Environments
Announcements
Environments for Higher-Order Functions
Environments Enable Higher-Order Functions
Environments Enable Higher-Order Functions

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

```python
def apply_twice(f, x):
    return f(f(x))

def square(x):
    return x * x

result = apply_twice(square, 2)
```
Names can be Bound to Functional Arguments

```python
def apply_twice(f, x):
    return f(f(x))

def square(x):
    return x * x

result = apply_twice(square, 2)
```
Names can be Bound to Functional Arguments

```python
def apply_twice(f, x):
    return f(f(x))

def square(x):
    return x * x

result = apply_twice(square, 2)
```

Applying a user-defined function:
- Create a new frame
- Bind formal parameters \((f \ & \ x)\) to arguments
- Execute the body:
  
  ```python
  return f(f(x))
  ```
Names can be Bound to Functional Arguments

```python
def apply_twice(f, x):
    return f(f(x))

apply_twice = apply_twice

def square(x):
    return x * x

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))
  ```
Names can be Bound to Functional Arguments

1. def apply_twice(f, x):
   2.     return f(f(x))

4. def square(x):
   5.     return x * x

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))
Names can be Bound to Functional Arguments

```python
def apply_twice(f, x):
    return f(f(x))

def square(x):
    return x * x

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))
Environments for Nested Definitions

(Demo)
Environment Diagrams for Nested Def Statements

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

add_three = make_adder(3)
add_three(4)
```
def make_adder(n):
    def adder(k):
        return k + n
    return adder

add_three = make_adder(3)
add_three(4)
Environment Diagrams for Nested Def Statements

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

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

Diagram showing the environment for nested def statements.
Environment Diagrams for Nested Def Statements

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

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

![Diagram showing the environment of nested def statements]

http://pythontutor.com/composingprograms.html#code=def%20make_adder%28n%29%3A%0A%20%20%20%20def%20adder%28k%29%3A%0A%20%20%20%20%20%20%20%20return%20k%20%2B%20n%0A%20%20%20%20return%20adder%0A%0A%20%20%20%20add_three%20=%20make_adder%283%29%0A%20%20%20%20add_three%284%29&cumulative=false&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D
def make_adder(n):
    def adder(k):
        return k + n
    return adder

add_three = make_adder(3)
add_three(4)
Environment Diagrams for Nested Def Statements

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

add_three = make_adder(3)
add_three(4)
```
Environment Diagrams for Nested Def Statements

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

add_three = make_adder(3)
add_three(4)
Environment Diagrams for Nested Def Statements

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

add_three = make_adder(3)
add_three(4)
```
Every user-defined function has a parent frame (often global).
Every user-defined function has a parent frame (often global)
The parent of a function is the frame in which it was defined
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)
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
How to Draw an Environment Diagram

When a function is defined:
How to Draw an Environment Diagram

When a function is defined:

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

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.
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]
```
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
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:
How to Draw an Environment Diagram

When a function is defined:
Create a function value: \( \text{func } <\text{name}>(<\text{formal parameters}>) \ [\text{parent}=<\text{label}>] \)
Its parent is the current frame.

\[
f1: \text{make_adder} \quad \text{func adder}(k) \ [\text{parent}=f1]
\]

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.
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>]

```python
f1: make_adder  func adder(k) [parent=f1]
```
How to Draw an Environment Diagram

When a function is defined:

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

Its parent is the current frame.

\[ \text{f1: make_adder} \quad \text{func adder(k) [parent=f1]} \]

Bind \text{name} to the function value in the current frame

When a function is called:

1. Add a local frame, titled with the \text{name} of the function being called.

\[ \star \text{2. Copy the parent of the function to the local frame: [parent=<label>]} \]

3. Bind the \text{<formal parameters>} to the arguments in the local frame.
How to Draw an Environment Diagram

When a function is defined:

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

Its parent is the current frame.

Bind \texttt{<name>} to the function value in the current frame

When a function is called:

1. Add a local frame, titled with the \texttt{<name>} of the function being called.

2. Copy the parent of the function to the local frame: \texttt{[parent=<label>]}

3. Bind the \texttt{<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

```python
# Global frame
func f(x, y) [parent=Global]

# Local frame

def f(x, y):
    return g(x)

def g(a):
    return a + y

result = f(1, 2)
```
Local Names are not Visible to Other (Non-Nested) Functions

```
def f(x, y):
    return g(x)

def g(a):
    return a + y

result = f(1, 2)
```
Local Names are not Visible to Other (Non-Nested) Functions

```python
def f(x, y):
    return g(x)

def g(a):
    return a + y

result = f(1, 2)
```
Local Names are not Visible to Other (Non-Nested) Functions

```
http://pythontutor.com/composingprograms.html?code=
def f(x, y):
    return g(x)
def g(a):
    return a + y
result = f(1, 2)
```

“y” is not found
Local Names are not Visible to Other (Non-Nested) Functions

```
def f(x, y):
    return g(x)
def g(a):
    return a + y
result = f(1, 2)
```

"y" is not found, again

"y" is not found
Local Names are not Visible to Other (Non-Nested) Functions

```
def f(x, y):
    return g(x)
def g(a):
    return a + y
result = f(1, 2)
```

"y" is not found, again

"y" is not found, again

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

```
def f(x, y):
    return g(x)

def g(a):
    return a + y

result = f(1, 2)
```

• An environment is a sequence of frames.
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.

