CS 61A Structure and Interpretation of Computer Programs
Fall 2014

Midterm 1

INSTRUCTIONS

- You have 2 hours to complete the exam.
- The exam is closed book, closed notes, closed computer, closed calculator, except one hand-written 8.5" × 11" crib sheet of your own creation and the official 61A midterm 1 study guide attached to the back of this exam.
- Mark your answers ON THE EXAM ITSELF. If you are not sure of your answer you may wish to provide a *brief* explanation.

Last name	
First name	
SID	
Login	
TA & section time	
Name of the person to your left	
Name of the person to your right	
All the work on this exam is my own. **(please sign)**	

For staff use only

Q. 1	Q. 2	Q. 3	Q. 4	Total
/12	/14	/8	/6	/40

1. (12 points) World Cup

(a) (10 pt) For each of the expressions in the tables below, write the output displayed by the interactive Python interpreter when the expression is evaluated. The output may have multiple lines.

Whenever the interpreter would report an error, write Error. You *should* include any lines displayed before an error.

Reminder: the interactive interpreter displays the value of a successfully evaluated expression, unless it is `None`.

The first three rows have been provided as examples.

Assume that you have started Python 3 and executed the following statements:

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

def argentina(n):
    print(n)
    if n > 0:
        return lambda k: k(n+1)
    else:
        return 1 / n

def germany(n):
    if n > 1:
        print('hallo')
    if argentina(n-2) >= 0:
        print('bye')
    return argentina(n+2)
```

Expression	Interactive Output
`5*5`	25
`print(5)`	5
`1/0`	Error
`print(1, print(2))`	
`argentina(0)`	

Expression	Interactive Output
`argentina(1)(square)`	
`germany(1)(square)`	
`germany(2)(germany)`	

(b) (2 pt) Fill in the blank with an expression so that the whole expression below evaluates to a number.
Hint: The expression `abs > 0` causes a `TypeError`.

```
(lambda t: argentina(t)(germany)(square))(_________________________________)
```

2. (14 points) Envy, Iron, Mint

(a) (6 pt) Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. *You may not need to use all of the spaces or frames.*

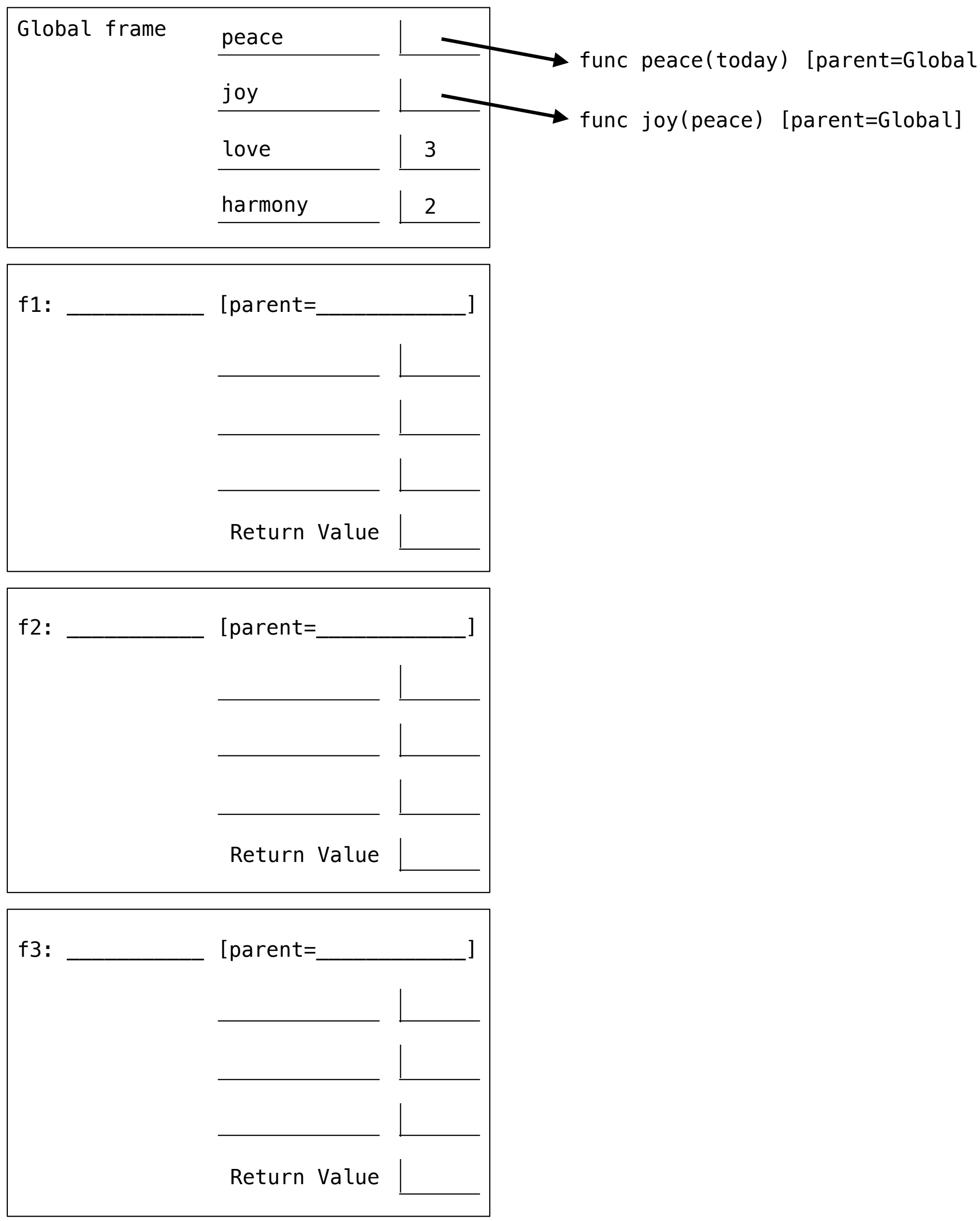
A complete answer will:

- Add all missing names, labels, and parent annotations to all local frames.
- Add all missing values created during execution.
- Show the return value for each local frame.

