CS61A Midterm 1 Review

Riyaz Faizullabhoy
Joy Jeng
Mark Miyashita
Jonathan Kotker

What will Python print?

```
def while_loop(n):
    i, j, k = 0, 1, 2
    while i < n:
        j += 1
        while j < n:
            i += 1
            while k < n:
                print(i, j, k)
                k += 1
            j += 1
        i += 1

while_loop(5)
```

What will Python print?

```
def while_loop(n):
    i, j, k = 0, 1, 2
    while i < n:
        j += 1
        while j < n:
            i += 1
            while k < n:
                print(i, j, k)
                k += 1
            j += 1
        i += 1

while_loop(5)
```

```
>>> while_loop(5)
1 2 2
1 2 3
1 2 4
```

Why?

What will Python print?

```
def best(n):
    pikachu = n
    bulbasaur = 2
    charmander = 3
    if pikachu < bulbasaur:
        pikachu += charmander
    elif pikachu < charmander:
        print('pikachu!')
    else:
        if pikachu % 2 == 1:
            return 'pika!'
        else:
            return 'charmander'
```

```
>>> best(4)
________
>>> best(3)
________
>>> best(2)
________
>>> best(1)
________
```

What will Python print?

```
def best(n):
    pikachu = n
    bulbasaur = 2
    charmander = 3
    if pikachu < bulbasaur:
        pikachu += charmander
    elif pikachu < charmander:
        print('pikachu!')
    else:
        if pikachu % 2 == 1:
            return 'pika!'
        else:
            return 'charmander'
```

```
>>> best(4)
'charmander'
>>> best(3)
________
>>> best(2)
________
>>> best(1)
________
```

What will Python print?

```
def best(n):
    pikachu = n
    bulbasaur = 2
    charmander = 3
    if pikachu < bulbasaur:
        pikachu += charmander
    elif pikachu < charmander:
        print('pikachu!')
    else:
        if pikachu % 2 == 1:
            return 'pika!'
        else:
            return 'charmander'
```

```
>>> best(4)
'charmander'
>>> best(3)
'pika!'
>>> best(2)
________
>>> best(1)
________
```

What will Python print?

```
def best(n):
    pikachu = n
    bulbasaur = 2
    charmander = 3
    if pikachu < bulbasaur:
        pikachu += charmander
    elif pikachu < charmander:
        print('pikachu!')
    else:
        if pikachu % 2 == 1:
            return 'pika!'
        else:
            return 'charmander'
```

```
>>> best(4)
'charmander'
>>> best(3)
'pika!'
>>> best(2)
pikachu!
>>> best(1)
________
```

Notice the lack of quotes.

What will Python print?

```
def best(n):
    pikachu = n
    bulbasaur = 2
    charmander = 3
    if pikachu < bulbasaur:
        pikachu += charmander
    elif pikachu < charmander:
        print('pikachu!')
    else:
        if pikachu % 2 == 1:
            return 'pika!'
        else:
            return 'charmander'
```

```
>>> best(4)
'charmander'
>>> best(3)
'pika!'
>>> best(2)
pikachu!
>>> best(1)
>>>
```

Boolean Expressions

For reference, look at the first lab titled "Control"

- A *boolean expression* is one that evaluates to either True, False, or sometimes an Error.
- When evaluating boolean expressions, we follow the same rules as for evaluating other statements and function calls.
- The order of operations for booleans (from highest priority to lowest) is: **not, and, or**
- The following will evaluate to true:

```
True and not False or not True and False
```

- You can rewrite it using parentheses to make it more clear:

```
(True and (not False)) or ((not True) and False)
```

More Boolean Expressions

Short-circuiting

- Expressions are evaluated from left to right in Python.
- **and** will evaluate to `True` only if *all* the operands are `True`. For multiple **and** statements, Python will go left to right until it runs into the first `False` value -- then the expression will immediately evaluate to `False`.
- **or** will evaluate to `True` if *at least one* of the operands is `True`. For multiple **or** statements, Python will go left to right until it runs into the first `True` value -- then the expression will immediately evaluate to `True`. For example:

```
5 > 6 or 4 == 2*2 or 1/0
```

This evaluates to `True` because of short-circuiting.

