST               0 (None)
             20 RETURN_VALUE
```

Putting the "E" in LEGB

```
def outer():
    run_counter = 0
    total = 0

    def inner(x):
        run_counter += 1
        total += x
        print(f'Run {run_counter}, total is {total}')

    return inner


func = outer()
for i in range(10, 100, 10):
    func(i)
```

UnboundLocalError

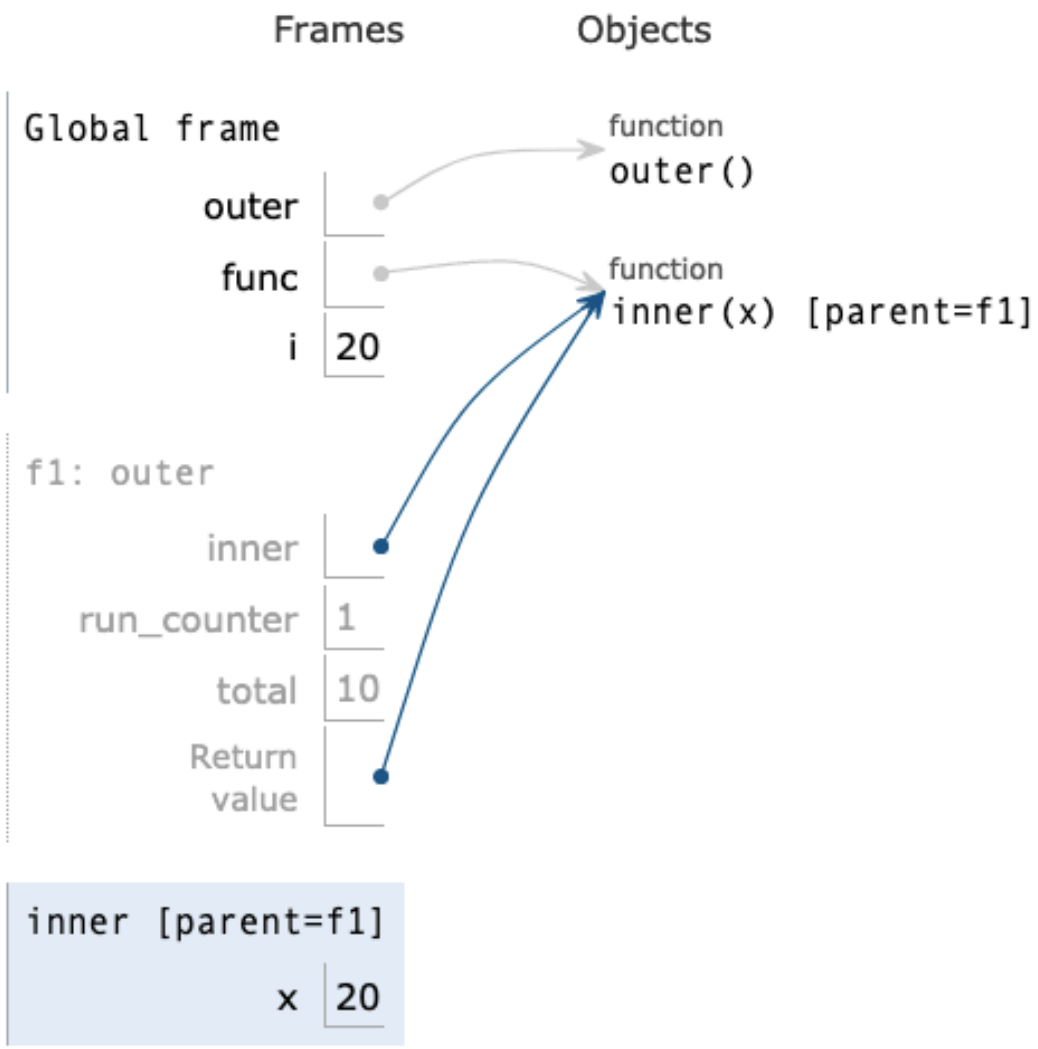
Make them … nonlocal

```
def outer():
    run_counter = 0
    total = 0

    def inner(x):
        nonlocal run_counter, total
        run_counter += 1
        total += x
        print(f'Run {run_counter}, total is {total}')

    return inner


func = outer()
for i in range(10, 100, 10):
    func(i)
```


Frames
Objects
Global frame
outer
func
i
20
function
outer()
function
inner(x) [parent=f1]
f1: outer
inner
run_counter
1
total
10
Return value
inner [parent=f1]
x
20

How?

```
>>> func = outer()
>>> func.__code__.co_freevars
('run_counter', ‘total')
```

- But that’s not all!
- “outer” knows which of its local variables are referenced:

```
>>> outer.__code__.co_cellvars
('run_counter', 'total')
```

What have we learned?

- “def” does two things
  - Creates a function object
  - Assigns that function object to a variable
- Function objects contain attributes
  - Byte codes
  - Hints to Python for running the function
- Attributes dictate behavior we often take for granted
  - Argument assignment
  - Scoping

Questions or comments?

- E-mail me: reuven@lerner.co.il
- Follow me on Twitter: @reuvenmlerner

- See you in Pittsburgh in 2021!