Environments
Class outline:

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

What happens?

**Def statement**

- A new function is created!
- Name bound to that function in the current frame.

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

**Call expression**

- Operator & operands evaluated
- Function (value of operator) called on arguments (values of operands)

```
square(2 + 2)
```

**Calling/applying**

- A new frame is created!
- Parameters bound to arguments
- Body is executed in that new environment

```
def square(x):
    # Body of function
```

```
def square(x):
    # Body of function
```

```
```
A nested call expression

def square(x):
    return x * x

square(square(3))
A nested call expression

1. 
2. 
3. 

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

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

def square(x):
    return x * x

square(square(3))
A nested call expression

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

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

Global frame

```
  square [•]  ----> func square(x) [parent=Global]
```
\textbf{square}( \textbf{square}(3) \ )
A nested call expression

def square(x):
    return x * x

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

func square(x)
A nested call expression

def square(x):
    return x * x

square(square(square(3)))

Global frame

    square [•] ----> func square(x) [parent=Global]
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]]
\textit{square( square(3) )}

\begin{itemize}
  \item \texttt{func square(x)}
  \item \texttt{square(3)}
  \item \texttt{func square(x)}
\end{itemize}
A nested call expression

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

square(square(3))
```

Global frame

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

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

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

square(3)

square( square(3) )
A nested call expression

1. def square(x):

2.     return x * x

3. square(square(square(3)))

Global frame

- square [parent=Global]

f1: square [parent=Global]

  x | 3
A nested call expression

def square(x):
    return x * x

square(square(3))
\textbf{func square}(x) \\
\textbf{square}(\textbf{square}(3)) \\
\textbf{square}(3) \\
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]
```
square(some_value)
A nested call expression

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

f2: square [parent=Global]
square(square(3))

func square(x)

3
def square(x):
    return x * x

square(square(3))

Global frame

f1: square [parent=Global]
    x 3
    Return value 9

f2: square [parent=Global]
\begin{align*}
\text{func square}(x) & \quad \text{9} \\
\text{square}(3) & \quad \text{square}(3) \\
\text{func square}(x) & \quad 3
\end{align*}
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
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
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
def square(x):
    return x * x

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.
Names have different meanings in different environments

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
Names have different meanings in different environments

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
  \[
  \text{summation}(5, \lambda x: x**2)
  \]
- A function that returns a function as a return value
  \[
  \text{make_adder}(3)(1)
  \]

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

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

def square(x):
    return x ** 2

apply_twice(square, 3)
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!")
Environments for nested def statements
Environments for nested def statements

- Every user-defined \textit{function} has a parent frame
- The parent of a \textit{function} is the \textit{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

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
Example: Composer

```python
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!')
Composer 2 expression tree

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

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

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

```plaintext
func composer(f, g)

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

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)

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

composer(happy, make_texter("☃"))

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

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

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

make_texter("☃")

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

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

\[
\text{func composer(f, g) func happy(text)}
\]

\[
\text{make_texter("❄")}
\]

\[
\text{func make_texter(emoji) "❄"}
\]
Composer 2 expression tree

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

make_texter("☃")
```

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

```
composer(happy, make_texter("☃"))
```

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

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

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

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

func texter(text)

func composed(x)

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

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

\[
\text{func composer}(f, g) \quad \text{func happy}(text) \quad \text{func texter}(text) \\
\text{func make_texter}(emoji) \quad "\hat{\text{snow}}" \\
\text{func composed}(x) \\
\text{composer}(\text{happy}, \text{make_texter}("\hat{\text{snow}}"))("snow day!")
\]
Composer 2 expression tree

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

func make_texter(emoji) "☃"

func composed(x)

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

```

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

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

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

```
func composed(x)
```

```
"☃snow day!☃"
```
Self-reference
A self-referencing function

A higher-order function could return a function that references its own name.

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

print_sums(1)(3)(5)

View in PythonTutor
Environment for print_sums
Understanding print_sums

The call:

```python
print sums(1)(3)(5)
```

produces the same result as:

```python
g1 = print sums(1)
g2 = g1(3)
g2(5)
```

A call to `print sums(x)` returns a function that:

- Prints `x` as a side-effect, and
- Returns a function that, when called with argument `y`, will do the same thing, but with `x+y` instead of `x`.

So these calls will...

- First print 1 and return `g1`,
- which when called with 3, will print 4 (= 1+3) and return `g2`,
- which when called with 5, will print 9 (= 4+5), and return. . . .
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)
```

```python
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