CS 61A Structure and Interpretation of Computer Programs
Spring 2013

Midterm 2

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 2 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	Q. 5	Total
/15	/12	/6	/6	/11	/50

1. (15 points) You Will Be Baked. And Then There Will Be Cake.

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

```
the_cake = [1, 2, [3], 4, 5]
a_lie = the_cake[1:4]
the_cake = the_cake[1:4]
great = a_lie
delicious = the_cake
moist = great[:-1]
```

For each of the following expressions, write the value to which it evaluates. If the value is a method value, write METHOD. If it is a function value, write FUNCTION. If evaluation causes an error, write ERROR. If evaluation would run forever, write FOREVER. Otherwise, write the resulting value as the interactive interpreter would display it.

Expression	**Evaluates to**
`the_cake`	
`the_cake is a_lie`	
`the_cake == great`	
`the_cake is delicious`	
`the_cake == moist + 4`	
`the_cake.append`	
`the_cake.append == a_lie.append`	
`the_cake[1] is a_lie[1]`	

(b) The following is the recursive list abstract data type from lecture:

```
empty_rlist = None

def rlist(first, rest):
    """Creates an rlist from the element first and the rlist rest."""
    return (first, rest)

def first(s):
    """Returns the first element of the rlist s"""
    return s[0]

def rest(s):
    """Returns the rest (itself an rlist) of s."""
    return s[1]

def len_rlist(s):
    """Returns the length of the rlist s."""
    if s == empty_rlist:
        return 0
    return 1 + len_rlist(rest(s))

def getitem_rlist(s, i):
    """Returns the element at index i in rlist s."""
    if i == 0:
        return first(s)
    return getitem_rlist(rest(s), i - 1)
```

For each of the following pieces of code, circle **Y** if the code contains at least one data abstraction violation, and **N** if the code contains no data abstraction violations. **Do not guess; leave the answer blank if you do not know it.** We will award one point for each correct answer, no points for an incorrect answer, and 0.5 points for each answer left blank.

Y	**N**	`rlist(4, rlist(5, None))`
Y	**N**	`rlist(1, (2, (3, empty_rlist)))`
Y	**N**	`rlist(rlist(1, empty_rlist), rlist(2, empty_rlist))`
Y	**N**	`first(rest( (1, (2, (3, empty_rlist))) ))`

Y	**N**	`x = rlist(5, rlist( (4, 3, 2), rlist(1, empty_rlist)))`
		`first(rest(x))[1]`

Y	**N**	`rlist(empty_rlist, empty_rlist)`
Y	**N**	`len(rlist(3, rlist(4, empty_rlist)))`

