Environments
Announcements
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

Applying a user-defined function:
- Create a new frame
- Bind formal parameters \((f & x)\) to arguments
- Execute the body: \(\text{return } f(f(x))\)

```python
def apply_twice(f, x):
    return f(f(x))
def square(x):
    return x * x
result = apply_twice(square, 2)
```
Environments for Nested Definitions
Environment Diagrams for Nested Def Statements

- 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

```python
def make_adder(n):
    def adder(k):
        return k + n
    return adder

add_three = make_adder(3)
add_three(4)
```
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.

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

- 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.
Function Composition

(Demo)
Return value of `make_adder` is an argument to `compose1`
Lambda Expressions

(Demo)
Lambda Expressions

```python
>>> 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
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

\[
\text{square} = \lambda x: x \times x \quad \text{VS} \quad \text{def square}(x): \\
\quad \text{return } x \times x
\]

- Both create a function with the same domain, range, and behavior.
- Both bind that function to the name `square`.
- Only the `def` statement gives the function an intrinsic name, which shows up in environment diagrams but doesn't affect execution (unless the function is printed).
Currying
Function Currying

```python
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

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