6.009: Fundamentals of Programming

Lecture 1: Functions

- Review of Functions
- Functions as First-Class Objects
- Closures
- Partial Application

Adam Hartz

`hz@mit.edu`

7 September 2020

6.009: Goals

Our goals involve helping you develop your programming skills, in multiple aspects:

- **Programming:** analyzing problems, developing plans
- **Coding:** translating plans into Python
- **Debugging:** developing test cases, verifying correctness, finding and fixing errors

So we will spend time discussing (and practicing!):

- high-level design strategies
- ways to manage complexity
- details and ”goodies” of Python
- a mental model of Python's operation
- testing and debugging strategies

The Power of Abstraction

Thinking about complicated systems can be *complicated*.

The Power of Abstraction

Thinking about complicated systems can be *complicated*.

Thinking about simpler systems is often simpler.

The Power of Abstraction

Thinking about complicated systems can be *complicated*.

Thinking about simpler systems is often simpler.

Framework for thinking about complicated systems ("PCAP"):

- **Primitives**
- Means of **Combination**
- Means of **Abstraction**
- Meaningful **Patterns**

The Power of Abstraction

Thinking about complicated systems can be *complicated*.

Thinking about simpler systems is often simpler.

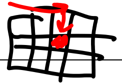
Framework for thinking about complicated systems ("PCAP"):

- **Primitives**
- Means of **Combination**
- Means of **Abstraction**
- Meaningful **Patterns**

Example (operations in Python):

- Primitives: `+`, `*`, `==`, `!=`, ...
- Combination: `if`, `while`, `f(g(x))`, ...
- Abstraction: `def`

Building Abstractions: Example



An example from lab 1: working with pixel values as a flat list in row-major order is a pain! How can we make our life easier? A couple of examples:

{width, height, pixel: [- - - - - - - - - - -]}

- define helper functions for working with the existing structure, for example:

```
def flat_index(image, x, y):
    return image['width'] * y + x
```

- define helper functions for converting to/from a more convenient representation, for example:

```
def to_2d_array(im):
    h, w, p = im['height'], im['width'], im['pixels']
    return [[p[r*w + c] for c in range(w)] for r in range(h)]

def from_2d_array(arr):
    return {'height': len(arr), 'width': len(arr[0]),
            'pixels': sum(arr, [])}
```

Review: Functions in Environment Model

Function definition with `def`:

1. Creates a new function object in memory. In our simplified model, this object contains:
   - The names of the formal parameters of the function
   - The code in the body of the function
   - A pointer back to the frame in which we were running when this object was created.
2. Associates that function object with a name

Note that the body of the function is not evaluated at definition time!

Review: Functions in Environment Model

Function application (”calling” or ”invoking”, with round brackets):

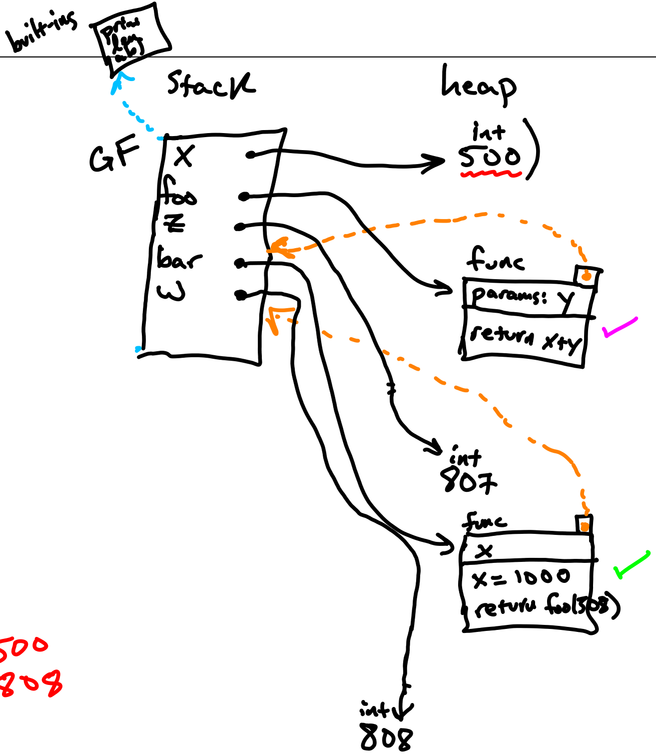
1. Evaluate the function to be called, followed by its arguments (in order)
2. Create a new frame for the function call, with a parent frame determined by the function we’re calling
3. Bind the parameters of the function to the given arguments in this new frame
4. Execute the body of the function in this new frame

