Welcome to CS61A Disc 3 Sect. 29/47 :D

Dickson Tsai
dickson.tsai+cs61a@berkeley.edu
OH: Tu, Th 4-5pm 411 Soda

Previous: Higher order functions **>>>**

Today: Recursion, Tree Recursion >>>

Next stop: Data Abstraction (a new unit!)

Section Quiz Logistics

- Please take out a piece of paper
- Put your name, login (e.g. cs61a-ty), SID, and section number (#47 for 11-12:30 pm, #29 for 6:30-8pm)
- Graded on effort: effort = revise your quiz with diff color

Section Quiz

1. How can I make your lab/discussion experience more worthwhile?
2. What would Python print?

```
>>> def foo(x, y):
...  if x == 0:
...    return y
...  print(x % 10)
...  return bar(x // 10, y)
...
>>> def bar(x, y):
...  return foo(x // 10, y * 10 + x % 10)
>>> foo(945113, 0)
```

Section Quiz [Solution]

1. What would Python print?

```
>>> def foo(x, y):
...   if x == 0:
...     return y
...   print(x % 10)
...   return bar(x // 10, y)
...
>>> def bar(x, y):
...   return foo(x // 10, y * 10 + x % 10)
...
>>> foo(945113, 0)
3
1
4
159
```

Foo prints the ones digit, removes it from x

Bar takes a digit, appends to y, and removes it

Section Quiz [Solution]

1. What would Python print?

```
>>> def foo(x, y):
...  if x == 0:
...    return y
...  print(x % 10)
...  return bar(x // 10, y)
...
>>> def bar(x, y):
...  return foo(x // 10, y * 10 + x % 10)
...
>>> foo(945113, 0)
3
1
4
159
```

Foo prints the ones digit, removes it from x

Bar takes a digit, appends to y, and removes it

If you know the rules well, recursion and HOF execution should not be anything special

Anuncios

- Midterm 1 is next Monday 9/22
- Hog Project 1 is due tonight for normal credit. Late submission by a day has 2/3 of earned score.
- Reminder: My OH are Tuesdays, Thursdays 4-5 PM in Soda 411
- **Review sessions for the midterm this Saturday and Sunday**
- **Do not** post any of your code on a public-viewable platform online. (e.g. Github, Pastebin)

Tip of the Day

- Trust your functions!
  - Seen in the section quiz: functions are meant to hide complexity
  - Recursion is incredibly expressive, because you can think in terms of the behavior and input of your function… when writing the same function!

```
def echo(word):
  ""” Takes in a word, prints the word, then
echoes the remaining word""”
  print(word)
```

Tip of the Day

- Trust your functions!
  - Seen in the section quiz: functions are meant to hide complexity
  - Recursion is incredibly expressive, because you can think in terms of the behavior and input of your function… when writing the same function!

```
def echo(word):
   ""” Takes in a word, prints the word, then
echoes the remaining word""”
   print(word)
   echo(word[1:])
```

Tip of the Day

- Trust your functions!
  - Seen in the section quiz: functions are meant to hide complexity
  - Recursion is incredibly expressive, because you can think in terms of the behavior and input of your function… when writing the same function!

```
def echo(word):
  ""” Takes in a word, prints the word, then
echoes the remaining word""”
  if word == “”:
    return
  print(word)
  echo(word[1:])
```