```
def peace(today):
    harmony = love+2
    return harmony + today(love+1)

def joy(peace):
    peace, love = peace+2, peace+1
    return love // harmony

love, harmony = 3, 2
peace(joy)
```



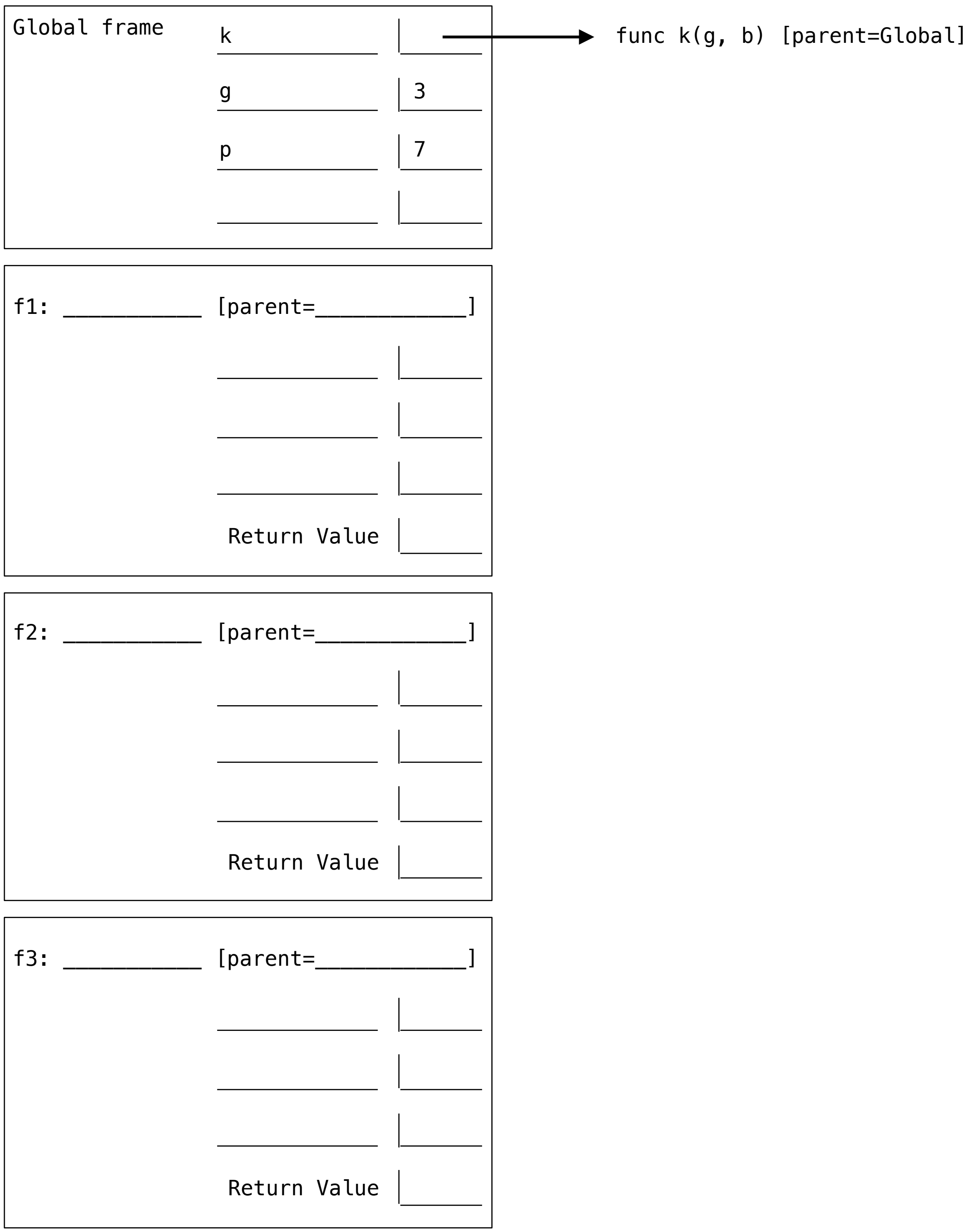
(b) **(8 pt)** Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. *You may not need to use all of the spaces or frames.*

A complete answer will:

- Add all missing names, labels, and parent annotations to all local frames.
- Add all missing values created during execution.
- Show the return value for each local frame.