Example

```
x = 500
def foo(y):
    return x+y
z = foo(307)
print('x:', x)
print('foo:', foo)
print('z:', z)


def bar(x):
    x = 1000
    return foo(308)
w = bar('hello')
print()
print('x:', x)
print('w:', w)
```

500
808



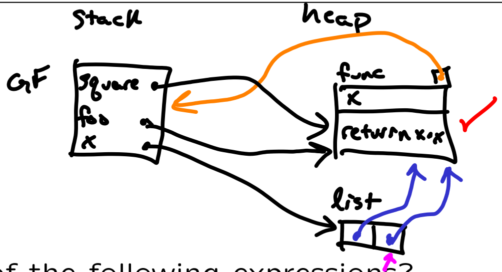
Functions are First-class Objects

Like most (but not all) modern programming languages, functions in Python are **first-class objects**, meaning that they are treated precisely the same way as other primitive types we've seen. Among other things, functions:

- can be the subject of assignment statements
- can be included in collections (lists, dictionaries, etc)
- can be the arguments to other functions
- can be returned as the results of other functions

Small Example

```
def square(x):
    return x*x
foo = square
x = [square, foo]
```



What is the **type** and **value** of each of the following expressions?

- square(2) → int 4
- foo(0.7) → float 0.49
- foo → function
- x → list [<func obj ....>, <func obj ...>]
- foo[1] → type ERROR ~~function object~~ is not subscriptable
- x[0] → function
- square() → type ERROR, expected 1 argument, got 0
- x[1](3.1) → float
- (square + foo)(7) → ERROR

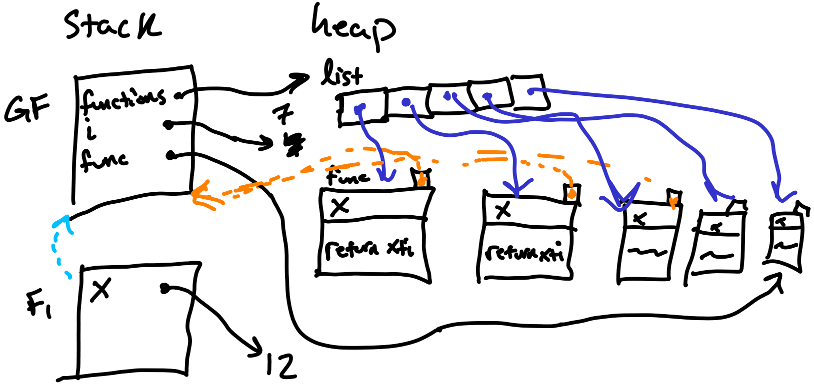
Explaining Last Lecture's Mystery

Surprisingly, the following piece of code printed the number 16 five times when we ran it:

```
functions = []
for i in range(5):
    def func(x):
        return x + i
    functions.append(func)

for f in functions:
    print(f(12))
```

16
16
16
16
16



This is perhaps surprising! But we can explain this behavior using an environment diagram (and the rules we developed for function definition/application above).

i=7

19
19
19
19
19

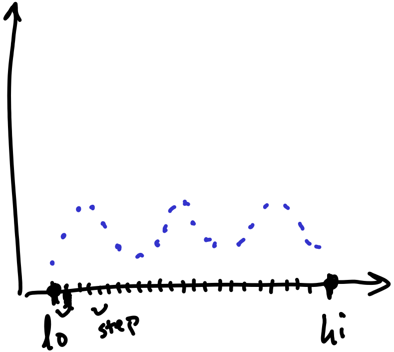
Examples of Using Functions as First-class Objects

Plotting the response of a function using matplotlib:

```
import math
import matplotlib.pyplot as plt

def sine_response(lo, hi, step):
    xs = []
    ys = []
    cur = lo
    while cur <= hi:
        xs.append(cur)
        ys.append(math.sin(cur))
        cur += step
    plt.plot(xs, ys)
```

current



Examples of Using Functions as First-class Objects

Plotting the response of a function using matplotlib:

```
import math
import matplotlib.pyplot as plt

def cosine_response(lo, hi, step):
    xs = []
    ys = []
    cur = lo
    while cur <= hi:
        xs.append(cur)
        ys.append(math.cos(cur))
        cur += step
    plt.plot(xs, ys)
```

Examples of Using Functions as First-class Objects

Plotting the response of a function using matplotlib:

```
import math
import matplotlib.pyplot as plt

def square_response(lo, hi, step):
    xs = []
    ys = []
    cur = lo
    while cur <= hi:
        xs.append(cur)
        ys.append(cur**2)
        cur += step
    plt.plot(xs, ys)
```

Examples of Using Functions as First-class Objects

That's a lot of repeated code! Let's rewrite it in a way that makes life a little bit easier if we want to plot a bunch of different functions within the same program.