Recall: Characteristics of Functions

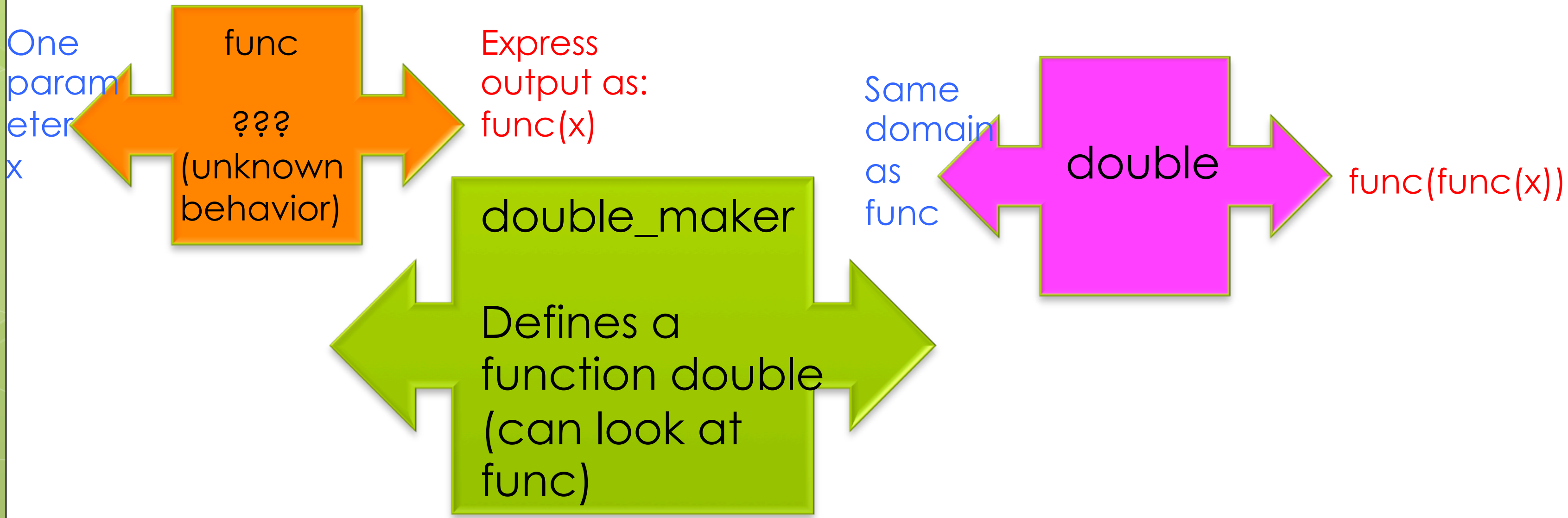
- Recall that a function has 3 characteristics: domain, range, and behavior
- Recall from Lab 2 (this is similar but not the same as make_buzzer):
- "Write a function `double_maker` that takes in a one-argument function `func` and returns a function (double) that takes in one argument and returns the result of calling `func` twice on that argument"
- [Discuss] What is the domain, range, and behavior of double_maker? What is a best way to draw all that information out visually?

Characteristics of Functions

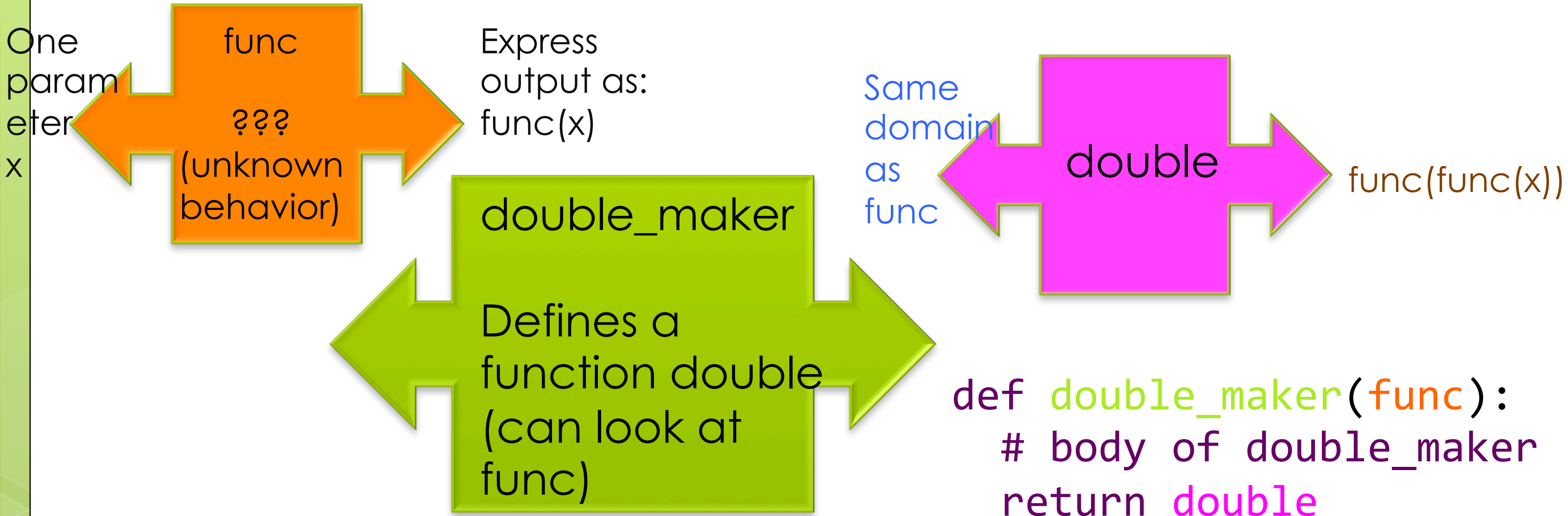
- “Write a function `double_maker` that takes in a one-argument function `func` and returns a function (double) that takes in one argument and returns the result of calling `func` twice on that argument”
- Domain: a one-argument function func
- Behavior: define a function that …
- Range: that defined function
  - Domain of returned fn: same domain as domain of func
  - Behavior: Call func twice on the input
  - Range: The value of that double function call