What will Python print?

```
def sir_bool(x, y, z):
    w = x and y
    z = x and (z or 1/0)
    print(w or z)
    print(not(x) and z)
```

```
>>> sir_bool(True, False, True)
__________
>>> sir_bool(True, False, False)
__________
>>>
```

What will Python print?

```
def sir_bool(x, y, z):
    w = x and y
    z = x and (z or 1/0)
    print(w or z)
    print(not(x) and z)
```

```
>>> sir_bool(True, False, True)
True
False
>>> sir_bool(True, False, False)
________
>>>
```

What will Python print?

```
def sir_bool(x, y, z):
    w = x and y
    z = x and (z or 1/0)
    print(w or z)
    print(not(x) and z)
```

```
>>> sir_bool(True, False, True)
True
False
>>> sir_bool(True, False, False)
ZeroDivisionError: Division by
zero
>>>
```

More printing...

```
def print_moar(stuff):
    other_stuff = stuff
    i = 0
    while i < 2:
        other_stuff = print(other_stuff, print(stuff))
        i += 1
    return other_stuff

>>> print_moar('stuff')
```

More printing...

```
def print_moar(stuff):
    other_stuff = stuff
    i = 0
    while i < 2:
        other_stuff = print(other_stuff, print(stuff))
        i += 1
    return other_stuff

>>> print_moar('stuff')
stuff
stuff None
stuff
None None
```

Number Fun

Write a function that prints out the first n fibonacci prime numbers (a number that is both a Fibonacci number and a prime number). Assume that we gave you a function `is_prime` that returns a boolean expressing whether or not a number is prime.

```
def nth_fib_prime(n):
```

Number Fun

Write a function that prints out the first n fibonacci prime numbers (a number that is both a Fibonacci number and a prime number). Assume that we gave you a function `is_prime` that returns a boolean expressing whether or not a number is prime.

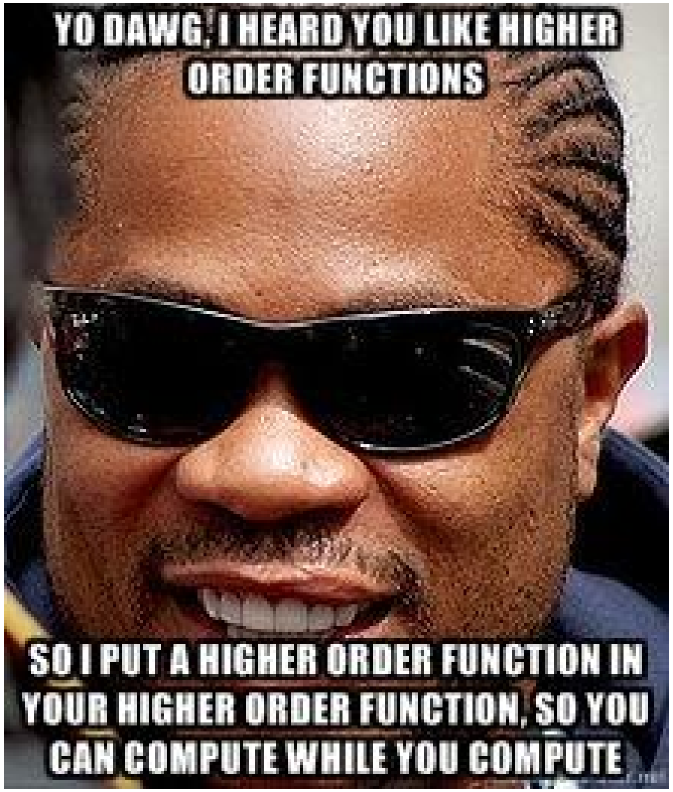
```
def nth_fib_prime(n):
    count = 0
    curr, next = 2, 3
    while count < n:
        if is_prime(curr):
            print(curr)
            count += 1
        curr, next = next, curr + next
```

Higher Order Functions

A function that takes in a function as an argument and/or returns a function.

```
def call_twice(func, x):
    return func(func(x))


def printXTimes(x):
    def multiple_print(stuff):
        for i in range(x):
            print(stuff)
    return multiple_print
```



Higher Order Functions

Write a function that takes two functions as arguments, `f` and `g`, and returns another function. The function returned takes in one integer argument, `x`. If `x` is odd, the output is `f(x)`; if `x` is even, the output is `g(x)`.

```
def alternate(f, g):
    *** YOUR CODE HERE ***

>>> h = alternate(lambda x: x + 1, lambda x: x**2)
>>> h(1)
2
>>> h(8)
64
```

