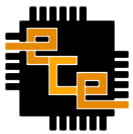
ECE 364
Software Engineering Tools Laboratory

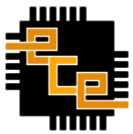
Lecture 9

Python: Advanced II



Lecture Summary

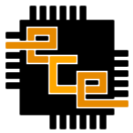
- Advanced Python Functions



Python Functions

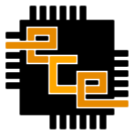
- A Python function can be treated like an object
  - Passed to other functions as an argument
  - Stored in a dictionary, list, tuple etc.
  - Assigned to variable or member variable

- Like objects, a Python function is referenced by its name
  - Use the function call operator `()` to invoke it



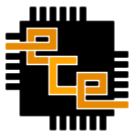
Python Functions (2)

```
def Foo():
  print("Foo Called")

def Bar():
  print("Bar Called")

# Assign function to a variable
X = Foo

# Call function through a variable
X()

# Call each function in a list
for f in [Foo, Bar]:
  f()
```



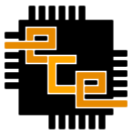
Python Functions (3)

```
def Add(x, y):
  return x+y

def Mul(x, y):
  return x*y

def Calc(op, x, y):
  # Works if op is a two argument function
  return op(x,y)

# Pass functions to other functions
Calc(Add, 5, 6)
Calc(Mul, 1, 2)
Calc(Add, Calc(Mul, 5, 5), 6)
```



Python Functions (4)

```
functions = {'foo':foo, 'bar':bar, 'baz':baz}

fn = raw_input('What function should I call?')
fn = fn.strip()

if fn in functions:
      # key maps to a function
      functions[fn]()
else:
      print("Unknown function!")
```

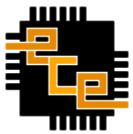


Lambda Functions

- A lambda function is an anonymous function

- Lambda functions are limited to a single statement
  - The statement can be a function call

```
lambda arg1, arg2,… argN: statement
```

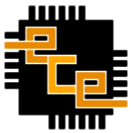


Lambda Functions (2)

- A lambda function consists of a set of arguments and a statement
  - The statement is the body of the function
  - When called, lambda functions return the value of the statement, no explicit return is required

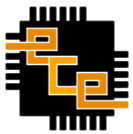
- The lambda expression produces a function not a value!

```
Add = lambda x, y: x+y
Mul = lambda x, y: x*y
A = Add(3, 4)
B = Mul(5, 5)
```



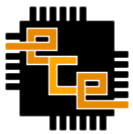
Lambda Functions (3)

```
def eq5(i):
  return i == 5

A = [1, 2, 5, 5, 6, 3] # map eq5 to A and get
B = map(eq5, A)        # list of booleans

# Create a function and name it eq6
eq6 = lambda x: x == 6

C = map(eq6, A)
D = map(lambda x: x == 6, A) # C and D are same
```



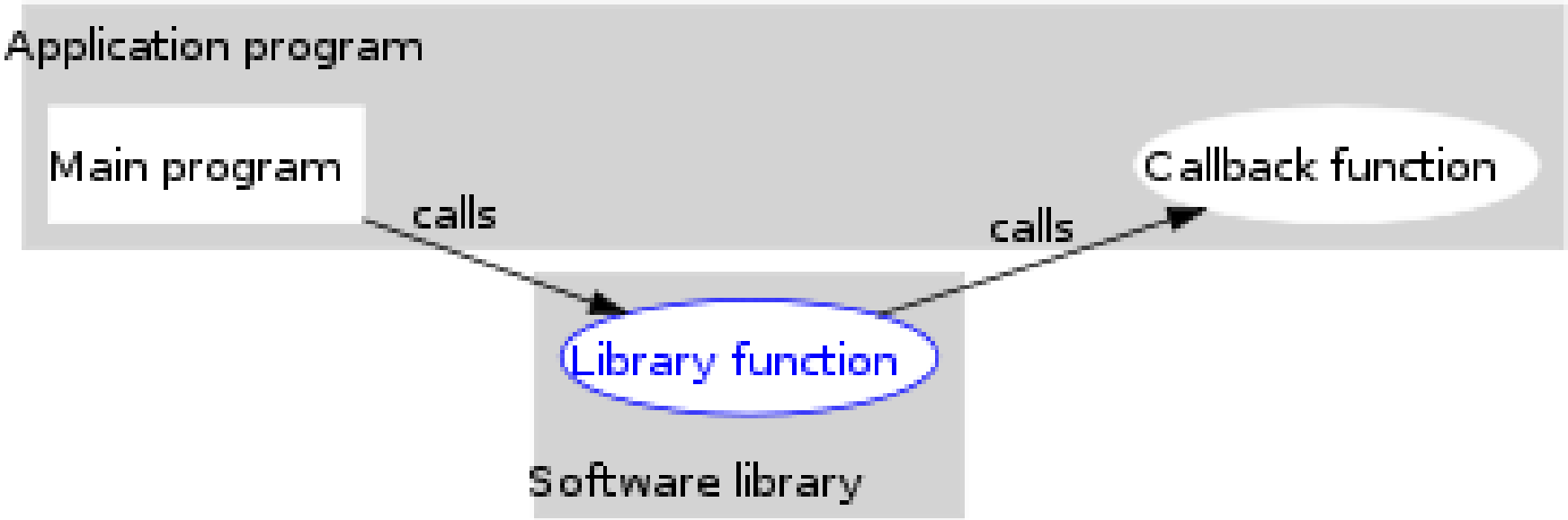
Lambda Functions (4)

```
# We can also create "factory" functions
# That return dynamic lambda functions

def make_incrementer(n):
   return lambda x: x+n

Inc = make_incrementer(1)
Inc2 = make_incrementer(2)

A = Inc(1)     # A == 2
B = Inc(5)     # B == 6
C = Inc2(8)    # C == 10
```



