

# EECS 490 – Lecture 8

## Functional Programming Examples

1

# Announcements

- Homework 2 due on Friday
- Project 2 due Fri 10/6

# Review: Environment of Use

- ▶ A function passed as a parameter has three environments that can be associated with it
  - ▶ The environment where it was defined
  - ▶ The environment where it was referenced
  - ▶ The environment where it was called
- ▶ Scope policy determines which names are visible in the function
  - ▶ Static/lexical scope: names visible at the definition point
  - ▶ Dynamic scope: names visible at the point of use
- ▶ In dynamic scope, point of use can be where a function is referenced or where it is called

# Review: Binding Policy

- *Shallow binding*: non-local environment is environment from where a function is called
- *Deep binding*: non-local environment is environment from where a function is referenced

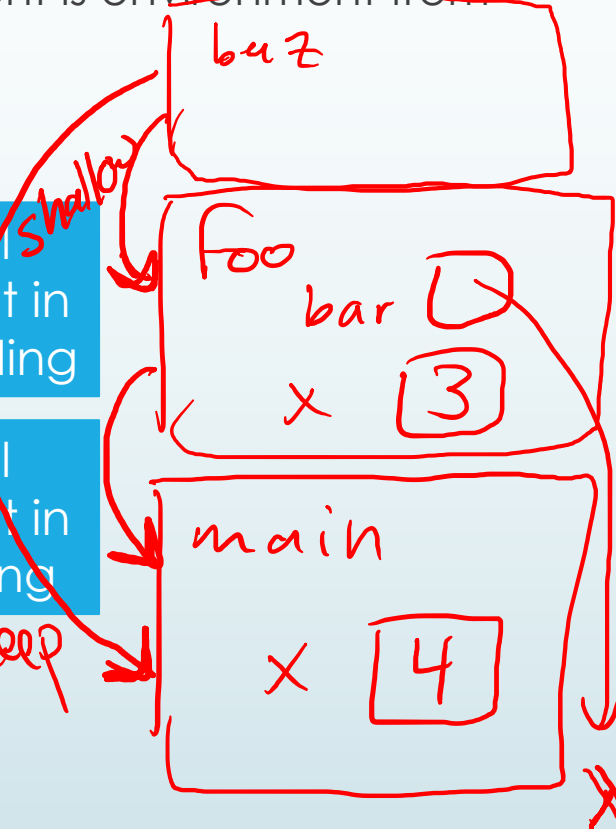
```
int foo(int (*bar)()) {
    int x = 3;
    return bar();
}
```

```
int baz() {
    return x;
}
```

```
int main() {
    int x = 4;
    print(foo(baz));
}
```

Non-local environment in shallow binding

Non-local environment in deep binding



# Agenda

- Nested Functions
- Example: Iterative Improvement

# Nested Functions and Closures

- ▶ The ability to create a function from within another function is a key feature of functional programming
- ▶ Static scope requires that the newly created function have access to its definition environment
- ▶ A closure is the combination of a function and its enclosing environment
- ▶ Variables from the enclosing environment that are used in the function are captured by the closure

# Nested Functions and State

- A closure encompasses state that can be accessed by the newly created function

```
def make_greater_than(threshold):
    def greater_than(x):
        return x > threshold
    return greater_than
```

threshold captured from non-local environment

```
>>> gt3 = make_greater_than(3)
>>> gt30 = make_greater_than(30)
>>> gt3(2)
False
>>> gt3(20)
True
>>> gt30(20), gt30(200)
(False, True)
```

mg+  
threshold [30]  
greater\_than [ ]

gt  
x [2]

mg+  
threshold [3]  
greater\_than [ ]

gt  
x [20]

global  
gt3 [ ]  
gt30 [ ]

parent  
10/16/17

# Modifying Non-Local State

- Languages may allow non-local variables to be modified

```
def make_account(balance):  
    def deposit(amount):  
        nonlocal balance  
        balance += amount  
        return balance  
    def withdraw(amount):  
        nonlocal balance  
        if 0 <= amount <= balance:  
            balance -= amount  
            return amount  
        else:  
            return 0  
    return deposit, withdraw
```


```
>>> deposit, withdraw = \  
    make_account(100)  
>>> withdraw(10)  
10  
>>> deposit(0)  
90  
>>> withdraw(20)  
20  
>>> deposit(0)  
70  
>>> deposit(10)  
80  
>>> withdraw(100)  
0  
>>> deposit(0)  
80
```




# Decorators

- A common pattern in Python is to transform a function or class by applying a higher-order function to it, called a *decorator*
- Standard syntax for decorating functions:

~~@<decorator>~~

 **def** <name>(<parameters>):  
    <body>

- Mostly equivalent to:

 **def** <name>(<parameters>):  
    <body>

<name> = <decorator>(<name>)

~~<name>~~

~~<decorator>~~



# Trace Example

*( 'a' , 'b' )*  
*'a' , 'b'*

- Example: decorator that traces function calls

```
def trace(fn):
    def tracer(*args):
        strs = (str(arg) for arg in args)
        print('{}({})'.format(fn.__name__,
                               ', '.join(strs)))
        return fn(*args)
    return tracer
```