Characteristics of Functions

- “Write a function `double_maker` that takes in a one-argument function `func` and returns a function (double) that takes in one argument and returns the result of calling `func` twice on that argument”
- Best way to visualize:



Characteristics of Functions



One parameter x

func

???
(unknown behavior)

Express output as:
func(x)

double_maker

Defines a function double (can look at func)

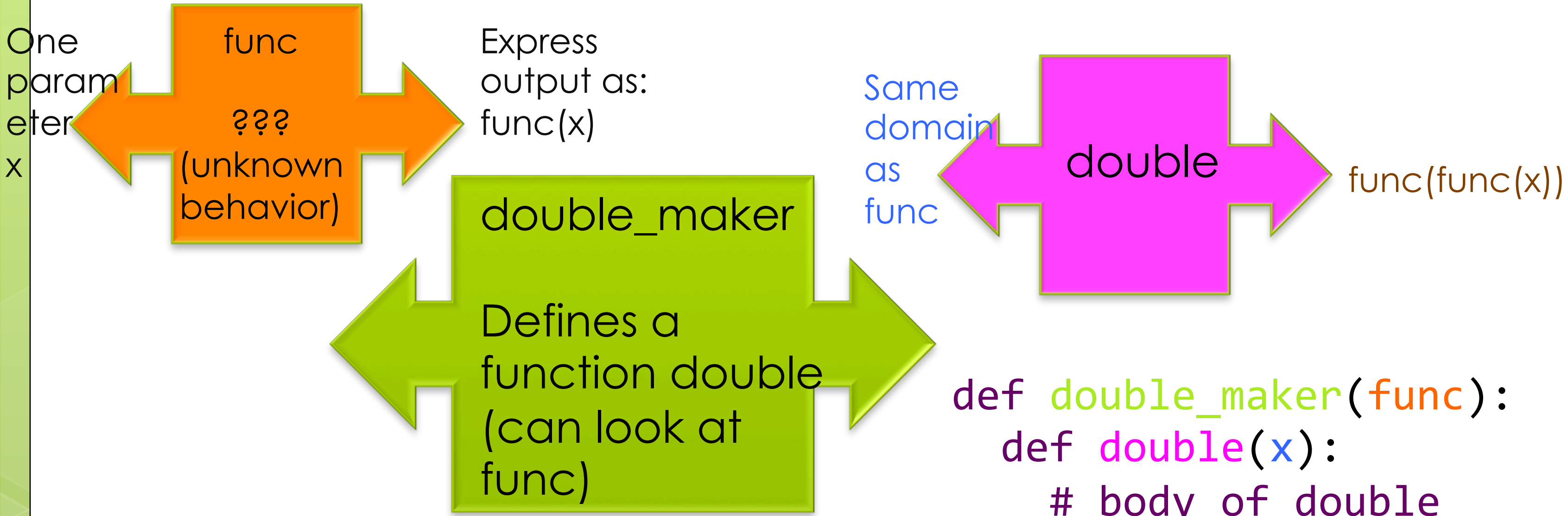
Same domain as func

double

func(func(x))

```
def double_maker(func):
  # body of double_maker
  return double
```