Higher Order Functions

Write a function that takes two functions as arguments, `f` and `g`, and returns another function. The function returned takes in one integer argument, `x`. If `x` is odd, the output is `f(x)`; if `x` is even, the output is `g(x)`.

```
def alternate(f, g):
    def even_odd(x):
        # If x is odd, return f(x).
        if x % 2 == 1:
            return f(x)
        # If x is even, return g(x).
        else:
            return g(x)
    return even_odd
```

Higher Order Functions

Write a function `call_until_one` that takes a function we are interested in as an argument. It returns another function that, when called on a number, will tell you how many times you can call that original function on the number until it will return a value less than or equal to 1. For instance:

```
>>> f = call_until_one(lambda x: x - 1)
>>> f(100)
99

>>> g = call_until_one(lambda x: x / 2)
>>> g(128)
7
```

Higher Order Functions

Write a function `call_until_one` that takes a function we are interested in as an argument. It returns another function that, when called on a number, will tell you how many times you can call that original function on the number until it will return a value less than or equal to 1. For instance:

```
def call_until_one(func):
    def count_calls(x):
        counter = 0
        while x > 1:
            # Update the result with another call.
            x = func(x)
            counter += 1
        return counter
    return count_calls
```

Lambda Functions

- Unnamed function, no assignments
- "lambda <arguments> : <return value>"

```
>>> f = lambda x : x*2 + 1
>>> f(5)
11
>>> g = lambda y : y%2
>>> g(4)
0
>>> g(7)
1
>>> h = lambda x : 1 if x == 1 else 0
>>> h(1)
1
>>> h(1000)
0
```

Lambda Functions

We can even use lambdas as higher order functions!

```
>>> f = lambda x : x*2 + 1
>>> f(5)
11
>>> h = lambda  func, x: lambda y: func(x) + y
>>> h(f, 1)
<function <lambda> at 0x1005ee341>
>>> h(f, 1)(8)
11
```

Lambda Functions

Try on your own!

```
>>> x = lambda x: lambda: lambda y: 2*x + 3*y
>>> x
______
>>> x(3)
______
>>> x(3)(4)
______
>>> x(3)()
______
>>> x(3)()(7)
______
```

Lambda Functions

Try on your own!

```
>>> x = lambda x: lambda: lambda y: 2*x + 3*y
```

We can rewrite the lambdas using HOFs:

```
def L1(x):
# The line above could be "def x(x):" Why?
    def L2():
        def L3(y):
            return 2*x + 3*y
        return L3
    return L2
```

Lambda Functions

Try on your own!

```
>>> x = lambda x: lambda: lambda y: 2*x + 3*y
>>> x
<function <lambda> at 0x100483cf8>
>>> x(3)
______
>>> x(3)(4)
______
>>> x(3)()
______
>>> x(3)()(7)
______
```

Lambda Functions

Try on your own!

```
>>> x = lambda x: lambda: lambda y: 2*x + 3*y
>>> x
<function <lambda> at 0x100483cf8>
>>> x(3)
<function <lambda> at 0x100496b90>
>>> x(3)(4)
______
>>> x(3)()
______
>>> x(3)()(7)
______
```

Lambda Functions

Try on your own!

```
>>> x = lambda x: lambda: lambda y: 2*x + 3*y
>>> x
<function <lambda> at 0x100483cf8>
>>> x(3)
<function <lambda> at 0x100496b90>
>>> x(3)(4)
TypeError
>>> x(3)()
______
>>> x(3)()(7)
______
```

Lambda Functions

Try on your own!

```
>>> x = lambda x: lambda: lambda y: 2*x + 3*y
>>> x
<function <lambda> at 0x100483cf8>
>>> x(3)
<function <lambda> at 0x100496b90>
>>> x(3)(4)
TypeError
>>> x(3)()
<function <lambda> at 0x1004836e0>
>>> x(3)()(7)
______
```

Lambda Functions

Try on your own!

```
>>> x = lambda x: lambda: lambda y: 2*x + 3*y
>>> x
<function <lambda> at 0x100483cf8>
>>> x(3)
<function <lambda> at 0x100496b90>
>>> x(3)(4)
TypeError
>>> x(3)()
<function <lambda> at 0x1004836e0>
>>> x(3)()(7)
27
```

Applications of Higher-Order Functions: Newton's Method

Newton's method is used to find (approximately) the roots (or *zeros*) of a function f, where the function evaluates to zero.