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

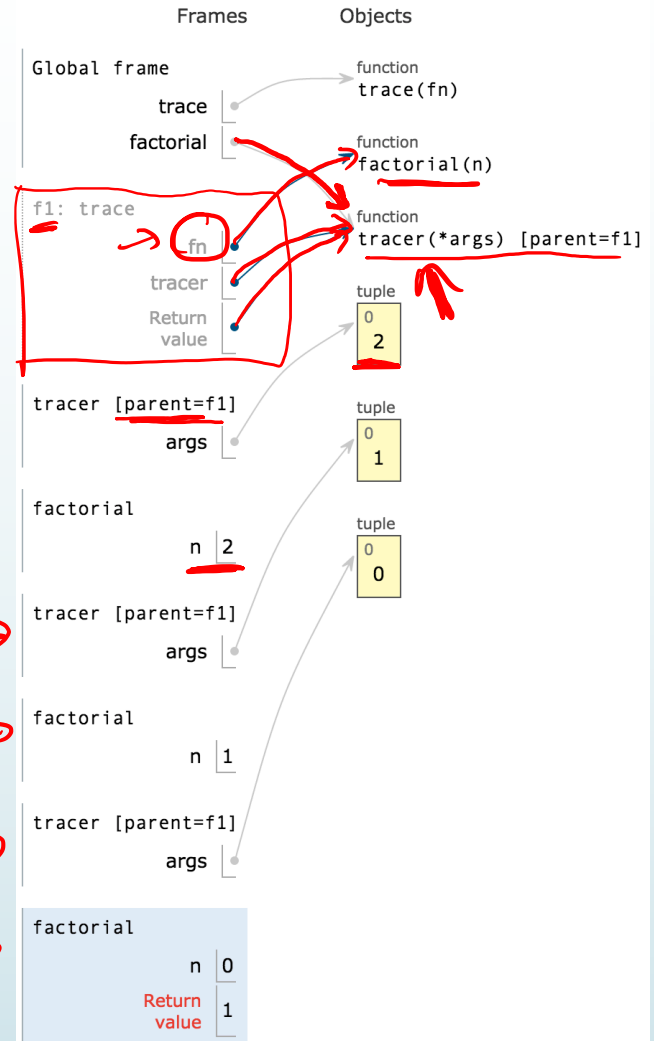
```
>>> factorial(5)
factorial(5)
factorial(4)
factorial(3)
factorial(2)
factorial(1)
factorial(0)
120
```

*factorial = trace(factorial)*

# Mutual Recursion

- A decorated recursive function results in *mutual recursion* where multiple functions make recursive calls indirectly through each other

```
>>> factorial(2)
factorial(2)
factorial(1)
factorial(0)
2
```



# Partial Application

- Specify some arguments to a function, then specify remaining arguments later
- If function takes  $n$  arguments and  $k$  are supplied, results in function that takes  $n-k$  arguments

```
def partial(func, *args):  
    def newfunc(*nargs):  
        return func(*args, *nargs)  
    return newfunc  
  
>>> power_of_two = partial(pow, 2)  
>>> power_of_two(3)  
8  
>>> power_of_two(7)  
128
```

C++ has bind() template to do this in `<functional>`.

set!

13

# Currying

- Transforms a function that takes  $n$  arguments into a series of  $n$  functions that each in one argument
- In some languages, all functions are curried

```
def curry2(func):  
    def curriedA(a):  
        def curriedB(b):  
            return func(a, b)  
        return curriedB  
    return curriedA
```

```
>>> curried_pow = curry2(pow)  
>>> curried_pow(2)(3)  
8
```

# Uncurrying

- We can also do the reverse transformation

```
def uncurry2(func):  
    def uncurried(a, b):  
        return func(a)(b)  
    return uncurried
```

```
>>> uncurried_pow = uncurry2(curried_pow)  
>>> uncurried_pow(2, 3)  
8
```

15

- ▶ We'll start again in five minutes.

# Iterative Improvement

- ▶ A pattern of computation that begins with an initial guess and repeatedly updates the guess until it is close enough to the intended value
- ▶ We can implement this using higher-order functions:

```
def improve(update, is_close, guess):  
    while not is_close(guess):  
        guess = update(guess)  
    return guess
```



# Golden Ratio

- Can be computed by repeatedly adding the inverse of a number to 1

```
def golden_update(guess):  
    return 1/guess + 1
```

- The ratio  $\varphi$  satisfies the equation  $\varphi^2 = \varphi + 1$

```
def is_golden(guess):  
    return approx_eq(guess * guess, guess + 1)
```

```
def approx_eq(x, y):  
    return abs(x - y) < 1e-10
```

```
>>> improve(golden_update, is_golden, 1)  
1.6180339887802426
```

# Scheme Implementation

- Improvement function uses tail recursion

```
(define (improve update close? guess)
  (if (close? guess)
      guess
      (improve update close? (update guess))))
```

```
(define (golden-update guess)
  (+ (/ 1 guess) 1))
```

```
(define (golden? guess)
  (approx-eq? (* guess guess) (+ guess 1)))
```

```
(define (approx-eq? x y)
  (< (abs (- x y)) 1e-10))
```

# Square Root

- We can use iterative improvement, but we don't want to write separate `update` and `is_close` functions for each input
- Instead, we define these functions dynamically as local functions:

```
(define (isqrt a)
  (define (update x)
    (average x (/ a x)))
  (define (close? x)
    (approx-eq? (* x x) a))
  (improve update close? 1.0))
```

```
(define (average x y)
  (/ (+ x y) 2))
```

# Newton's Method

- Generalized method to find the roots of a mathematical function using iterative improvement:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

- We can implement this using a higher-order function:

```
(define (find-root f df)
  (define (update x)
    (- x (/ (f x) (df x))))
  (define (close? x)
    (approx-eq? (f x) 0))
  (improve update close? 1.0))
```

# Nth Roots

- We can now use Newton's method to compute the nth root of a number:

```
(define (nth-root n a)
  (define (f x)
    (- (expt x n) a))
  (define (df x)
    (* n (expt x (- n 1))))
  (find-root f df))
```

```
> (nth-root 2 4)
2.0000000000000002
> (nth-root 4 4)
1.4142135623730951
> (nth-root 8 16)
1.414213562373095
```