Callback Functions

- A callback function is a function that is passed to other functions as an argument
  - Callback functions are heavily used in GUI programming to respond to events

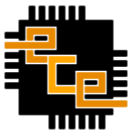


Source: http://en.wikipedia.org/wiki/File:Callback-notitle.svg

Memoization

- Memoization is a technique used to speedup computer programs by avoiding recalculation of results generated from previous inputs to a program or function

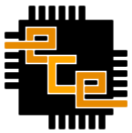
- Using a table (or some other structure) we can store the input to output mapping of a function and lookup the result



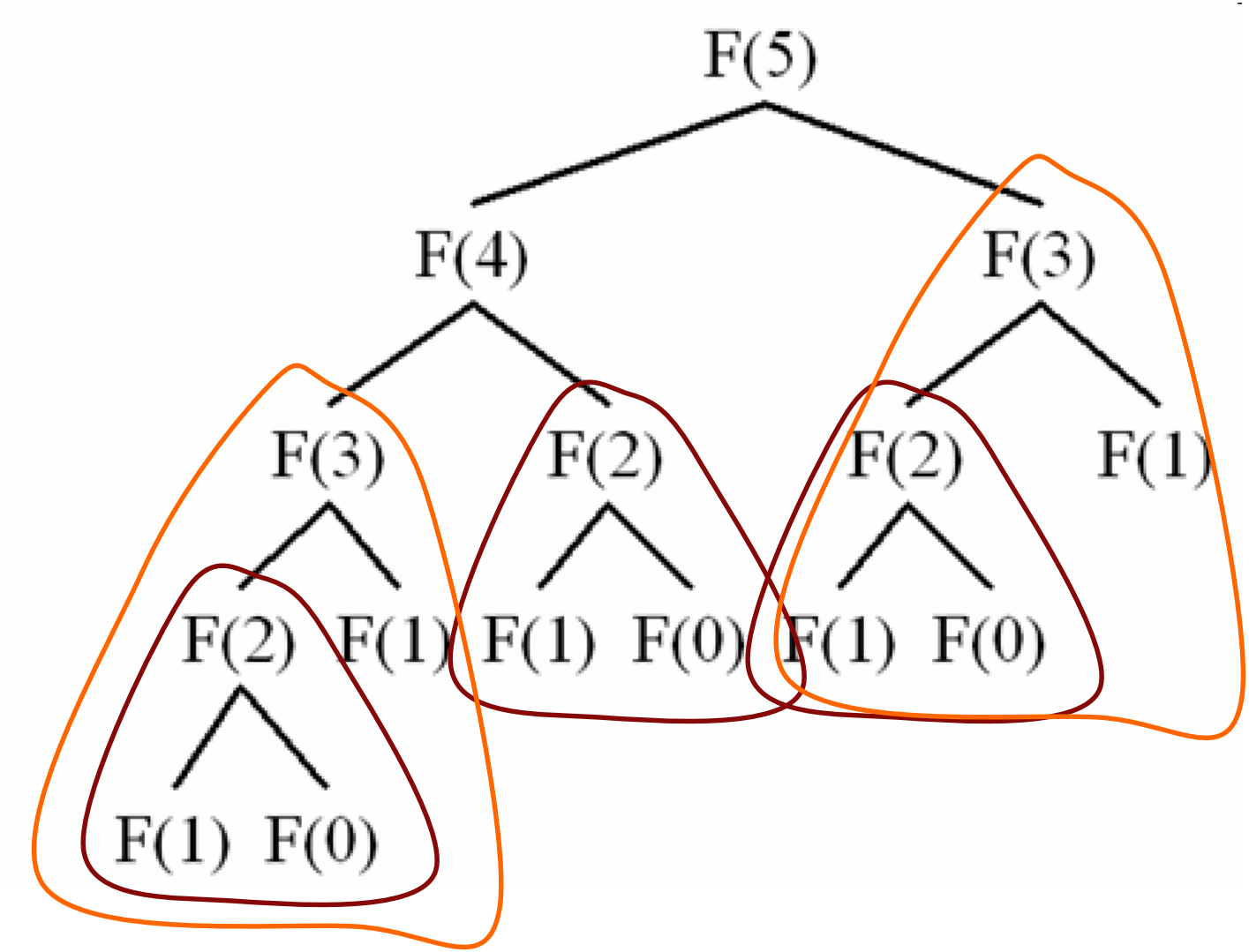
Memoization (2)

```
def fib(n):
    if n < 2:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

- To see how this recursive function could be optimized unwind the recursion



Memoization (3)

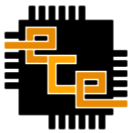


Source: http://www.cs.cmu.edu/~adamchik/15-121/lectures/Recursions/recursions.html

Memoization (4)

- Observe that when we recursively solve fib(4) we are also solving fib(3), fib(2) etc.

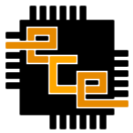
- When the recursion unwinds we come back to solve fib(3), but we already solved fib(3) before!
  - Without memoization we will repeat the work to solve fib(3)
  - With memoization we will have remembered the answer and not recalculated fib(3)



Memoization (5)

- A first approach to memoization

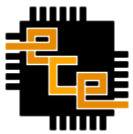
```
def fib(n, table):
  if n in table:
    return table[n]
  elif n < 2:
    table[n]=n
    return n
  else:
    table[n]=fib(n-2, table)+fib(n-1, table)
    return table[n]
```



Memoization (6)

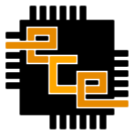
- The first approach works really well except we have introduced a second argument to maintain the memoization table