```
def k(g, b):
    def n(s, a):
        return g-p
    return b(n(b, p))

g, p = 3, 7
k(p+1, lambda s: g+3)
```



3. (8 points) Express Yourself

(a) (3 pt) A k-bonacci sequence starts with K-1 zeros and then a one. Each subsequent element is the sum of the previous K elements. The 2-bonacci sequence is the standard Fibonacci sequence. The 3-bonacci and 4-bonacci sequences each start with the following ten elements:

```
               n: 0, 1, 2, 3, 4, 5, 6,  7,  8,  9, ...

kbonacci(n, 2): 0, 1, 1, 2, 3, 5, 8, 13, 21, 35, ...

kbonacci(n, 3): 0, 0, 1, 1, 2, 4, 7, 13, 24, 44, ...

kbonacci(n, 4): 0, 0, 0, 1, 1, 2, 4,  8, 15, 29, ...
```

Fill in the blanks of the implementation of `kbonacci` below, a function that takes non-negative integer `n` and positive integer `k` and returns element n of a k-bonacci sequence.

```
def kbonacci(n, k):
    """Return element N of a K-bonacci sequence.

    >>> kbonacci(3, 4)
    1
    >>> kbonacci(9, 4)
    29
    >>> kbonacci(4, 2)
    3
    >>> kbonacci(8, 2)
    21
    """

    if n < k - 1:

        return 0

    elif n == k - 1:

        return 1

    else:

        total = 0


        i = ____________________________________________________________________

        while i < n:


            total = total + ___________________________________________________

            i = i + 1

        return total
```

(b) **(5 pt)** Fill in the blanks of the following functions defined together in the same file. **Assume that all arguments to all of these functions are positive integers that do not contain any zero digits**. For example, 1001 contains zero digits (not allowed), but 1221 does not (allowed). You may assume that `reverse` is correct when implementing `remove`.

```
def combine(left, right):
    """Return all of LEFT's digits followed by all of RIGHT's digits."""
    factor = 1
    while factor <= right:
        factor = factor * 10
    return left * factor + right

def reverse(n):
    """Return the digits of N in reverse.

    >>> reverse(122543)
    345221
    """

    if n < 10:

        return n

    else:

        return combine(_________________________ , __________________________)


def remove(n, digit):
    """Return all digits of N that are not DIGIT, for DIGIT less than 10.