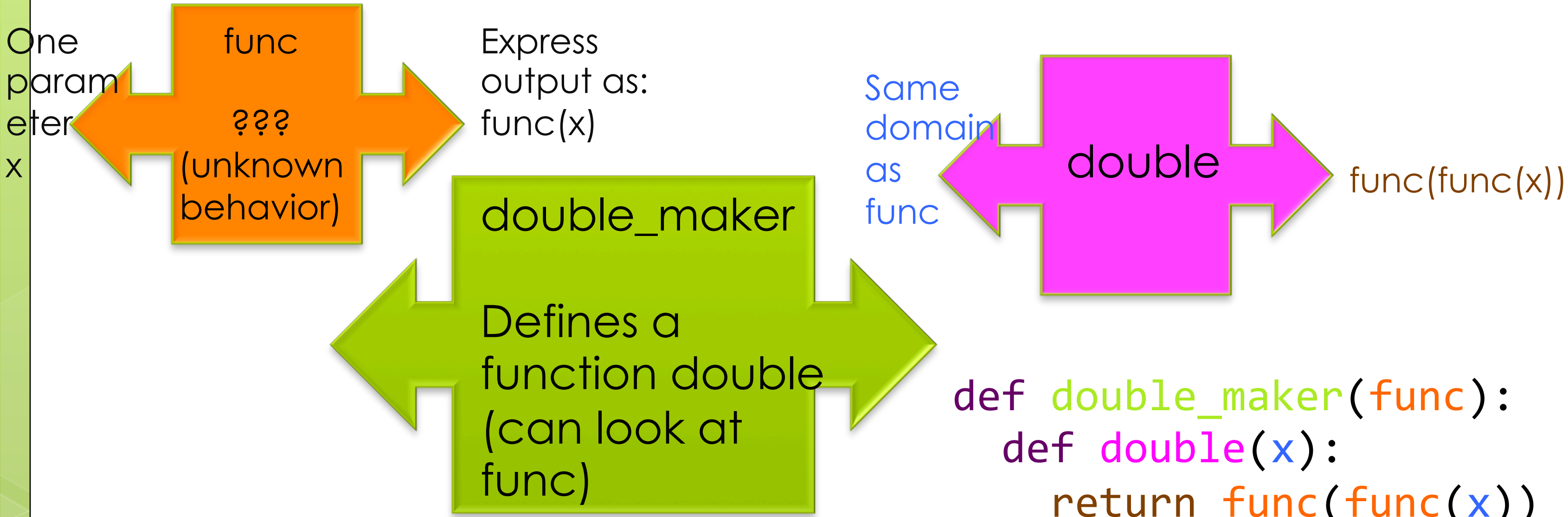
Characteristics of Functions



```
def double_maker(func):
  def double(x):
    # body of double
  return double
```

Even though getting the domain/range right might not seem to be important, it’s an important step in the right direction.
It’s like aiming your basketball shot. An errant shot has no chance of going in. A straight shot always has more of a chance.

Characteristics of Functions



One param eter x

func

???
(unknown behavior)

Express output as: func(x)

double_maker

Defines a function double (can look at func)

Same domain as func

double

func(func(x))

```
def double_maker(func):
  def double(x):
    return func(func(x))
  return double
```

double_maker can now make functions that we can customize. It combines the power of argument passing and function definitions!

Now: try writing call expressions for double_maker and its output and draw environment diagrams!

Recursion Important Q's

1. What is the desired behavior of your function?
2. What vars do you want to keep track of? (Input)
    1. Remember that your return value holds values too!
    2. If you have more than # of params, you need to define helper
    3. Where should each var start?
3. How can I reuse the function with smaller input?
4. How do I combine the result with the current pieces of information to solve the problem?
5. Base cases: When should we stop? What to return?

Thinking about these questions *before you code* will save you a lot of time.

You can see that once you answer these questions thoroughly, the code follows directly (watch the colors in the next few examples).

Goal

- Not to memorize the questions.
- Use the questions until you know what to look for in a problem.

