6.009: Fundamentals of Programming

Lecture 1: Functions

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

Adam Hartz
hz@mit.edu

13 September 2021

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

6.009: Pedagogy

Learning to program is a lot like learning a musical instrument or a sport.

How does one learn those things?

6.009: Pedagogy

Learning to program is a lot like learning a musical instrument or a sport.
How does one learn those things?

Just like with music/sports, **deliberate practice** is key!

To improve as a programmer, it helps to:

- watch how experienced programmers approach problems
- program!
- receive feedback from more experienced programmers

6.009 aims to provide you with *lots* of opportunities for all of these

6.009: Pedagogy

Learning to program is a lot like learning a musical instrument or a sport.
How does one learn those things?

Just like with music/sports, **deliberate practice** is key!

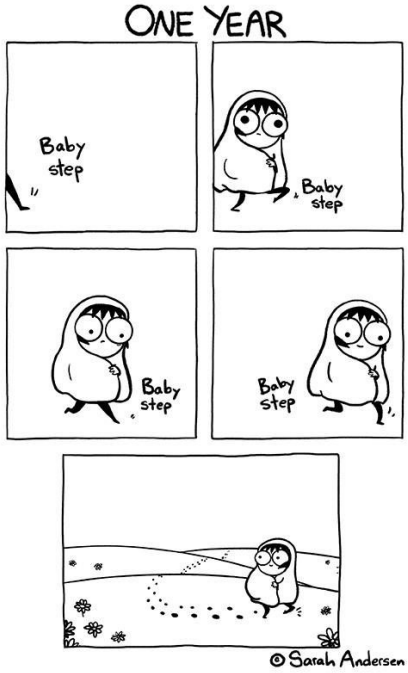
To improve as a programmer, it helps to:

- watch how experienced programmers approach problems
- program!
- receive feedback from more experienced programmers

6.009 aims to provide you with *lots* of opportunities for all of these

- Labs give opportunities to practice new techniques/skills to solve interesting problems.
- Lectures/recitations equip you with tools useful for attacking those problems.
- Checkoffs and office hours give opportunities to receive expert feedback.

Growth, not Perfection



Getting the Most Out of 6.009

Getting the Most Out of 6.009

Lectures/Recitations:

- Step 1: Come to lecture/recitation, and participate!
- Take notes *in your own words* and review them later
- **Ask questions!** We want to have a conversation.

Getting the Most Out of 6.009

Lectures/Recitations:

- Step 1: Come to lecture/recitation, and participate!
- Take notes *in your own words* and review them later
- **Ask questions!** We want to have a conversation.

Labs:

- Start early (labs are week-long assignments)
- Formulate a plan before writing code
  - Try to understand the problem thoroughly before writing code
  - When things go wrong, step away from the code and revisit the plan
- Work through problems on your own
- Ask for help when you need it!
  - Labs are intentionally challenging
  - Bugs are a natural part of life
  - Lots of opportunities for help (office hours / forum)
- Be mindful of our policies about academic integrity

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.

What tools do we have for combining simple ideas into more complex ideas?

- **Primitives**
- Means of **Combination**
- Means of **Abstraction**

The Power of Abstraction

Thinking about complicated systems can be *complicated*.
Thinking about simpler systems is often simpler.

What tools do we have for combining simple ideas into more complex ideas?

- **Primitives**
- Means of **Combination**
- Means of **Abstraction**

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:

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

```
def flat_index(image, x, y):
    # return the proper index into the pixel array
```

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

```
def to_2d_array(im):
    # return an equivalent list-of-lists


def from_2d_array(arr):
    # given a list-of-lists, return an equivalent image
```

Check Yourself!

What happens when the following program is run?

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

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

1. It prints 12, then 13, then ..., then 16
2. It prints 13, then 14, then ..., then 17
3. It prints 16, then 15, then ..., then 12
4. It prints 17, then 16, then ..., then 13
5. A Python error occurs
6. Something else

Environment Diagrams

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 reference to the function's enclosing frame.
2. Associates that function object with a name

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

```
def double(x):
    return x * 2
```

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 (parent frame is the function's enclosing frame)
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

```
z = double(4)
```

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
z = [square, foo]
```

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

- `square(2)`
- `foo(0.7)`
- `foo`
- `x`
- `z`
- `foo[1]`
- `z[0]`
- `square()`
- `z[1](3.1)`
- `(square + foo)(7)`

Explaining the Earlier Mystery

The output of this piece of code may have been surprising:

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

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

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

Another Example of Functions as First-class Objects

Plotting the response of a function using matplotlib:

```
import math
import matplotlib.pyplot as plt

def plot_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)
```

Closures

It turns out that we can also define functions inside of other functions! 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)
add2 = add_n(2)

print(add2(3))
print(add1(7))
print(add_n(8)(9))
```

Example: Derivatives

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

Fixing the Mystery

The output of this piece of code may have been surprising:

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

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

Summary

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