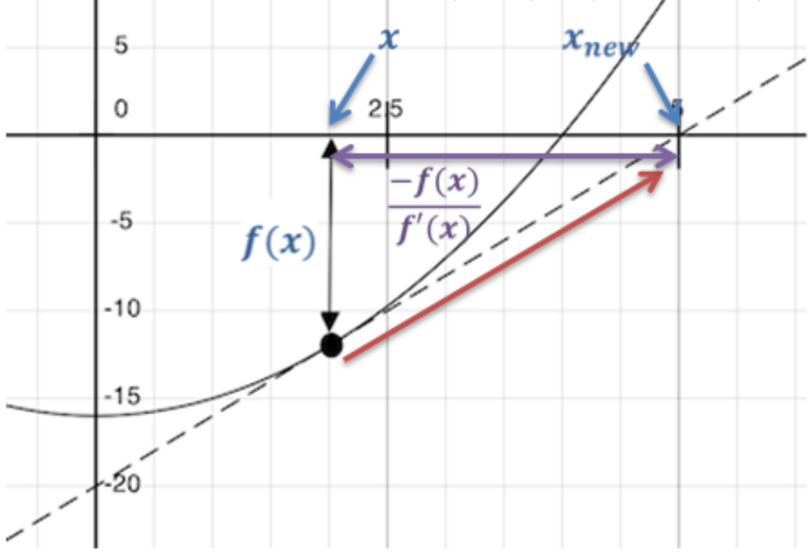
Many mathematical problems are equivalent to finding roots of specific functions.

"Square root of 2 is x, $x^2 = 2$" is equivalent to "Find the root of $x^2 - 2$."

"Power of 2 that is 1024" is equivalent to "Find the root of $2^x - 1024$."

Applications of Higher-Order Functions: Newton's Method

1. Start with a function `f` and a guess `x`.
2. Compute the value of the function `f` at `x`.
3. If zero, we are done; else, compute the derivative of `f` at `x`, `f'(x)`.
4. Update guess to be `x - (f(x)/f'(x))`.



Animation: http://en.wikipedia.org/wiki/File:NewtonIteration_Ani.gif

Applications of Higher-Order Functions: Newton's Method

Newton's method is an instance of *iterative improvement*.

Step 1: ***Guess*** an answer to the problem.

Step 2: If the guess is ***(approximately) correct***, it is the solution; otherwise, ***update*** the guess and repeat this step.

Applications of Higher-Order Functions: Newton's Method

```
def iter_improve(update, done, guess=1,
                 max_updates=1000):
    k = 1
    while not done(guess) and k < max_updates:
        guess = update(guess)
        k = k + 1
    return guess
```

Applications of Higher-Order Functions: Newton's Method

```
def derivative(fn, x, dx=1e-6):
    return (fn(x + dx) - fn(x))/dx

def newtons_method(fn, guess=1,
                   max_updates=1000):
    def newtons_update(guess):
        return guess - (fn(guess) / derivative(fn, guess))
    def newtons_isdone(guess):
        return abs(fn(guess)) <= 1e-6
    return iter_improve(newtons_update,
                        newtons_isdone,
                        guess, max_updates)
```

Applications of Higher-Order Functions: Newton's Method

"Square root of 2 is x, $x^2 = 2$"

`newtons_method(lambda x: x**2 - 2)`

"Power of 2 that is 1024"

`newtons_method(lambda x: 2**x - 1024)`

"Number x that is one less than its square, or $x = x^2 - 1$"

`newtons_method(________________________)`

Applications of Higher-Order Functions: Newton's Method

"Square root of 2 is x, $x^2 = 2$"

```
newtons_method(lambda x: x**2 - 2)
```

"Power of 2 that is 1024"

```
newtons_method(lambda x: 2**x - 1024)
```

"Number x that is one less than its square, or `x = x`^{2} `- 1`"

```
newtons_method(lambda x: x*x - x - 1)
```

Applications of Higher-Order Functions: Newton's Method

Write a function `find_extremum` that takes a function and an initial guess, and finds a local extremum of that argument function. A *local extremum* is a point at which the derivative of the function is zero. (You may assume that the argument function will never have any saddle points.)