Recursion Qs: Example

1 + 2 + 3 + … + n

1. What is the desired behavior of your function?

Sum up to n. The desired value can be expressed as sum_up_to(n)

2. What vars do you want to keep track of?

The current number being summed
Note: the return value takes care of storing the total

3. How can I reuse the function with simpler input?
4. How do I combine the result with the current pieces of information to solve the problem?
5. Base cases: When should we stop? What to return?

Recursion Qs: Example

1 + 2 + 3 + … + n

1. What is the desired behavior of your function?
2. What vars do you want to keep track of?
3. How can I reuse the function with simpler input?

Sum numbers up to (n-1)

4. How do I combine the result with the current pieces of information to solve the problem?

sum_up_to(n-1) + n

5. Base cases: When should we stop? What to return?

When n = 1, we know that the answer is just 1

Recursion Qs: Example

1. Desired behavior:

Sum up to n. Express as sum_up_to(n)

2. Vars to keep track of:

The current number being summed

3. Recursive call/reuse of func:

Sum numbers up to (n-1)

4. Building up from reuse:

sum_up_to(n-1) + n

5. When to end/base case:

When n = 1, we know that the answer is just 1

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

Recursion Qs: Example

```
def sum_up_to(n):
  “””Sum from 1 up to n”””
  if n == 1:
    return 1
  return n + sum_up_to(n–1)

def sum_up_to(n):
  if n == 1:
    return sum from 1 up to 1
  return n + sum from 1 up to (n-1)
```

Recursion Qs: Example

1 + 3 + 5 + … + n or 2 + 4 + 6 + … + n

1. What is the desired behavior of your function?

Sum every other to n. Expressed desired value as sum_every_other(n)

2. What vars do you want to keep track of?

The current number being summed
Note: the return value takes care of storing the total

3. How can I reuse the function with simpler input?
4. How do I combine the result with the current pieces of information to solve the problem?
5. Base cases: When should we stop? What to return?

Recursion Qs: Example

1 + 3 + 5 + … + n

1. What is the desired behavior of your function?
2. What vars do you want to keep track of?
3. How can I reuse the function with simpler input?

Sum numbers up to (n-2)

4. How do I combine the result with the current pieces of information to solve the problem?

sum_up_to(n-2) + n

5. Base cases: When should we stop? What to return?

When n < 2, we know that the answer is just n

Recursion Qs: Example

1. Desired behavior:

Sum every other to n. Express as sum_every_other(n)

2. Vars to keep track of:

The current number being summed

3. Recursive call/reuse of func:

Sum numbers up to (n-2)

4. Building up from reuse:

sum_up_to(n-2) + n

5. When to end/base case:

When n < 2, we know that the answer is just n

```
def sum_every_other(n):
  if n < 2:
    return n
  return n + \
    sum_every_other(n-2)
```

Recursion Qs: Example

```
def sum_every_other(n):
  “””Sum every other up to n”””
  if n < 2:
    return n
  return n + sum_every_other(n-2)

def sum_every_other(n):
  if n < 2:
    return sum every other up to 0, 1
  return n + sum every other up to (n-2)
```

Recursion Qs: Example

```
def cascade(n):
  “””Print a cascade of n (larger->smaller->larger)”””
  if n < 10:
    print(n)
  else:
    print(n)
    cascade(n // 10)
    print(n)
```

```
def cascade(n):
  if n < 10:
    print cascade of one digit
    (just the current number)
  else:
    print current number
    print cascade of curr num
          w/o ones digit
    print current number again
```

Recursion Qs: Example

```
def cascade(n):
  if n < 10:
    print cascade of one digit
    (just the current number)
  else:
    print current number
    print cascade of curr num
            w/o ones digit
    print current number again

cascade(123)
123
12
1            Solution to smaller problem expressed as
12           cascade(123 // 10) -> cascade(12)
123
```

