61A Lecture 5

Friday, January 30

Announcements

- Quiz 1 scores will be posted eventually, but you already know what you'll get
  - 0/3: Please talk to your TA for advice on how to proceed
  - 1/3: Make sure to spend time understanding all lab & discussion questions
  - 2/3: Practice is extremely helpful in learning how to solve CS problems
- Guerrilla Section 1 on higher-order functions is on Saturday 1/31 in 271 Soda
  - Optional discussion to promote mastery of core concepts (prepares you for midterms)
  - 2pm – 4pm is the vanguard section (you commit to helping teach the main section)
  - 4pm – 6pm is the main section
  - Please do not bring questions about homework or projects to guerrilla sections
- Small-group tutoring begins next week! Apply online by Sunday if you want a (free) tutor
- Homework 2 (which is small) is due Monday 2/2 at 11:59pm
- Project 1 (which is BIG) us due Thursday 2/5 at 11:59pm
- Midterm 1 on Monday 2/9 7pm-9pm
  - Conflict? Fill out the conflict form today! http://goo.gl/2P5fKq

Environments for Higher-Order Functions

Environments Enable Higher-Order Functions

Functions are first-class: Functions are values in our programming language

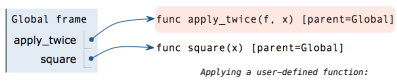
Higher-order function: A function that takes a function as an argument value **or**
A function that returns a function as a return value

Environment diagrams describe how higher-order functions work!

(Demo)

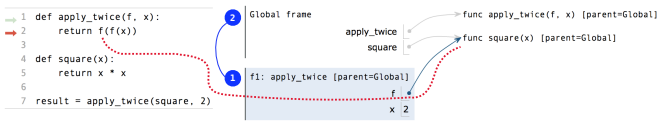
Names can be Bound to Functional Arguments

```
1 def apply_twice(f, x):
2     return f(f(x))
3
4 def square(x):
5     return x * x
6
7 result = apply_twice(square, 2)
```



Applying a user-defined function:

- Create a new frame
- Bind formal parameters (f & x) to arguments
- Execute the body: return f(f(x))



Interactive Diagram

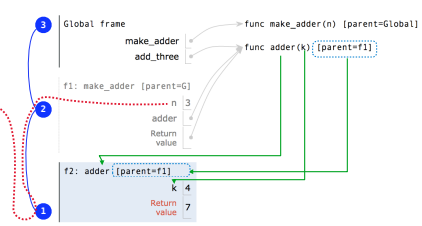
Environments for Nested Definitions

(Demo)

Environment Diagrams for Nested Def Statements

```
1 def make_adder(n):
2     def adder(k):
3         return k + n
4     return adder
5
6 add_three = make_adder(3)
7 add_three(4)
```

Nested def



- Every user-defined function has a parent frame (often global)
- The parent of a function is the frame in which it was defined
- Every local frame has a parent frame (often global)
- The parent of a frame is the parent of the function called

Interactive Diagram

How to Draw an Environment Diagram

When a function is defined:

Create a function value: func <name>(<formal parameters>) [parent=<label>]

Its parent is the current frame.

f1: make_adder func adder(k) [parent=f1]

Bind <name> to the function value in the current frame

When a function is called:

1. Add a local frame, titled with the <name> of the function being called.
2. Copy the parent of the function to the local frame: [parent=<label>]
3. Bind the <formal parameters> to the arguments in the local frame.
4. Execute the body of the function in the environment that starts with the local frame.

Local Names

(Demo)

Local Names are not Visible to Other (Non-Nested) Functions

```
1  def f(x, y):
2      return g(x)
3
4  def g(a):
5      return a + y
6
7  result = f(1, 2)
```

"y" is not found, again

Error

"y" is not found

Global frame

f → func f(x, y) [parent=Global]

g → func g(a) [parent=Global]

f1: f [parent=Global]

x 1

y 2

f2: g [parent=Global]

a 1

- An environment is a sequence of frames.
- The environment created by calling a top-level function (no def within def) consists of one local frame, followed by the global frame.

Interactive Diagram

Function Composition

(Demo)

The Environment Diagram for Function Composition

```
1  def square(x):
2      return x * x
3
4  def make_adder(n):
5      def adder(k):
6          return k + n
7      return adder
8
9  def compose1(f, g):
10     def h(x):
11         return f(g(x))
12     return h
13
14 compose1(square, make_adder(2))(3)
```

Return value of make_adder is an argument to compose1

Global frame

square → func square(x) [parent=Global]

make_adder → func make_adder(n) [parent=Global]

compose1 → func compose1(f, g) [parent=Global]

f1: make_adder [parent=Global]

n 2

adder → func adder(k) [parent=f1]

Return value → func h(x) [parent=f2]

f2: compose1 [parent=Global]

f → func square(x) [parent=Global]

g → func make_adder(n) [parent=Global]

h → func h(x) [parent=f2]

Return value

f3: h [parent=f2]

x 3

f4: adder [parent=f1]

k 3

Interactive Diagram

Lambda Expressions

(Demo)

Lambda Expressions

```
>>> x = 10
```

An expression: this one evaluates to a number

```
>>> square = x * x
```

Also an expression: evaluates to a function

```
>>> square = lambda x: x * x
```

Important: No "return" keyword!

A function
with formal parameter x
that returns the value of "x * x"

```
>>> square(4)
16
```

Must be a single expression

Lambda expressions are not common in Python, but important in general

Lambda expressions in Python cannot contain statements at all!

Lambda Expressions Versus Def Statements

```
square = lambda x: x * x
```

VS

```
def square(x):
    return x * x
```

- Both create a function with the same domain, range, and behavior.
- Both functions have as their parent the frame in which they were defined.
- Both bind that function to the name square.
- Only the def statement gives the function an intrinsic name.

Global frame

square → func λ(x) <1line 1> [parent=Global]

f1: λ <1line 1> [parent=Global]

x 4

Return value 16

The Greek letter lambda

Global frame

square → func square(x) [parent=Global]

f1: square [parent=Global]

x 4

Return value 16

Currying

Function Currying

```
def make_adder(n):
    return lambda k: n + k
```

```
>>> make_adder(2)(3)
5
>>> add(2, 3)
5
```

There's a general relationship between these functions

(Demo)

Curry: Transform a multi-argument function into a single-argument, higher-order function