An example call is provided:

```
>>> f = lambda x: (x-1)**2 # parabola with minimum at x=1
>>> find_extremum(f, 0)
0.999995 # very close to 1
```

Applications of Higher-Order Functions: Newton's Method

Write a function `find_extremum` that takes a function and an initial guess, and finds a local extremum of that argument function. A *local extremum* is a point at which the derivative of the function is zero. (You may assume that the argument function will never have any saddle points.)

```
def find_extremum(fn, guess):
    return newtons_method(lambda x: derivative(fn, x),
                          guess)
```

Applications of Higher-Order Functions: Newton's Method

Write a function `intersection(f, g)` that takes two functions, `f` and `g`, and finds a point at which the two are equal.

Applications of Higher-Order Functions: Newton's Method

Write a function `intersection(f, g)` that takes two functions, `f` and `g`, and finds a point at which the two are equal.

```
def intersection(f, g):
    return newtons_method(lambda x: f(x) - g(x))
```

Environment Diagrams

An Overview of Purpose

- A series of **frames**
- Represent different **scopes** that exist within a program
- Reflect **state**, so
- Keeps track of **variable bindings**

The **Online Python Tutor** (link from Resources on CS61A Webpage) can teach you to draw environment diagrams

Environment Diagrams

Rule One - Global Frame

- The frame in which all (python) programs begin

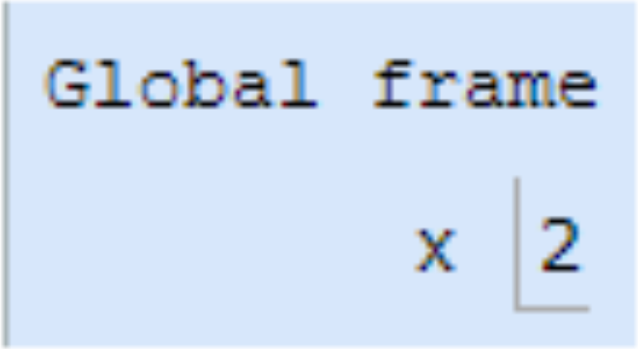
Draw a box and label it "Global Frame"

Environment Diagrams

Rule Two - Assignment / Bindings

- Variable **names** are bound to their **values** in the **current frame**

```
>>> x = 2
```



Variable names go inside the current frame, bound to their values

Environment Diagrams

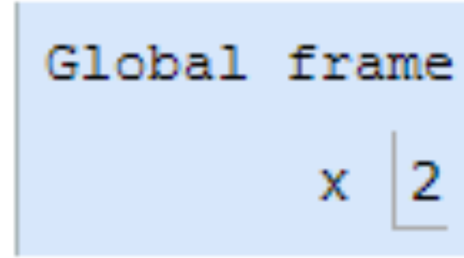
Rule Two - Assignment / Bindings

- **Evaluate** the **right side**
- **Bind** the variable **name** on left side to whatever the right side **evaluated to**

```
>>> x = add(2, mul(3, 1))
```

This is an expression. Evaluate this before binding x!

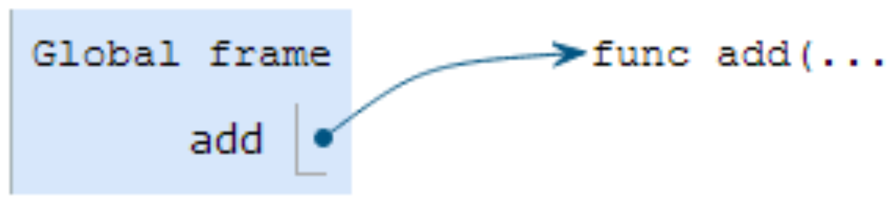
Environment Diagrams



Rule Two - Assignment / Bindings

- `def` statements and `import` statements are also assignments!

```
from operator import add
```




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

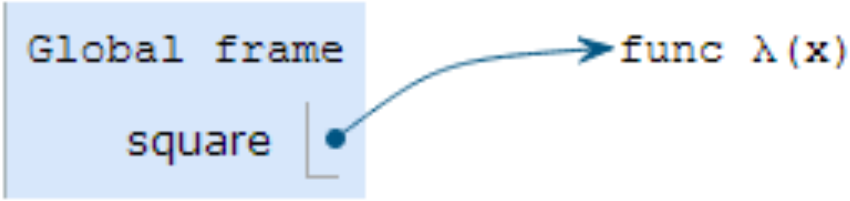
Environment Diagrams

Rule Two - Assignment / Bindings

- `lambdas` are **expressions** and only show up when bound (as return values or to names)

$$\left.\begin{array}{l}\texttt{>>> lambda x: x * x} \\ \texttt{>>> 2 + 3}\end{array}\right\}\ \text{expressions}$$