    >>> remove(243132, 3)
    2412
    >>> remove(243132, 2)
    4313
    >>> remove(remove(243132, 1), 2)
    433
    """

    removed = 0

    while n != 0:


        ____________, ____________ = ____________________, ____________________

        if ___________________________________________________________________:

            removed = ________________________________________________________

    return reverse(removed)
```

4. (6 points) Lambda at Last

(a) (2 pt) Fill in the blank below with an expression so that the second line evaluates to 2014. **You may only use the names `two_thousand`, `two`, `k`, `four`, and `teen` and parentheses in your expression (no numbers, operators, etc.)**.

```
two_thousand = lambda two: lambda k: ___________________________________________
two_thousand(7)(lambda four: lambda teen: 2000 + four + teen)
```

(b) (4 pt) The `if_fn` returns a two-argument function that can be used to select among alternatives, similar to an `if` statement. Fill in the return expression of `factorial` so that it is defined correctly for non-negative arguments. **You may only use the names `if_fn`, `condition`, `a`, `b`, `n`, `factorial`, `base`, and `recursive` and parentheses in your expression (no numbers, operators, etc.)**.

```
def if_fn(condition):
    if condition:
        return lambda a, b: a
    else:
        return lambda a, b: b

def factorial(n):
    """Compute N! for non-negative N.  N! = 1 * 2 * 3 * ... * N.

    >>> factorial(3)
    6
    >>> factorial(5)
    120
    >>> factorial(0)
    1
    """
    def base():
        return 1
    def recursive():
        return n * factorial(n-1)


    return ____________________________________________________________________
```

Scratch Paper

Scratch Paper

Scratch Paper

Import statement

```
from math import pi
tau = 2 * pi
```

Assignment statement

Global frame: pi 3.1416

Name | Binding | Value

Code (left):

Statements and expressions
Red arrow points to next line.
Gray arrow points to the line just executed

Frames (right):

A name is bound to a value

In a frame, there is at most one binding per name

```
from operator import mul
def square(x):
    return mul(x, x)
square(-2)
```

Global frame
mul → func mul(...) [parent=Global] — Built-in function
square → func square(x) [parent=Global] — User-defined function

Intrinsic name of function called

f1: square [parent=Global] — Local frame
x -2 — Formal parameter bound to argument
Return value 4 — Return value is not a binding!

```
from operator import mul
def square(x):
    return mul(x, x)
square(square(3))
```

Global frame: mul, square
f1: square [parent=Global] x 3, Return value 9
f2: square [parent=Global] x 9, Return value 81

A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

208
mul(add(2, mul(4, 6)), add(3, 5))
mul | 26 add(2, mul(4, 6)) | 8 add(3, 5)
add | 2 | 24 mul(4, 6) | add | 3 | 5
mul | 4 | 6

Pure Functions

-2 ▶ *abs*(number): ▶ 2

2, 10 ▶ *pow*(x, y): ▶ 1024

Non-Pure Functions

-2 ▶ *print*(...): ▶ None

display "-2"

Defining:

```
>>> def square( x ):
        return mul(x, x)
```

Formal parameter
Return expression
Def statement
Body (*return statement*)

Call expression: square(2+2)

operand: 2+2
argument: 4

operator: square
function: func square(x)

Calling/Applying: 4 ▶ *square*(x): return mul(x, x) ▶ 16

Argument
Intrinsic name
Return value

Compound statement | Clause

<header>:
    <statement>
    <statement>
    ...
<separating header>:
    <statement>
    <statement>
    ...
...

Suite

```
def abs_value(x):
    if x > 0:
        return x
    elif x == 0:
        return 0
    else:
        return -x
```

1 statement,
3 clauses,
3 headers,
3 suites,
2 boolean contexts

```
def f(x, y):
    return g(x)

