Environments
Class outline:

- Multiple environments
- Environments for HOFs
- Local names
- Function composition
- Currying
Multiple Environments
## Life cycle of a function

### What happens?

<table>
<thead>
<tr>
<th>Def statement</th>
<th>What happens?</th>
</tr>
</thead>
</table>
| `def square(x):` | - A new function is created!  
- Name bound to that function in the current frame. |
| `return x * x` | |

<table>
<thead>
<tr>
<th>Call expression</th>
<th>What happens?</th>
</tr>
</thead>
</table>
| `square(2 + 2)` | - Operator & operands evaluated  
- Function (value of operator) called on arguments (values of operands) |

<table>
<thead>
<tr>
<th>Calling/applying</th>
<th>What happens?</th>
</tr>
</thead>
</table>
| `def square(x)` | - A new frame is created!  
- Parameters bound to arguments  
- Body is executed in that new environment |
| |

```python
def square(x):
    return x * x
```
A nested call expression

1.
2.
3.

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

square(square(3))
```
A nested call expression

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

square(square(3))
```

next
A nested call expression

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

square(square(3))
```

Global frame

- `square [func] square(x) [parent=Global]`
def square(x):
    return x * x

square(square(square(3)))
square( square(3) )
A nested call expression

1. 
2. 
3. 

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

square(square(3))
```

Global frame

```
square [ func square(x) [parent=Global] ]
```
func square(x)
square( square(3) )
A nested call expression

def square(x):
    return x * x

square(square(square(3)))
func square(x)

square(square(3))

square(3)
A nested call expression

def square(x):
    return x * x

square(square(square(3)))

Global frame
   square [•] --> func square(x) [parent=Global]
\[ \text{func } \text{square}(x) \]

\[ \text{square}(\text{square}(3)) \]

\[ \text{func } \text{square}(x) \]

\[ \text{square}(3) \]
A nested call expression

1. 
2. 
3. 

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

square(square(3))
```

Global frame

```
square [func square(x) [parent=Global]]
```
\texttt{\textbf{func} \textbf{square}(x)}

\texttt{\textbf{square}(3)}

\texttt{\textbf{func} \textbf{square}(x) \ 3}
A nested call expression

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

square(square(square(3)))
```
func square(x)

square(3)

square( square(3) )
A nested call expression

1. def square(x):
   return x * x

2. square(square(3))

Global frame

  square —> func square(x) [parent=Global]

f1: square [parent=Global]
    x | 3
func square(x)

square(3)

square(square(3))
A nested call expression

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

square(square(3))
```
func square(x)

{{
    func square(x)  3
}}

square(square(3))
A nested call expression

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

square(square(3))
```

Global frame

```
square [•] ----> func square(x) [parent=Global]
```

f1: square [parent=Global]

```
x [3]
Return value [9]
```
\[
\text{func } \text{square}(x) \quad 3
\]

\[
square( \text{square}(3) )
\]
A nested call expression

1. def square(x):
   return x * x

2. square(square(3))

Global frame

f1: square [parent=Global]

x

Return value 9
\text{func square}(x) \quad 3

\text{square}( \text{square}(3) )

\text{square}(3)
A nested call expression

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

square(square(3))
```
$$\text{func square}(x)$$

$$\text{square}(\text{square}(3))$$

$$\text{func square}(x)$$

$$\text{square}(3)$$

$$\text{func square}(x)$$

$$3$$
A nested call expression

1. 
2. 
3. 

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

square(square(3))
```
\[
\text{func square}(x) \quad 3 \\
\text{square}(3) \\
\text{square}(\text{square}(3)) \\
\]

Return value 81
Multiple environments in one diagram!

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

square(square(square(3)))
```

Global frame

```
square [func] square(x) [parent=Global]
```

f1: square [parent=Global]

```
x | 3
Return value | 9
```

f2: square [parent=Global]

```
x | 9
Return value | 81
```

An environment is a sequence of frames.
Multiple environments in one diagram!

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

square(square(square(3)))
```

An environment is a sequence of frames.
• Environment: Global frame
def square(x):
    return x * x

square(square(3))

An environment is a sequence of frames.
- Environment: Global frame
- Environment: Local frame (f1), then global frame
Multiple environments in one diagram!

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

square(square(square(3)))
```

An environment is a sequence of frames.
• Environment: Global frame
• Environment: Local frame (f1), then global frame
• Environment: Local frame (f2), then global frame
Names have no meanings without environments

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

square(square(square(3)))
```

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

Names have different meanings 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
Names have different meanings in different environments

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

square(4)
```

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
Environments for higher-order functions
Review: Higher-order functions

A higher-order function is either...

- A function that takes a function as an argument value
  `summation(5, lambda x: x**2)`
- A function that returns a function as a return value
  `make_adder(3)(1)`

**Functions are first class:** Functions are values in Python.
Example: Apply twice

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

def square(x):
    return x ** 2

apply_twice(square, 3)
```

[View in PythonTutor]
Arguments bound to functions
Arguments bound to functions
Arguments bound to functions
Environments for nested definitions
Example: Make texter

def make_texter(emoji):
    def texter(text):
        return emoji + text + emoji
    return texter

happy_text = make_texter("😊")
result = happy_text("lets go to the beach!")