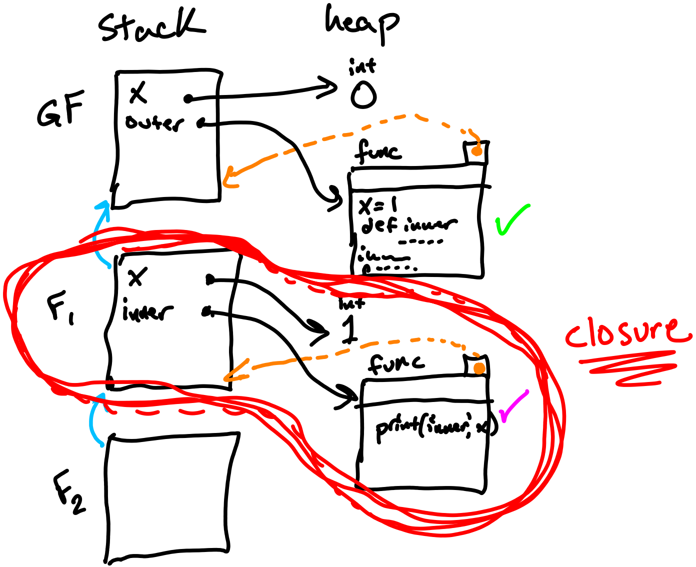
Functions within Functions

It turns out that we can also define functions inside of other functions! Let's think about what the following piece of code does:

```
x = 0

def outer():
    x = 1
    def inner():
        print('inner:', x)
    inner()
    print('outer:', x)

print('global:', x) →0
outer()
inner()
print('global:', x)
```



Closures

Importantly, a function definition "remembers" the frame in which it was defined, so that later, when the function is being called, it has access to the variables defined in that "enclosing" frame.

We call this combination (of a function and its enclosing frame) a **closure**, and it turns out to be a really useful structure. For example:

```
def add_n(n):
    def inner(x):
        return x + n
    return inner
```

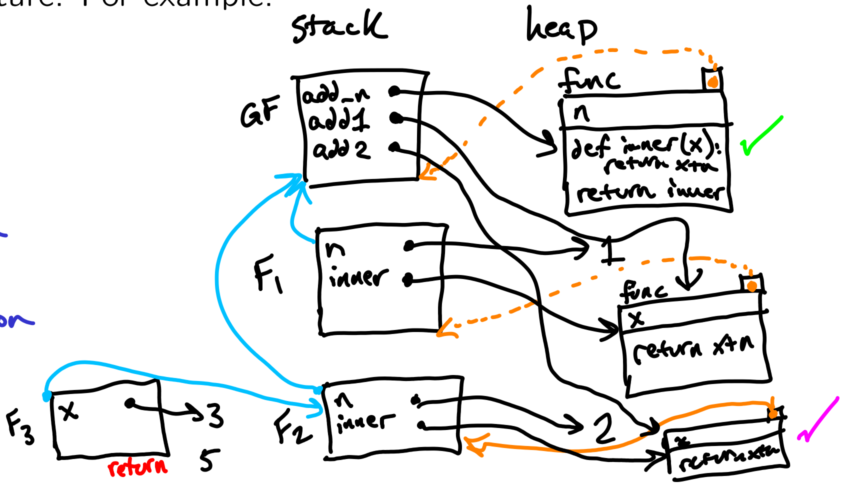
```
add1 = add_n(1)
```
→ function

```
add2 = add_n(2)
```
→ function

```
print(add2(3))
```
→ 5

```
print(add1(7))
```
→ 8

```
print(add_n(8)(9))
```
→ 17



Example: Derivatives

Let's take a look at a small piece of code that computes the derivative of an arbitrary function.

$$f(x)$$

$$f'(x) \approx \frac{f(x+\partial x) - f(x-\partial x)}{2 \cdot \partial x}$$

Example: Partial Application

Let's take a look at an interesting use of closures: *partial application*