Announcements
Office Hours: You Should Go!
Office Hours: You Should Go!

You are not alone!
Office Hours: You Should Go!

You are not alone!
Office Hours: You Should Go!

You are not alone!

http://cs61a.org/office-hours.html
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

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

```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

Applying a user-defined function:

- Create a new frame
- Bind formal parameters \((f \ & \ x)\) to arguments
- Execute the body:
  \[
  \text{return } f(f(x))
  \]
Names can be Bound to Functional Arguments

1 def apply_twice(f, x):
   return f(f(x))
2
4 def square(x):
   return x * x
5
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))

```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:
  return \(f(f(x))\)
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 \(f(f(x))\)
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)
```

Global frame

- `func make_adder(n) [parent=Global]`
- `func adder(k) [parent=f1]`
- `f1: make_adder [parent=G]`
  - `n = 3`
  - `adder`
  - `Return value`

- `f2: adder [parent=f1]`
  - `k = 4`
  - `Return value`
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

```
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

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

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

2. add_three = make_adder(3)

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

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

2. return adder

3. add_three = make_adder(3)

4. 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:   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: \( \text{func } \langle \text{name} \rangle(\langle \text{formal parameters} \rangle) [\text{parent}=\langle \text{label} \rangle] \)

Its parent is the current frame.

\[
\begin{align*}
\text{f1: make_adder} & \quad \text{func adder}(k) [\text{parent}=\text{f1}]
\end{align*}
\]
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:  `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.
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>]
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.
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.

```
func make_adder
```

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

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

def g(a):
    return a + y

result = f(1, 2)
```

**Global frame**

- `func f(x, y) [parent=Global]`
- `func g(a) [parent=Global]`

**Local scope**

- `f1: f [parent=Global]`
  - `x 1`
  - `y 2`

- `f2: 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)
```

Diagram:
- `func f(x, y) [parent=Global]`
- `func g(a) [parent=Global]`
- `f1: f [parent=Global]`
  - `x 1`
  - `y 2`
- `f2: 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)
```

![Diagram of Python function execution and scope](image)
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
Local Names are not Visible to Other (Non-Nested) Functions

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

```
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)
```

"y" is not found

"y" is not found, again

>Error

Global frame

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

func g(a) [parent=Global]

f1: f [parent=Global]
    x 1

f2: g [parent=Global]
    a 1

y 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)
```

- An environment is a sequence of frames.

"y" is not found again

"y" is not found

Error

- The def g(a): function is not defined within the f(x, y) function.
- The variable "y" is not visible outside of the g(a) function.
- The code will not compile and will throw an error.

Diagram:
- The global frame contains the variables x, y, and a.
- The f(x, y) function calls g(a) which returns a - y.
- The result of f(1, 2) is incorrect due to the local names not being visible.
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)
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
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

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
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
Return value of make_adder is an argument to compose1

```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)

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
The Environment Diagram for Function Composition

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

(Demo)
Returning a Function by 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)
```
Lambda Expressions

(Demo)
Lambda Expressions
Lambda Expressions

```python
>>> x = 10
```
Lambda Expressions

```python
>>> x = 10

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

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

An expression: this one evaluates to a number

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

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

An expression: this one evaluates to a number

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

```python
>>> x = 10
An expression: this one evaluates to a number

>>> square = x * x;

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

1. >>> x = 10
2. An expression: this one evaluates to a number
3. >>> square = x * x;
4. >>> 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
```
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

```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
```
Lambda Expressions

```python
>>> x = 10

An expression: this one evaluates to a number

>>> square = x * x

Also an expression: evaluates to a function

A function

    with formal parameter x

    that returns the value of "x * x"

>>> 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
A function

Important: No "return" keyword!

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
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

>>> 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

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

VS
Lambda Expressions Versus Def Statements

\[
square = \lambda x: x \times x
\]  VS
Lambda Expressions Versus Def Statements

\[
square = \lambda x: x \times x
\]

vs

\[
def \text{square}(x):
    \text{return } x \times x
\]
Lambda Expressions Versus Def Statements

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

- Both create a function with the same domain, range, and behavior.
Lambda Expressions Versus Def Statements

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

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

- Both create a function with the same domain, range, and behavior.
- Both functions have as their parent the frame in which they were defined.
Lambda Expressions Versus Def Statements

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

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

- Both create a function with the same domain, range, and behavior.
- Both functions have as their parent the frame in which they were defined.
- Both bind that function to the name square.
Lambda Expressions Versus Def Statements

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

- 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.
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 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.
Lambda Expressions Versus Def Statements

\[
square = \lambda x: x \times x
\]

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

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

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

- 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.
Lambda Expressions Versus Def Statements

\[
square = \lambda x: x \times x
\]

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

• 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.
Lambda Expressions Versus Def Statements

\begin{verbatim}
square = lambda x: x * x  
def square(x):
    return x * x
\end{verbatim}

- 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.