Environments
Announcements

- Hog, HW1, and Lab 1 have been released!
  - Lab 1 is due tomorrow
  - HW 1 is due Thursday
  - Hog Checkpoint is due Friday

- Tutoring section sign ups released!
  - tutorials.cs61a.org

- Regular OH this week!
  - Calendar: https://cs61a.org/office-hours/

- Instructor OH Schedule in Soda 781
  - Jordan: Mondays, 12:45 – 1:45 pm
  - Noor: Tuesdays, 9:30 – 10:30 am
  - Tim: Thursdays, 12:45 – 1:45 pm

- Sections will be finalized 6/30
  - sections.cs61a.org
Environment Diagrams
Environment Diagrams

Environment diagrams visualize the interpreter’s process.

Just executed
1. `from math import pi`
2. `tau = 2 * pi`

Next to execute
Assignment statement

Code (left):
Statements and expressions
Arrows indicate evaluation order

Frames (right):
Each name is bound to a value
Within a frame, a name cannot be repeated
Why Use Environment Diagrams?

- They help us understand why the programs we design work the way they do!
  - Predict how a program will behave

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

- They can also be useful in debugging!
  - When we run into an unexpected error, we can trace back our steps!
What We Have Seen So Far

• Assignment Statements
  • x = 2
  • var = 5

• Def Statements
  • def square(x):
    return x * x

• Call Expressions
  • square(var)

Environment Diagram
Assignment Statements

Execution rule for assignment statements:

1. Evaluate all expressions to the right of = from left to right.

2. Bind all names to the left of = to those resulting values in the current frame.
Procedure for calling/applying user-defined functions:

1. Add a local frame
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment

```
from operator import mul

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

square(-2)
```
Calling User-Defined Functions

Procedure for calling/applying user-defined functions:

1. Add a local frame
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment

A function’s signature has all the information needed to create a local frame

```
from operator import mul

def square(x):
    return mul(x, x)
square(-2)
```
Frames

- A frame keeps track of variable-to-value bindings

- By default, the global frame is the starting frame
  - It doesn’t correspond to a specific call expression

- Every call expression has a corresponding frame

- The parent of a function is the frame in which it was defined, not called
  - Important for variable lookup!
  - If you cannot find a name in the current frame, you can go up to its parent until you reach the global frame
    - If it is not found, you get a NameError: name 'x' is not defined

Demo
When a function is defined:

Create a function value: func <name>(<formal parameters>) [parent=<label>]

Its parent is the current frame.

```
1    def square(x):
2       return x * x
3
4    def make_adder(n):
5       def adder(k):
6           return n + k
7       return adder
```

Frames

Objects

Global frame

- func square(x) [parent=Global]
- func make_adder(n) [parent=Global]
How to Draw an Environment Diagram

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.

```python
1   def square(x):
2       return x * x
3
4  def make_adder(n):
5      def adder(k):
6          return n + k
7      return adder

make_adder(5)
```
Check Your Understanding: Calling Functions

1 from operator import pow
2
3 def pow(x, y):
4     return x ** y
5
6 def power_of_pow(x, y):
7     return pow(pow(y, x), pow(x, y))
8
9 power_of_pow(2, 3)
Evaluation Order

- An environment diagram reflects Python evaluation order
  - Evaluate the operator, then the operands, finally apply the operator to the operands
Lambda Expressions
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
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!
Check Your Understanding: Calling Lambda

1  y = 6
2
3  def apply_func(f, x):
4     return f(x)(y)
5
6
7  apply_func(lambda x: lambda y: x + y + 1, 5)
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!
Revisiting Evaluation Order

• Even with higher-order function, the rules remain the same and the environment diagram reflects Python evaluation order!

• Evaluate the operator, then the operands, finally apply the operator to the operands

```python
def make_adder(n):
    def adder(k):
        return n + k
    return adder

make_adder(3)(5)
```

(Demo)
Currying
Function Currying

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

>>> make_adder(2)(3)
5
>>> add(2, 3)
5
```

There's a general relationship between these functions (Demo)

**Curry:** Transform a multi-argument function into a single-argument, higher-order function
Summary

- Using **environment diagrams** to visualize and understand programming
  - Diagramming follow the evaluation procedure for Python
  - Think deeply about how the code you write actually works

- **Lambda expressions**
  - Similar to user-defined functions but are anonymous
  - They are simple and can be created for one-time use or stored by assigning it to a variable

- The same rules of diagramming apply to **HOFs**, which take in a function as an input to return a function as an output

- To **curry** a multi-argument function is to transform it into a single-argument, multi-nested HOF