Recursion Takeaways

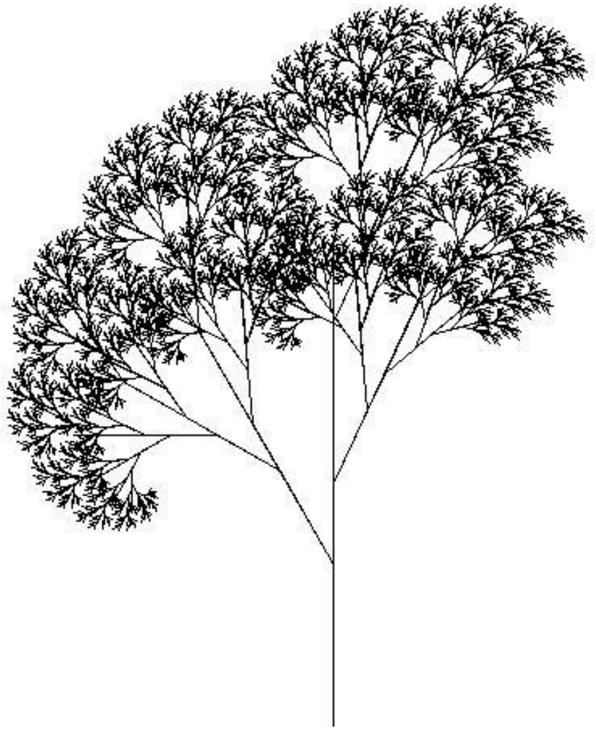
- Remember that all your output must match the function's range.
  - This means your base cases and regular calls. Everything!
  - E.g hailstone. You should always return the number of steps, no matter in base case or regular func.
- A good way to find base cases is to see where the input to your functions can be problematic.
  - E.g. sum_every_other(n-2) is problematic if n-2 < 0!
- Call expressions are expressive! You can describe "The 25th fibonacci number" easily as fib(25).
  - In fact, call expressions express **solutions** to the same problem you are trying to solve on smaller input

Break

Quote of the day: “the huge tables of square roots used by engineers and schoolchildren prior to the 1970s have now been replaced by a five-line program for Newton’s method running on electronic calculators” – CS188 textbook Artificial Intelligence: A Modern Approach (p. 47)

Iterative improvement is simple but powerful!

Tree Recursion



http://upload.wikimedia.org/wikipedia/commons/f/f7/RecursiveTree.JPG

Tree Recursion Important Q's

1. What is the desired behavior of your function?
2. What vars do you want to keep track of?
    1. Remember that your return value holds values too!
    2. If you have more than # of params, you need to define helper
    3. Where should each var start?
3. How can I reuse the function many times with smaller input?
4. How do I combine the result with the current pieces of information to solve the problem?
5. Base cases: When should we stop? What to return?

Thinking about these questions *before you code* will save you a lot of time.

You can see that once you answer these questions thoroughly, the code follows directly (watch the colors in the next few examples).

Tree Recursion Qs: Example

Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

1. What is the desired behavior of your function?

Fib(n) should be fib(n-1) + fib(n-2). Expressed as fib(n)

2. What vars do you want to keep track of?

The current number being summed
Note: the return value takes care of storing the total

3. How can I reuse the function with simpler input?
4. How do I combine the result with the current pieces of information to solve the problem?
5. Base cases: When should we stop? What to return?

Tree Recursion Qs: Example

Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

1. What is the desired behavior of your function?
2. What vars do you want to keep track of?
3. How can I reuse the function with simpler input?

To calculate fib(n), we need fib(n-1) and fib(n-2)

4. How do I combine the result with the current pieces of information to solve the problem?

fib(n-1) + fib(n-2)

5. Base cases: When should we stop? What to return?

When n < 2, we know that the answer is just n

Recursion Qs: Example

1. Desired behavior:

Fibonacci number defined by recurrence relation. Express as fib(n)

2. Vars to keep track of:

The current number being summed

3. Recursive call/reuse of func:

fib(n-1), fib(n-2)

4. Building up from reuse:

fib(n-1) + fib(n-2)

5. When to end/base case:

When n < 2, we know that the answer is just n. To be explicit, we can have if n == 0: return 0 ; if n == 1: return 1

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

What Makes Tree Recursion Special?

- [Class Discussion]
- Very easy to explore different branches, because now you can express sub-branches as call expressions
  - E.g. Give me values for fib(5) and fib(6)!
  - E.g. Give me ways to count change for $0.20 with just pennies or $0.15 with nickels + pennies.
    - count_change(20, [1]), count_change(15, [5, 1])
- Useful when you have to return a value based on the various sub-possibilities/sub-problems
  - E.g. make a decision or sum up sub-values
- Again, expressiveness of call expressions saves the day!

