CS 61A SU26

Lecture 04: Environments

June 25, 2026

Rebecca Dang

Join pollev.com/rebeccadang831 (or scan QR code) on your phone or laptop

We'll begin at Berkeley time (10 minutes after), as per tradition!

Potpourri

5 min

- Announcements
- Review: HOFs

Announcements

- **Exam prep tutoring section** sign-ups (optional) are released on Ed
- Out of fairness to all students and to preserve Rebecca's work-life balance:
  - We will have a **cutoff for post-lecture questions at 7 pm**
  - **Please ask 1 question at a time**
  - If you didn't get a chance to ask your question, please come to OH (either instructor or regular), or ask on Ed instead!
- **Rebecca's instructor OH on Thursdays** occur right after lecture, so it will be a **"walk and talk"** until we arrive at Soda Hall
- For today, since we're doing environment diagrams I recommend **downloading a PDF and writing on the practice slides** if you have a tablet (**or write on a piece of paper**)

When poll is active respond at **PollEv.com/rebeccadang831**

Activate this Poll with the Poll Everywhere Live app.

If video conferencing, you must be sharing your full screen to share the activated poll.

HOFs: What would Python display?

Demo: On which line would the error occur?

Surprisingly, not at the line with `x += 1` even though it's the root cause!

Instead, it errors at `print(x)`.

"This is because when you make an assignment to a variable in a scope, that variable becomes local to that scope and shadows any similarly named variable in the outer scope. Since the last statement in `foo` assigns a new value to `x`, the **compiler** recognizes it as a local variable. Consequently when the earlier `print(x)` attempts to print the uninitialized local variable and an error results." (Python Docs)

Environments

15 min

- Environment Diagram Rules
- WWPD Conventions
- Example: Nested lambdas from Lec 03

Recall: Environment Diagram Rules (1 of 4)

- An **environment** is a sequence of **frames**
- A **frame** is a set of bindings between names and values
- All Python programs begin in the special **global frame** (which has no parent)

Recall: Environment Diagram Rules (2 of 4)

- A function's **parent frame** is the **frame in which the function is <u>defined</u>** (not the frame in which it's called)
- Each time we call a function, we open a new **local frame**
- When we try to retrieve the value of a name, we first look in the current frame
  - If it's not there, we recursively check if it exists in the **parent frame**
  - If we go all the way up to the global frame and it's still not there, throw a `NameError`
- ⚠ **Caution:** You can't update a variable outside your local frame (results in `UnboundLocalError`)

Recall: Environment Diagram Rules (3 of 4)

Defining functions

- Bind function name to function object
- For lambdas, use Greek letter `λ <line #>` instead of intrinsic name
  - Why track line number? You can have multiple lambdas
- Do not execute the body!

Recall: Environment Diagram Rules (4 of 4)

Calling functions

- Call expression steps
  - Evaluate operator (check if function exists)
  - Evaluate operands from left to right (simplify arguments and bind them to parameters in new local frame)
  - Apply operator to operands (execute the body)
- Every function returns something, either explicitly or implicitly
- Immediately exit frame once `return` statement is reached

WWPD Conventions

- **Tip**: Draw the environment diagram when doing WWPD questions
- If the expression would evaluate to a function object, write **Function**
- If the expression would cause an error, write **Error**
- If the expression would not cause anything to be displayed by the Python interpreter (ex: it evaluates to None), write **Nothing**

Example: Nested lambdas from Lec 03

```
(lambda x, y: lambda z: x * y + z)(1, 2)(3)
```

```
def outer(x, y):
    def inner(z):
        return x * y + z
    return inner
outer(1, 2)(3)
```

Tip: Rewrite `lambda` into `def` if you're confused on what it's doing! Or do the opposite if we ask you to write a nested `lambda`.

Practice

60 min

- Q1: Make Adder
- Q2: Too many letters!
- Q3: Statements Galore
- Q4: Sheep Detectives

Q1: Make Adder

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

>>> a = make_adder(2)
>>> a
_____________
>>> a()
_____________
>>> b = make_adder(3)
>>> a(1)
_____________
>>> b(1)
_____________
```

Q1: Make Adder (Solution)

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

>>> a = make_adder(2)
>>> a
Function
>>> a()
Error
>>> b = make_adder(3)
>>> a(1)
3
>>> b(1)
4
```

Q2: Too many letters!

```
j = 3
def h(i):
    j = 4
    return lambda z: j * i(z)
h(lambda z: j + z)(j)
```

5 min break

Q3: Statements Galore (not all blanks may be used) (1 of 2)

```
def f1(x, y):
    if x > (x + y):
        print(x)
        y = x
    if x > (x - y):
        print(y)
        x = y
    x + y

def f2(a, b):
    if a:
        return b and a
    else:
        return a or b
```

```
>>> z = f1(3, -5)
_____________
_____________
_____________
>>> z
_____________
>>> f1(-4, 7)
_____________
_____________
_____________
>>> f2(print('a'), 10 % 3)
_____________
_____________
```

Q3: Statements Galore (not all blanks may be used) (2 of 2)

```
def f1(x, y):
    if x > (x + y):
        print(x)
        y = x
    if x > (x - y):
        print(y)
        x = y
    x + y

def f2(a, b):
    if a:
        return b and a
    else:
        return a or b
```

```
>>> z = f1(3, -5)
_____________
_____________
_____________
>>> z
_____________
>>> f1(-4, 7)
_____________
_____________
_____________
>>> f2(print('a'), 10 % 3)
_____________
_____________
```

Q3: Statements Galore (Solution)

```
def f1(x, y):
    if x > (x + y):
        print(x)
        y = x
    if x > (x - y):
        print(y)
        x = y
    x + y

def f2(a, b):
    if a:
        return b and a
    else:
        return a or b
```

```
>>> z = f1(3, -5)
3
3
>>> z
Nothing
>>> f1(-4, 7)
7
>>> f2(print('a'), 10 % 3)
a
1
```

Q4: Sheep Detectives (a great movie!)

```
winter_lamb = lambda george: 2 ** george
lily = lambda detective: lambda: detective
mopple = 3
def sebastian(f, g, h):
    return f(g(h))
sebastian(lily, winter_lamb, mopple)()
```