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

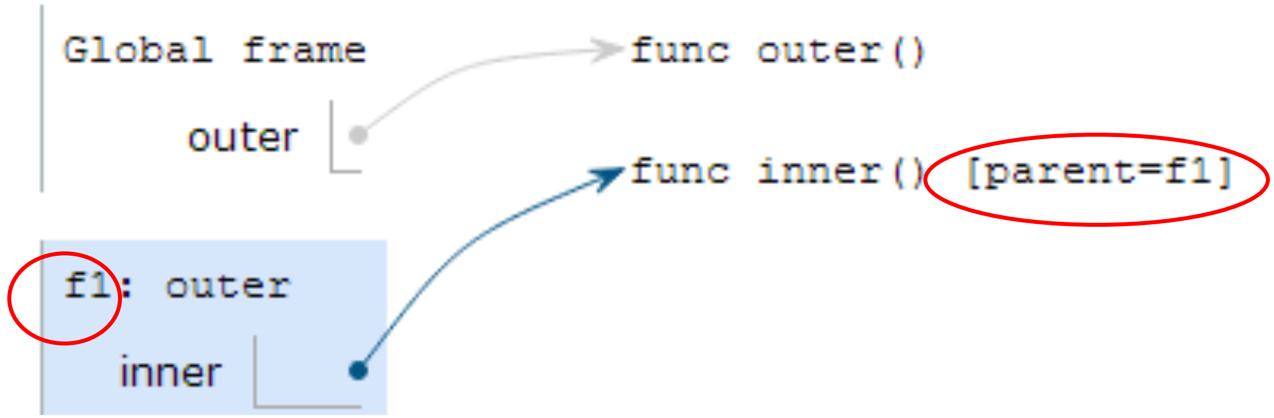


Note: **assignment** and lack of intrinsic name

Environment Diagrams

Rule Two - Assignment / Bindings

- Note that for functions, sometimes you need to denote the parent frame

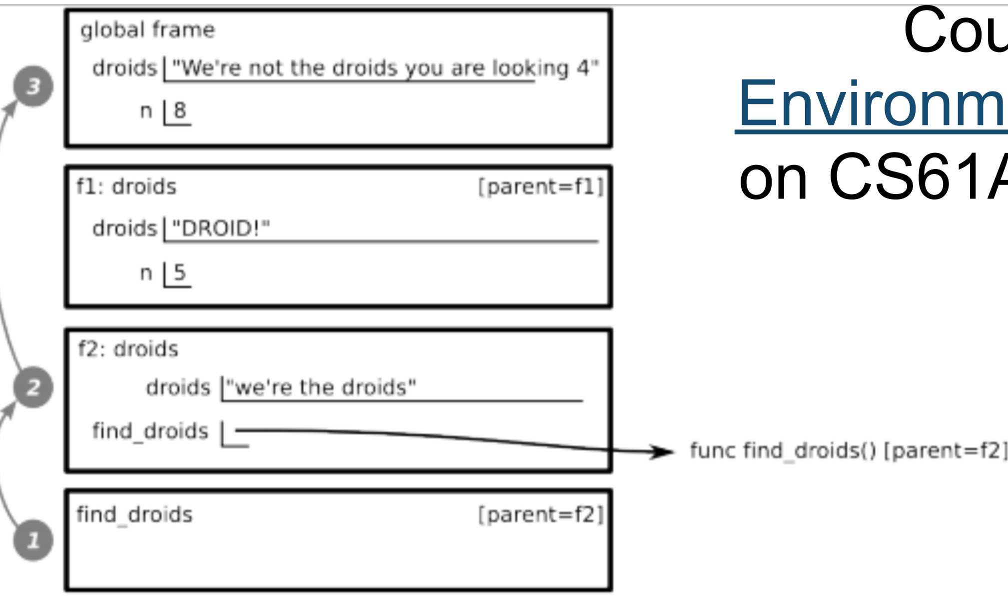



```
def outer():
    def inner():
        # Something
    # Stuff
outer()
```