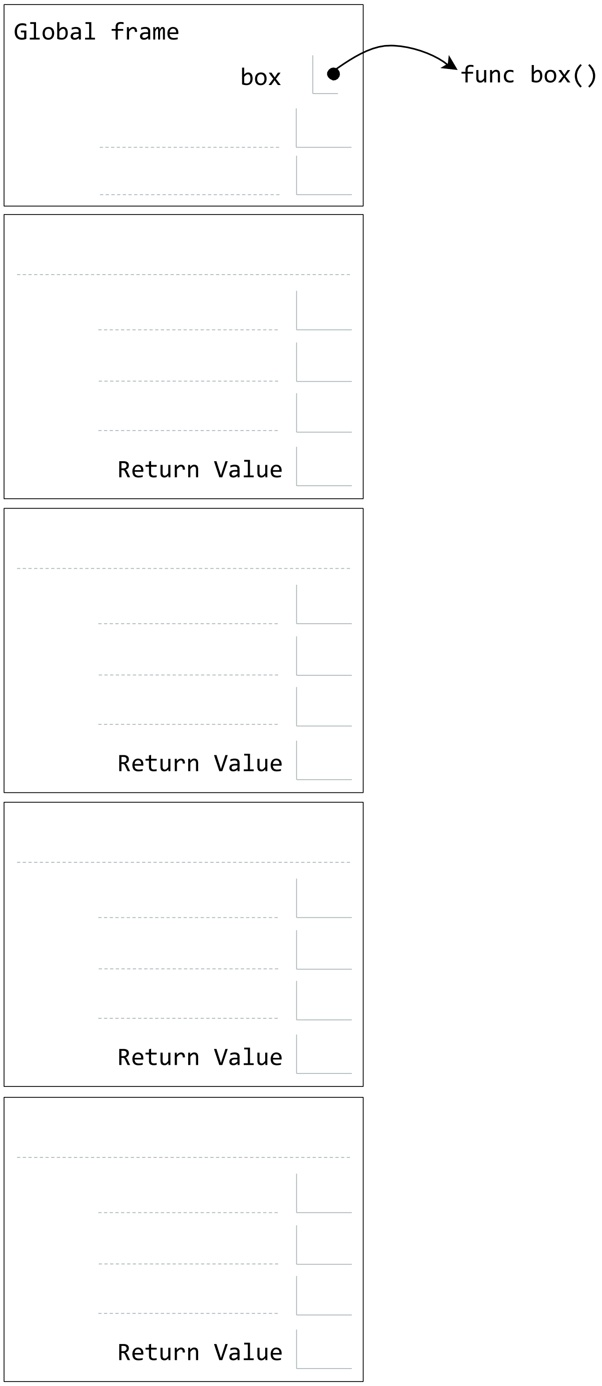
2. (12 points) Environmental Disaster

(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 need only show the final state of each frame. *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 box():
    toy = 0
    def disp(f, box):
        nonlocal toy
        if f == 0:
            toy += box
        elif f == 1:
            toy *= box
        return toy
    return disp
toybox = box()
toybox(0, 2)
toybox(0, 3)
toybox(1, 4)
```

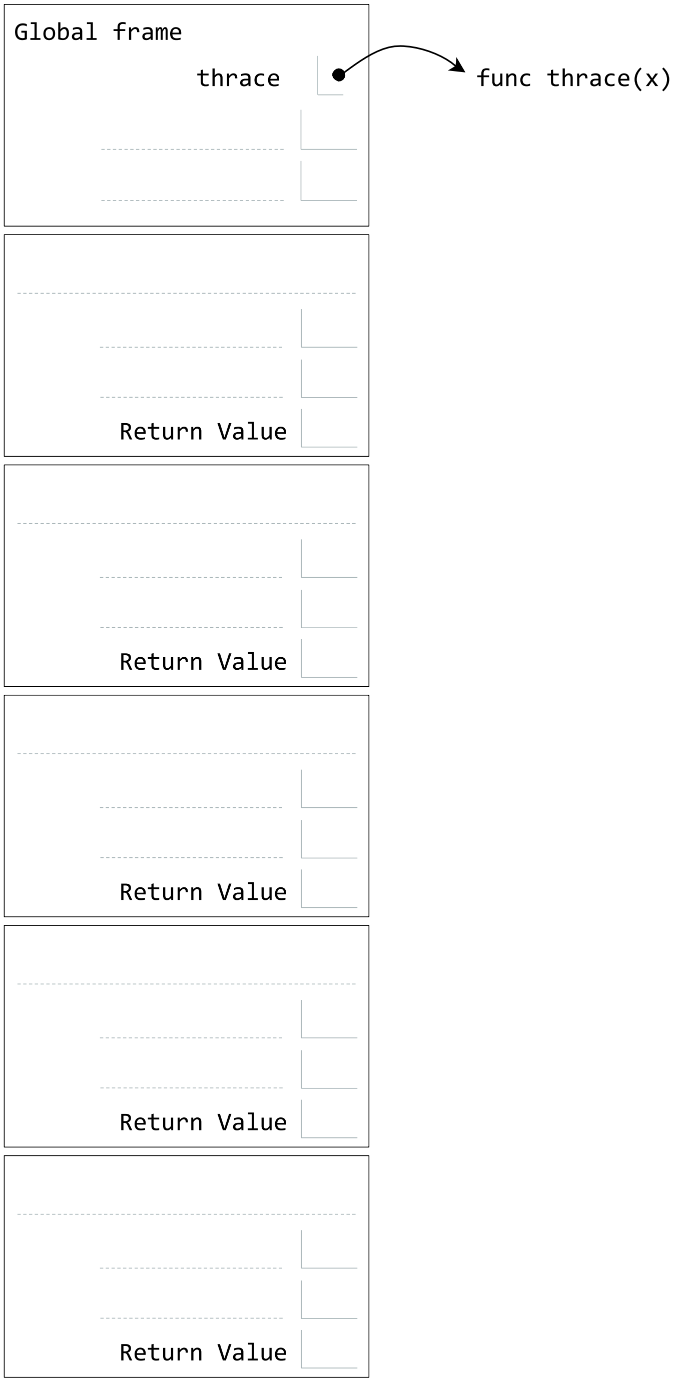


(b) **(6 pt)** Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. You need only show the final state of each frame. *You may not need to use all of the spaces or frames.*

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 thrace(x):
    def star(y):
        print(x, y)
        return x(y)
    return star

def kara(x):
    if x > 7:
        return x
    return kara(x * 2)

kara = thrace(kara)
buck = kara(5)
```



