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
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)
```
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

```
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:
  
  ```python
  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)
```
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))
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)
```
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)
```
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 frames 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)
```

---

The diagram shows the environment for nested `def` statements, illustrating how the `make_adder` function creates a nested `adder` function, which itself returns a new function `add_three`. The diagram also visualizes the value flow and how the `adder` function is called with `k = 4`, resulting in a return value of 7.
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 frames and call stack 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)
```
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.
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)

```
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).

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.
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:  \texttt{func }<\texttt{name}>\langle<\texttt{formal parameters}>\rangle\texttt{ [parent=\langle\texttt{label}\rangle]}\texttt{ }

Its parent is the current frame.
How to Draw an Environment Diagram

When a function is defined:

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

Its parent is the current frame.

\[ f_1: \text{make_adder} \quad \text{func } \text{adder}(k) \ [\text{parent}=f_1] \]
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.

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

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

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:    \texttt{func <name>(<formal parameters>) [parent=<label>]} 
Its parent is the current frame.

\texttt{f1: make_adder} \quad \texttt{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.
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>]`
How to Draw an Environment Diagram

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

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

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

When a function is called:
1. Add a local frame, titled with the \langle \text{name}\rangle of the function being called.
2. Copy the parent of the function to the local frame: \[\text{parent}=\langle \text{label}\rangle\]
3. Bind the \langle \text{formal parameters}\rangle 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.

\texttt{f1: make_adder} \hspace{1cm} \texttt{func adder(k) [parent=f1]} 

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.

\star 2. Copy the parent of the function to the local frame: [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
def f(x, y):
    return g(x)
def g(a):
    return a + y
result = f(1, 2)
```

### Global Frame

- **f**: Parent = Global
  - `x`: 1
  - `y`: 2

- **g**: Parent = Global
  - `a`: 1

---

# Python Tutor Diagram

- **f**: Parent = Global
  - `x`: 1
  - `y`: 2

- **g**: Parent = Global
  - `a`: 1
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

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

def g(a):
    return a + y

result = f(1, 2)
```

Diagram showing the visibility of local names within and between functions.
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)
```

- `f(x, y)` refers to `g(a)` which is defined in the outer scope.
- `g(a)` is defined in the local scope of `f(x, y)`.
- `y` is not found in the outer scope of `f(x, y)`.

In Python, local names are not visible to other non-nested functions.
Local Names are not Visible to Other (Non-Nested) Functions

```python
# Local Names are not Visible to Other Functions

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

def g(a):
    return a + y

result = f(1, 2)
```

This code demonstrates that the variable `y` is not visible in the outer scope of the function `f`. When `f` is called with `f(1, 2)`, it attempts to access the local variable `y` from the inner function `g`, which is not defined in the outer scope of `f`. Therefore, Python raises a `NameError` stating that the name `y` is not defined again.
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

"y" is not found, again

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

- An environment is a sequence of frames.

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

"y" is not found, again

Error

HTTP://PYTHONTUTORS.COM/COMPOSINGPROGRAMS.HTML#CODE=DEFF(X, Y)%0A%0A    RETURN G(X)%0A%0ADEF G(A)%0A%0A    RETURN A + Y%0ARESULT = F(1, 2)
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)
```

![Diagram showing variable lookup in nested functions](http://pythontutor.com/composingprograms.html?mode=display&origin=composingprograms.js&code=def%20f%28x%2C%20y%29%3A%0A%20%20%20%20return%20g%28x%29%0A%0Adef%20g%28a%29%3A%0A%20%20%20%20return%20a%20%2B%20y%0A%20%20%20%20%0Aresult%20%3D%20f%281%2C%202%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D)
Function Composition

(Demo)
The Environment Diagram for Function Composition

```python
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)
```
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

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

```
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

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

3  def make_adder(n):
4      def adder(k):
5          return k + n
6      return adder

7  def compose1(f, g):
8      def h(x):
9          return f(g(x))
10     return h

11 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

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

3 def make_adder(n):
4     def adder(k):
5         return k + n
6     return adder

7 def compose1(f, g):
8     def h(x):
9         return f(g(x))
10    return h

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

Return value of make_adder is an argument to compose1

```
Global frame

square

make_adder

compose1

func square(x) [parent=Global]

func make_adder(n) [parent=Global]

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

f1: make_adder [parent=Global]

add n

Return value

f2: compose1 [parent=Global]

f g h

Return value

f3: h [parent=f1]

x 3

f4: adder [parent=f1]

k 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)

Return value of make_adder is an argument to compose1
Self-Reference

(Demo)
Returning a Function Using Its Own Name

```python
def print_sums(n):
    print(n)
def next_sum(k):
    return print_sums(n+k)
return next_sum

print_sums(1)(3)(5)
```

Diagram showing the execution flow of the function calls.