Environment Diagrams

Rule Three - Variable Lookup

- When you **evaluate** an expression and look up the **value** of a variable, start in the **current frame** and **follow the parent frames**

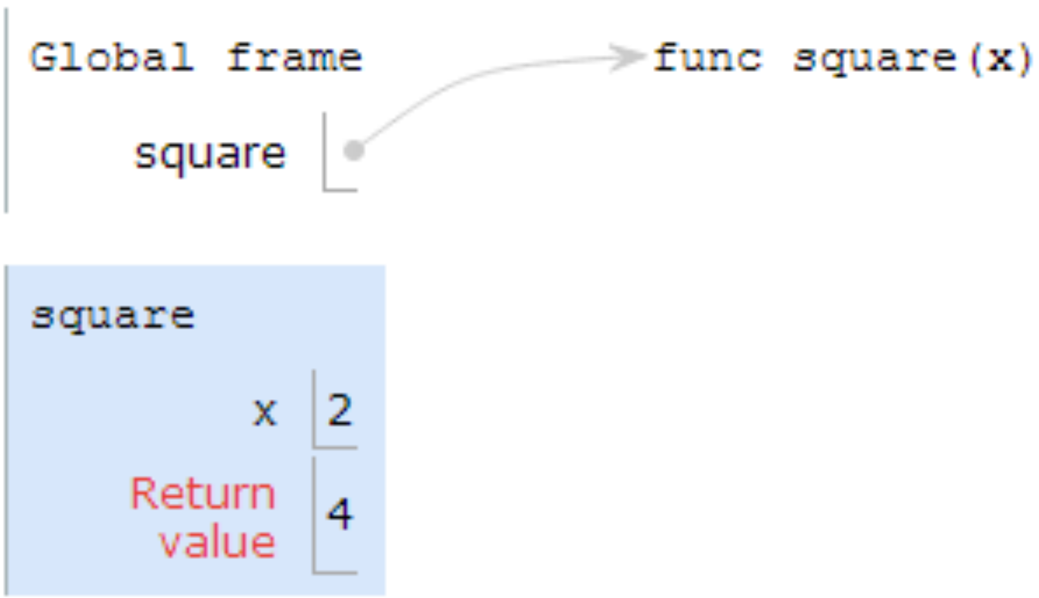


Courtesy of Drawing Environment Diagrams Handout on CS61A Webpage Resources

Environment Diagrams

Rule Four - Function Calls

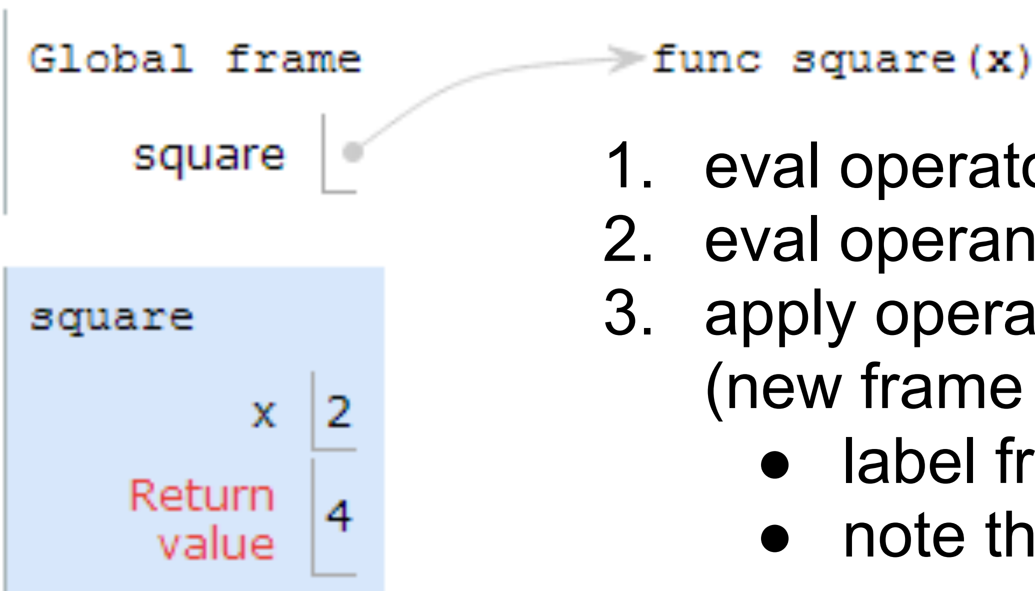
- For a **user-defined function call**, draw a **new frame!** Note: This is the **only situation** in which you draw a new frame.