```python
def f(x, y):
    return g(x)

def g(a):
    return a + y

result = f(1, 2)
```
Function Composition

(Demo)
The Environment Diagram for Function Composition

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

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

def compose1(f, g):
    def h(x):
        return f(g(x))
    return h

compose1(square, make_adder(2))(3)
```
The Environment Diagram for Function Composition

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

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

def compose1(f, g):
    def h(x):
        return f(g(x))
    return h

compose1(square, make_adder(2))(3)
```
The Environment Diagram for Function Composition

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

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

def compose1(f, g):
    def h(x):
        return f(g(x))
    return h

compose1(square, make_adder(2))(3)
```
def square(x):
    return x * x

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

def compose1(f, g):
    def h(x):
        return f(g(x))
    return h

compose1(square, make_adder(2))(3)
The Environment Diagram for Function Composition

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

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

def compose1(f, g):
    def h(x):
        return f(g(x))
    return h

compose1(square, make_adder(2))(3)
```

Return value of make_adder is an argument to compose1
The Environment Diagram for Function Composition

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

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

def compose1(f, g):
    def h(x):
        return f(g(x))
    return h

compose1(square, make_adder(2))(3)
```

Return value of make_adder is an argument to compose1
def square(x):
    return x * x

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

def compose1(f, g):
    def h(x):
        return f(g(x))
    return h

compose1(square, make_adder(2))(3)

Return value of make_adder is an argument to compose1
The Environment Diagram for Function Composition

Return value of make_adder is an argument to compose1
The Environment Diagram for Function Composition

1. def square(x):
   return x * x
2. def make_adder(n):
   def adder(k):
     return k + n
   return adder
3. def compose1(f, g):
   def h(x):
     return f(g(x))
   return h
4. compose1(square, make_adder(2))(3)

Return value of make_adder is an argument to compose1
The Environment Diagram for Function Composition

1. def square(x):
   return x * x
2. def make_adder(n):
   def adder(k):
       return k + n
   return adder
3. def compose1(f, g):
   def h(x):
       return f(g(x))
   return h
4. compose1(square, make_adder(2))(3)

Return value of make_adder is an argument to compose1
Return value of make_adder is an argument to compose1
Lambda Expressions

(Demo)
Lambda Expressions
>>> x = 10
Lambda Expressions

```python
>>> x = 10

>>> square = x * x
```
Lambda Expressions

```python
>>> x = 10

>>> square = x * x

An expression: this one evaluates to a number
```
Lambda Expressions

```python
>>> x = 10

>>> square = x * x

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

An expression: this one evaluates to a number

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

A function

   with formal parameter x
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
A function
   with formal parameter x
   that returns the value of "x * x"
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
Important: No "return" keyword!
A function
with formal parameter x
that returns the value of "x * x"
```
Lambda Expressions

```python
>>> x = 10
>>> square = x * x
>>> square = lambda x: x * x
```

An expression: this one evaluates to a number

Also an expression: evaluates to a function

A function

with formal parameter x

that returns the value of "x * x"

Important: No "return" keyword!

Must be a single expression
Lambda Expressions

```python
>>> x = 10
>>> square = x * x
>>> square = lambda x: x * x
>>> square(4)
16
```

An expression: this one evaluates to a number

Also an expression: evaluates to a function

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

Important: No "return" keyword!

Must be a single expression
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
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

```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
Lambda Expressions Versus Def Statements

VS
Lambda Expressions Versus Def Statements

\[
\text{square} = \lambda x: x \times x
\]

VS
Lambda Expressions Versus Def Statements

\[ \text{square} = \text{lambda } x: x \times x \quad \text{VS} \quad \text{def square}(x): \text{return } x \times x \]
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.
Lambda Expressions Versus Def Statements

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

- Both create a function with the same domain, range, and behavior.
- Both bind that function to the name square.
Lambda Expressions Versus Def Statements

\[
square = \lambda x: x \ast x
\]
\[
\text{def square(x): return } x \ast x
\]

- 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).
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).
Lambda Expressions Versus Def Statements

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

- 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).
Lambda Expressions Versus Def Statements

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

- 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).
Lambda Expressions Versus Def Statements

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

- 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).
Lambda Expressions Versus Def Statements

\[
square = \text{lambda } x: x \times x
\]

\[
def \text{square}(x):
    \text{return } x \times x
\]

- Both create a function with the same domain, range, and behavior.
- Both bind that function to the name \text{square}.
- Only the \text{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
Function Currying

```python
def make_adder(n):
    return lambda k: n + k
```
**Function Currying**

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

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

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

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

There's a general relationship between these functions.
Function Currying

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

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

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