def g(a):
    return a + y

result = f(1, 2)
```

"y" is not found
Error
"y" is not found

Global frame
f → func f(x, y) [parent=Global]
g → func g(a) [parent=Global]
f1: f [parent=Global] x 1, y 2
f2: g [parent=Global] a 1

- An environment is a sequence of frames
- An environment for a non-nested function (no def within def) consists of one local frame, followed by the global frame

Evaluation rule for call expressions:

1. Evaluate the operator and operand subexpressions.
2. Apply the function that is the value of the operator subexpression to the arguments that are the values of the operand subexpressions.

Applying user-defined functions:

1. Create a new local frame with the same parent as the function that was applied.
2. Bind the arguments to the function's formal parameter names in that frame.
3. Execute the body of the function in the environment beginning at that frame.

Execution rule for def statements:

1. Create a new function value with the specified name, formal parameters, and function body.
2. Its parent is the first frame of the current environment.
3. Bind the name of the function to the function value in the first frame of the current environment.

Execution rule for assignment statements:

1. Evaluate the expression(s) on the right of the equal sign.
2. Simultaneously bind the names on the left to those values, in the first frame of the current environment.

Execution rule for conditional statements:

Each clause is considered in order.
1. Evaluate the header's expression.
2. If it is a true value, execute the suite, then skip the remaining clauses in the statement.

Evaluation rule for or expressions:

1. Evaluate the subexpression <left>.
2. If the result is a true value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

Evaluation rule for and expressions:

1. Evaluate the subexpression <left>.
2. If the result is a false value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

Evaluation rule for not expressions:

1. Evaluate <exp>; The value is True if the result is a false value, and False otherwise.

Execution rule for while statements:

1. Evaluate the header's expression.
2. If it is a true value, execute the (*whole*) suite, then return to step 1.

```
from operator import mul
def square(square):
    return mul(square, square)
square(4)
```

A | B

1 A Global frame: mul, square
2 B
1 f1: square [parent=Global] square 4, Return value 16

A call expression and the body of the function being called are evaluated in different environments

```
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1   # Zeroth and first Fibonacci numbers
    k = 1               # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```

```
def cube(k):
    return pow(k, 3)

def summation(n, term):
    """Sum the first n terms of a sequence.

    >>> summation(5, cube)
    225
    """
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total
```

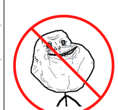
Function of a single argument (not called term)

A formal parameter that will be bound to a function

The cube function is passed as an argument value

The function bound to term gets called here

$0 + 1^3 + 2^3 + 3^3 + 4^3 + 5^5$



Higher-order function: A function that takes a function as an argument value or returns a function as a return value

Nested def statements: Functions defined within other function bodies are bound to names in the local frame

```
square = lambda x,y: x * y
```

Evaluates to a function. No "return" keyword!

A function
with formal parameters x and y
that returns the value of "x * y"

Must be a single expression

```
def make_adder(n):
    """Return a function that takes one argument k and returns k + n.

    >>> add_three = make_adder(3)
    >>> add_three(4)
    7
    """
    def adder(k):
        return k + n
    return adder
```

A function that returns a function

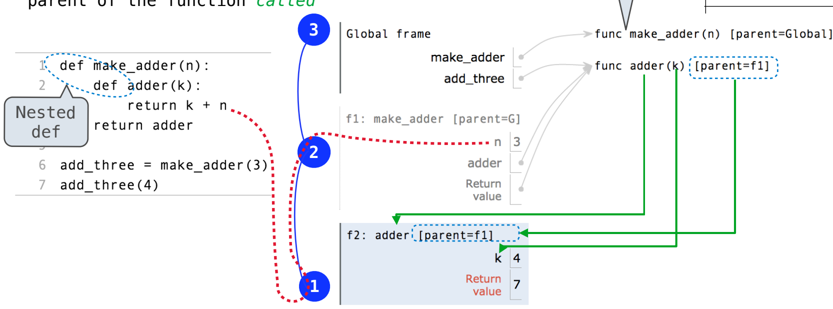
The name add_three is bound to a function

A local def statement

Can refer to names in the enclosing function

- Every user-defined **function** has a ***parent frame*** (often global)
- The parent of a **function** is the frame in which it was *defined*
- Every local **frame** has a *parent* ***frame*** (often global)
- The parent of a **frame** is the parent of the function *called*

A function's signature has all the information to create a local frame




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

VS

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

- Both create a function with the same domain, range, and behavior.
- Both functions have as their parent the environment in which they were defined.
- Both bind that function to the name square.
- Only the def statement gives the function an intrinsic name.