  - This makes things messy and messy code is difficult to debug and maintain

- A better approach would hide the implementation detail of memoization
  - Encapsulation!
  - What construct provides encapsulation?



Memoization (7)

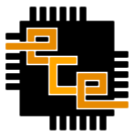
```
class fibonacci:
  def __init__(self):
    self.table={}   # state hidden in object

  def fib(self, n):
    if n in self.table:
      return self.table[n]
    elif n < 2:
      self.table[n]=n
      return n
    else:
      self.table[n]=self.fib(n-2)+self.fib(n-1)
      return self.table[n]
```



Memoization (8)

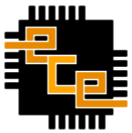
- The table that originally existed outside of the fib function is now encapsulated within an object
  - Now we do not have to keep track of it
  - The fib object will be responsible for maintaining the table

```
Memo_fib = fibonacci()
X = Memo_fib.fib(20)  # fast
Y = Memo_fib.fib(21)  # very fast
```



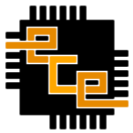
Memoization (9)

- `__call__` is a special function that lets you use the function call operator on an object
  - **Lets you treat objects like functions**
  - **Now you can create functions that contain state!**



Memoization (10)

```
class fibonacci:
  def __init__(self):
    self.table={} # state hidden in object

  def fib(self, n):
    if n in self.table:
      return self.table[n]
    elif n < 2:
      self.table[n]=n
      return n
    else:
      self.table[n]=self.fib(n-2)+self.fib(n-1)
      return self.table[n]

  def __call__(self, n):
    return self.fib(n)
```



Memoization (11)

```
Memo_fib = fibonacci()
X = Memo_fib(20)   # fast
Y = Memo_fib(21)   # very fast

def double(f, n):
   return 2 * f(n)

# Can pass object as a function
Z = double(Memo_fib, 10)
```