Break

Iteration vs Recursion

1. Every time you make a recursive call, you'll loop through the same code
    1. More convenient than iteration because parameters will take care of "name-updating" for you
    2. Also, you have the power to make multiple mini-loops (tree recursion)
2. Iteration uses a header to stop. Recursion uses base cases to stop.
3. Iterations has current frame to keep track of info. Recursion must use parameters + RV because a new frame is created for each recursive call

Iteration vs Recursion

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

Keep track:
  n – current number
  RV – total
Loop:
  Next iteration: n = n – 1
Stop/Loop until:
  When n == 0. Then return what
  the RV should start with

```
def fact_iter(n):
  total = 1
  while n > 0:
    total *= n
    n -= 1
  return total
```

Keep track:
  n – current number
  total – total
Loop:
  Next iteration: n = n – 1
Loop while:
  n > 0. Then return the accumulated result

Env Diagram Rules Source: Andrew Huang

Creating a function using def/lambda:

1. Draw func <name>(<param1>, <param2>, …) [parent=?]
2. The parent of the function is the current frame
3. For def only: bind this function to the same name in the current frame

Calling user-defined functions:

1. Evaluate the operator and operands
2. Create a new frame. Label the frame with the info from the function object <name> [parent=?]
3. Bind the formal parameters to the arguments (values of operands)
4. Evaluate the body with new frame as current frame
5. Return back to the frame that called the function

Assignment

1. Evaluate the RHS 2. Bind name on LHS to value on RHS in current frame

Lookup

1. Start at current frame. If name is there, great! If not, go to the **parent** frame. Keep going until you cannot find the name in the global frame.

Env Diagram Rules

1. Don’t follow your intuition, because you have never seen rules like these (unless you programmed before, kudos!)
2. Follow these rules until you get the right intuition.
3. Env diagrams are fun! They just take time to master.
4. Some sacrifice now will be rewarding in the long run ☺
5. Now let’s look at the rules again.

Env Diagram Rules Source: Andrew Huang

Creating a function using def/lambda:

1. Draw func <name>(<param1>, <param2>, …) [parent=?]
2. The parent of the function is the current frame
3. For def only: bind this function to the same name in the current frame

Calling user-defined functions:

1. Evaluate the operator and operands
2. Create a new frame. Label the frame with the info from the function object <name> [parent=?]
3. Bind the formal parameters to the arguments (values of operands)
4. Evaluate the body with new frame as current frame
5. Return back to the frame that called the function

Assignment

1. Evaluate the RHS 2. Bind name on LHS to value on RHS in current frame

Lookup

1. Start at current frame. If name is there, great! If not, go to the **parent** frame. Keep going until you cannot find the name in the global frame.

Midterm Tips

1. Try out simple examples with your code! Best debugging tool for simple code (and sanity check).
2. Understand our doctests and/or try to write your own doctests.
3. Do heavy lifting with what the problem's asking before you write code!
   1. Function boxes
   2. High-level pseudocode: e.g. print this, then make this recursive call, etc.
4. Write out your thoughts. It's really hard to remember everything in your head at once. Plus, once you review you test, you can see what thoughts you made on the exam.
5. Remember that concepts can be combined together. Practice combining concepts. It's like learning a language!
6. You cannot avoid knowing the rules for a CS class. Please practice the rules on simple examples. The same rules apply to easy and hard examples!

Optional Conceptual Questions (Wednesday 9/17 lecture)

1. In cascade, why does the first print function print in decreasing order and why does the second one print in increasing order? Please explain using an environment diagram.
   1. print(n)
   2. cascade(n // 10)
   3. print(n)
2. How would you summarize the behavior of cascade? Now write pseudocode for cascade, replacing the recursive call with the behavior summary you came up with. Compare it with the actual output.
3. What makes tree recursion a special case of recursion? Why is tree recursion useful?
4. In count_partitions, what are the two simpler instances of the problem? How can you express the solution to those simpler instances using call expressions with count_partitions?