Multiple Environments
Life Cycle of a User-Defined Function

**Def statement:**
- `square(x):`
- `return mul(x, x)`

**Call expression:**
- `square(2+2)`
- `operand: 2+2 argument: 4`

**Calling/Applying:**
- `4 → square(x):`
- `return value: 16`

**What happens?**
- A new function is created!
- Name bound to that function in the current frame
- Operator & operands evaluated
- Function (value of operator) called on arguments (values of operands)
- A new frame is created!
- Parameters bound to arguments
- Body is executed in that new environment
Multiple Environments in One Diagram!

```python
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(square(3))
```

Interactive Diagram
Multiple Environments in One Diagram!

1. from operator import mul
2. def square(x):
3.     return mul(x, x)
4. square(square(3))

Global frame

mul
square

func mul(...)  
func square(x) [parent=Global]

f1: square [parent=Global]

<table>
<thead>
<tr>
<th>x</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return value</td>
<td>9</td>
</tr>
</tbody>
</table>

Interactive Diagram
Multiple Environments in One Diagram!

An environment is a sequence of frames.

- The global frame alone
- A local, then the global frame

Interactive Diagram
Names Have No Meaning Without Environments

Every expression is evaluated in the context of an environment.

A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

An environment is a sequence of frames.

- The global frame alone
- A local, then the global frame

Interactive Diagram
Names Have Different Meanings in Different Environments

A call expression and the body of the function being called are evaluated in different environments.

Every expression is evaluated in the context of an environment.

A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.
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

```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

- 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.

```
    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

- 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.
Lambda Expressions

(Demo)
Lambda Expressions

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

An expression: this one evaluates to a number

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

Also an expression: evaluates to a function

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

Important: No "return" keyword!

A function

```
with formal parameter x
```

that returns the value of "x * x"

```python
>>> 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

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

- 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).
Self-Reference

(Demo)
Returning a Function Using Its Own Name

```python
1  def print_sums(n):
2      print(n)
3  def next_sum(k):
4      return print_sums(n+k)
5  return next_sum
6
7  print_sums(1)(3)(5)
```
Review
What Would Python Display?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul

def square(x):
    return mul(x, x)
```

A function that takes any argument and returns a function that returns that arg

```
def delay(arg):
    print('delayed')
    def g():
        return arg
    return g
```

Names in nested def statements can refer to their enclosing scope

<table>
<thead>
<tr>
<th>This expression</th>
<th>Evaluates to</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>print(5)</td>
<td>None</td>
<td>5</td>
</tr>
<tr>
<td>print(print(5))</td>
<td>None</td>
<td>5 None</td>
</tr>
<tr>
<td>delay(delay)()()</td>
<td>6</td>
<td>delayed delayed 6</td>
</tr>
<tr>
<td>delay(delay)()()</td>
<td>None</td>
<td>delayed delayed 6</td>
</tr>
<tr>
<td>print(delay(print)()()4))</td>
<td>None</td>
<td>delayed 4 None</td>
</tr>
</tbody>
</table>
What Would Python Print?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul

def square(x):
    return mul(x, x)

def pirate(arggg):
    print('matey')

def plunder(arggg):
    return arggg
```

<table>
<thead>
<tr>
<th>This expression</th>
<th>Evaluates to</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add(pirate(3)(square)(4), 1)</code></td>
<td>17</td>
<td>Matey 17</td>
</tr>
<tr>
<td><code>func square(x)</code></td>
<td>16</td>
<td></td>
</tr>
<tr>
<td><code>pirate(pirate(pirate))(5)(7)</code></td>
<td>Error</td>
<td>Matey Error</td>
</tr>
<tr>
<td><code>Identity function</code></td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.
```python
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)
horse(mask)
```