When a function is defined:

1. Create a **function value**: func *<name>*(*<formal parameters>*)
2. Its parent is the current frame.

   f1: make_adder   func adder(k) [parent=f1]

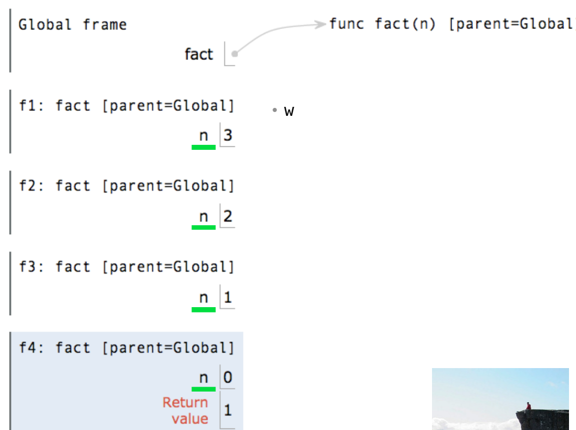
3. Bind *<name>* to the **function value** in the current frame (which is the first frame of the current environment).

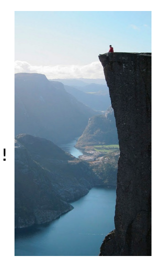
When a function is called:

1. Add a **local frame**, titled with the *<name>* of the function being called.
2. Copy the parent of the function to the **local frame**: [parent=*<label>*]
3. Bind the *<formal parameters>* to the arguments in the **local frame**.
4. Execute the body of the function in the environment that starts with the **local frame**.

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

fact(3)
```





Is fact implemented correctly?

1. Verify the base case.
2. Treat fact as a functional abstraction!
3. Assume that fact(n-1) is correct.
4. Verify that fact(n) is correct, assuming that fact(n-1) correct.

```
def curry2(f):
    """Returns a function g such that g(x)(y) returns f(x, y)."""
    def g(x):
        def h(y):
            return f(x, y)
        return h
    return g
```

Currying: Transforming a multi-argument function into a single-argument, higher-order function.

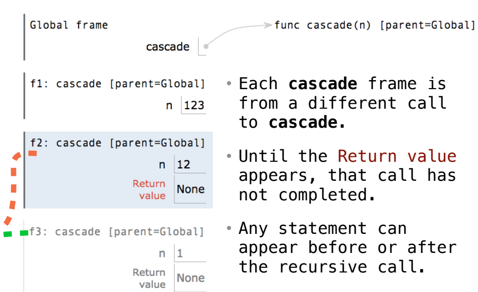
Anatomy of a recursive function:

- The **def statement header** is similar to other functions
- Conditional statements check for **base cases**
- Base cases are evaluated **without recursive calls**
- Recursive cases are evaluated **with recursive calls**

```
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = n // 10, n % 10
        return sum_digits(all_but_last) + last
```

```
def cascade(n):
    if n < 10:
        print(n)
    else:
        print(n)
        cascade(n//10)
        print(n)

cascade(123)
```



Program output:

```
123
12
1
12
```

- Each **cascade** frame is from a different call to **cascade.**
- Until the Return value appears, that call has not completed.
- Any statement can appear before or after the recursive call.

- Recursive decomposition: finding simpler instances of a problem.
- E.g., **count_partitions(6, 4)**
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
- Solve two simpler problems:
  - **count_partitions(2, 4)**
  - **count_partitions(6, 3)**
- Tree recursion often involves exploring different choices.

```
def count_partitions(n, m):
    if n == 0:
        return 1
    elif n < 0:
        return 0
    elif m == 0:
        return 0
    else:
        with_m = count_partitions(n-m, m)
        without_m = count_partitions(n, m-1)
        return with_m + without_m
```

```
1
12
123
1234
123
12
1
```

```
def inverse_cascade(n):
    grow(n)
    print(n)
    shrink(n)

def f_then_g(f, g, n):
    if n:
        f(n)
        g(n)

grow =   lambda n: f_then_g(grow,  print,  n//10)
shrink = lambda n: f_then_g(print, shrink, n//10)
```

n:	0,	1,	2,	3,	4,	5,	6,	7,	8,
fib(n):	0,	1,	1,	2,	3,	5,	8,	13,	21,

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

```
from operator import floordiv, mod
def divide_exact(n, d):
    """Return the quotient and remainder of dividing N by D.

    >>> q, r = divide_exact(2012, 10)
    >>> q
    201
    >>> r
    2
    """
    return floordiv(n, d), mod(n, d)
```

Multiple assignment to two names

Multiple return values, separated by commas