Environment Diagrams

Rule Four - Function Calls

```
>>> square(2)
```



1. eval operator
2. eval operand(s)
3. apply operator to operands
   (new frame here)
   - label frame with intrinsic name
   - note the parent
   - copy parameters, bind to arguments
   - *Then*, run function body

Environment Diagrams

Putting it all together

- Only four rules (and one of them was the global frame)! Practice and you'll be fine.

```
def best_function_evar(underling):
    return "jk I do almost nothing"
```

Environment Diagrams

```
>>> def square(r):
...     return mul(r, r)
>>> from operator import mul
>>> joy, mark = 6, -3
>>> square = lambda x: x + 1
>>> def riyaz(mark):
...     mark = mark + joy
...     def jon(k):
...         return k(mark)
...     mark = 7
...     return jon
>>> riyaz(joy)(lambda x: 17)
```

```
>>> answer =
  riyaz(square(mul(joy, mark)))
>>> answer(print)
>>> add = print # bwahahahaha
```

Answer: http://goo.gl/J3wGr

Recursion

A function is ***recursive*** if the body calls the function itself, either directly or indirectly.

```
def factorial(n):
    if n == 1 or n == 0:
        return 1
    return n * factorial(n - 1)
```

Recursion

A recursive function has two important components:

1. A *base case*, where the function does not recursively call itself, and instead returns a direct answer. This is reserved for the simplest inputs.
2. A *recursive case*, where the function calls itself. The call must be made with an argument that drives the function towards the base case.

Recursion

A recursive function has two important components:

1. A *base case*.
2. A *recursive case*.

```
def factorial(n):
    if n == 1 or n == 0:
        return 1
    return n * factorial(n - 1)
```

Visualization: http://goo.gl/2rvW8

Recursion

Write a recursive function `add_until` that adds all of the integers from `1` to a positive integer `n`.

Recursion

Write a recursive function `add_until` that adds all of the integers from `1` to a positive integer `n`.

```
def add_until(n):
    if n == 1:
        return 1
    return n + add_until(n - 1)
```

Recursion

Write a recursive function `log` that takes a base `b` and a number `x`, and returns $\texttt{log}_{\texttt{b}}(\texttt{x})$, the power of `b` that is `x`. (Assume that `x` is some power of `b`.)

Recursion

Write a recursive function `log` that takes a base `b` and a number `x`, and returns $\log_b(x)$, the power of `b` that is `x`. (Assume that `x` is some power of `b`.)

```
def log(b, x):
    if x == 1:
        return 0
    return 1 + log(b, (x / b))
```

Recursion

Write a recursive function `eat_chocolate` that takes in a number of chocolate pieces and returns a string as follows:

```
>>> eat_chocolate(5)
"nom nom nom nom nom"
>>> eat_chocolate(2)
"nom nom"
>>> eat_chocolate(1)
"nom"
>>> eat_chocolate(0)
"No chocolate :("
```

Recursion

Write a recursive function `eat_chocolate` that takes in a number of chocolate pieces and returns a string:

```
def eat_chocolate(num_pieces):
    if num_pieces == 0:
        return "No chocolate :("
    elif num_pieces == 1:
        return "nom"
    return "nom " + \
        eat_chocolate(num_pieces - 1)
```

Recursion

Write the function `call_until_one` recursively. Recall that it takes a function we are interested in as an argument. It returns another function that, when called on a number, will tell you how many times you can call that original function on the number until it will return a value less than or equal to 1.

```
>>> f = call_until_one(lambda x: x - 1)
>>> f(100)
99

>>> g = call_until_one(lambda x: x / 2)
>>> g(128)
7
```

Recursion

Write the function `call_until_one` recursively. Recall that it takes a function we are interested in as an argument. It returns another function that, when called on a number, will tell you how many times you can call that original function on the number until it will return a value less than or equal to 1.

```
def call_until_one(func):
    def count_calls(x):
        if x <= 1:
            return 0
        return 1 + count_calls(func(x))
    return count_calls
```