View in PythonTutor
Environments for nested def statements
Environments for nested def statements

- Every user-defined function has a parent frame
- The parent of a function is the frame in which it was defined
Environments for nested def statements

- Every user-defined function has a parent frame.
- The parent of a function is the frame in which it was defined.
- Every local frame has a parent frame.
- The parent of a frame is the parent of the called function.
Environments for nested def statements

• Every user-defined function has a parent frame.
• The parent of a function is the frame in which it was defined.
• Every local frame has a parent frame.
• The parent of a frame is the parent of the called function.
• An environment is a sequence of frames.
How to draw an environment diagram

When a function is defined:

1. Create a function value:
   ```
   func <name>(<formal parameters>) [parent=<label>]
   ```
2. Its parent is the current frame.
3. 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
Example: Thingy Bobber

```python
def thingy(x, y):
    return bobber(y)

def bobber(a):
    return a + y

result = thingy("ma", "jig")
```

What do you think will happen?
Example: Thingy Bobber

```python
def thingy(x, y):
    return bobber(y)

def bobber(a):
    return a + y

result = thingy("ma", "jig")
```

What do you think will happen?

View in PythonTutor
Local name visibility

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 consists of one local frame followed by the global frame.
Function Composition
def happy(text):
    return "☺" + text + "☺"

def sad(text):
    return "☺☺" + text + "☺☺"

def composer(f, g):
    def composed(x):
        return f(g(x))
    return composed

msg1 = composer(sad, happy)("cs61a!")
msg2 = composer(happy, sad)("eecs16a!")

What do you think will happen?
Example: Composer (Part 2)

One of the composed functions could itself be an HOF...

def happy(text):
    return "☺" + text + "☺"

def sad(text):
    return "☺" + text + "☺"

def make_texter(emoji):
    def texter(text):
        return emoji + text + emoji
    return texter

def composer(f, g):
    def composed(x):
        return f(g(x))
    return composed

composer(happy, make_texter("☃"))("snow day!")

View in PythonTutor
Composer 2 expression tree

composer(happy, make_texter("☃"))("snow day!")
Composer 2 expression tree

\[
\text{composer(happy, make_texter("☃"))("snow day!")}
\]

\[
\text{composer(happy, make_texter("☃"))}
\]
Composer 2 expression tree

```python
func composer(f, g)

composer(happy, make_texter("☃"))("snow day!")
```

```python
composer(happy, make_texter("☃"))
```
Composer 2 expression tree

```
func composer(f, g)  func happy(text)

composer(happy, make_texter("☃"))("snow day!")

composer(happy, make_texter("☃"))
```
Composer 2 expression tree

```
func composer(f, g) func happy(text)

make_texter("☃")

composer(happy, make_texter("☃"))("snow day!")
```
Composer 2 expression tree

```
composer(happy, make_texter("☃"))("snow day!")
```

```
func composer(f, g) func happy(text)
```

```
make_texter("☃")
```

```
func make_texter(emoji)
```
Composer 2 expression tree

```
func composer(f, g) func happy(text)

make_texter("☃")

func make_texter(emoji)
"☃"
```

```
composer(happy, make_texter("☃"))("snow day!")
```
Composer 2 expression tree

```
composer(happy, make_texter("☃"))("snow day!")
```

```
func composer(f, g) func happy(text)
```

```
make_texter("☃")
```

```
func make_texter(emoji) "☃"
```
Composer 2 expression tree

```
composer(happy, make_texter("☃"))("snow day!")
```

```
func composer(f, g)  func happy(text)  func texter(text)
make_texter("☃")
```

```
func make_texter(emoji)  "☃"
```
Composer 2 expression tree

```text
composer(happy, make_texter("osomesay"))("snow day!")
```
Composer 2 expression tree

```
func composer(f, g) func happy(text) func texter(text)

func composed(x) composer(happy, make_texter("☃"))

composer(happy, make_texter("☃"))("snow day!"))

func make_texter(emoji) "☃"
```
Composer 2 expression tree

\[
\text{func} \\ \text{composed}(x) \\
\text{composer}(\text{happy, make_texter("☃")})("snow day!")
\]
composer(happy, make_texter("☃"))("snow day!")

func composed(x)

composer(happy, make_texter("☃"))

func composer(f, g) func happy(text) func texter(text)

make_texter("☃")

func make_texter(emoji) "☃"
Composer 2 expression tree

```
func composer(f, g) {  
  function happy(text) {  
    function texter(text) {  
      make_texter("☃")  
    }  
  }  
  composed(x) {  
    composer(happy, make_texter("☃"))("snow day!")  
  }  
}

"☃ snow day! ☃"
```

Diagram:

- `composer(happy, make_texter("☃"))("snow day!")`
- `func composed(x)`
- `func happy(text)`
- `func texter(text)`
- `func make_texter(emoji) "☃"`
Currying
add vs. make_adder

Compare...

```python
from operator import add

add(2, 3)
```

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

make_adder(2)(3)
```

What's the relationship between `add(2, 3)` and `make_adder(2)(3)`?
Function currying

**Currying:** Converting a function that takes multiple arguments into a single-argument higher-order function.

A function that currys any two-argument function:

```python
def curry2(f):
    def g(x):
        def h(y):
            return f(x, y)
        return h
    return g
```
**Function currying**

**Currying:** Converting a function that takes multiple arguments into a single-argument higher-order function.

A function that currys any two-argument function:

```python
def curry2(f):
    def g(x):
        def h(y):
            return f(x, y)
        return h
    return g
```

```python
make_adder = curry2(add)
make_adder(2)(3)
```
**Function currying**

**Currying:** Converting a function that takes multiple arguments into a single-argument higher-order function.

A function that currys any two-argument function:

```python
def curry2(f):
    def g(x):
        def h(y):
            return f(x, y)
        return h
    return g
make_adder = curry2(add)
make_adder(2)(3)
curry2 = lambda f: lambda x: lambda y: f(x, y)
```
Why "currying"?

It's not food! ✗  ✗

Named after American logician Haskell Curry, but actually published first by Russian Moses Schönfinkel, based on principles by German Gottlob Frege.

See also: Stigler's law of eponymy