3. (6 points) Cue the Queue that Starts with a Q

For each of the following, cross out any incorrect or unnecessary lines in the following code so that the doctests pass for both classes. **Do not cross out class declarations, doctests, or docstrings.** You can cross out anything else, including method declarations, and your final code should make as much use of inheritance as possible. **Make sure to cross out the entire line for anything you wish to remove.**

Note: The `pop` method of a `list` removes the item at the given position and returns it.

(a)
```
class Queue(object):
    """Creates a Queue, which is like a list that supports 2
    operations: enqueue (adding an item to the back of the queue) and
    dequeue (removing an item from the front of the queue).

    >>> q = Queue()
    >>> q.enqueue(5)
    >>> q.enqueue(3)
    >>> q.enqueue(2)
    >>> q.dequeue()
    5
    """

    self.items = []

    def __init__(self, items):

    def __init__(self):

        self.enqueue(items)

        self.items = []

    def enqueue(item):

    def enqueue(self, item):

        items.append(self, item)

        self.items.append(item)

        items += item

        self.items += item

    def dequeue():

    def dequeue(self):

        self.items.pop(0)

        self.items.pop(len(self.items) - 1)

        return self.items.pop(0)

        return self.items.pop(len(self.items) - 1)
```

(b)
```
class PriorityQueue(Queue):
    """A PriorityQueue is like a sorted list that supports two
    operations: enqueue (adding an item to the PriorityQueue) and
    dequeue (removing the smallest item from the PriorityQueue).

    >>> p = PriorityQueue()
    >>> p.enqueue(5)
    >>> p.enqueue(3)
    >>> p.enqueue(2)
    >>> p.dequeue()
    2
    """

    self.items = []

    def __init__(self, items):

    def __init__(self):

        Queue.__init__()

        Queue.__init__(self)

        PriorityQueue.__init__()

        PriorityQueue.__init__(self)

        self.items = []

        self.items.sort()

    def enqueue(item):

    def enqueue(self, item):

        self.enqueue(item)

        Queue.enqueue(self, item)

        PriorityQueue.enqueue(self, item)

        items.append(self, item)

        self.items.append(item)

        items += item

        self.items += item

        self.items.sort()

    def dequeue():

    def dequeue(self):

        return self.dequeue()

        return Queue.dequeue(self)

        return PriorityQueue.dequeue(self)
```

4. (6 points) Prime RBIs

The Cal Mathletic Club is a group of math enthusiasts who compete in mathematical competitions. Since a sharp mind requires a sharp body, they also field an intramural baseball team. Unfortunately, this team has not been very good in recent years. In fact, they only had 2 runs batted in (RBIs) in all of 2010! The next two years were nearly as dreadful, with 3 RBIs in 2011, and 5 RBIs in 2012.

Being mathletes, they notice that their RBI totals have been consecutive prime numbers in each of the last three years. Being mathletes, they decide they should continue this trend, slowly improving their play each year by batting in the next prime number of runs.

Help the mathletes to determine their long-term goals by writing a higher-order function `make_prime_generator` that returns a function to generate primes. The latter function should return 2 the first time it is called, 3 the next time, then 5, 7, 11, and so on, returning the next prime number each time it is called.

(For the non-mathletic, a prime number can be defined as an integer greater than 1 that is not divisible by any other integer greater than 1. Thus, a prime number *p*'s only positive divisors are 1 and *p*.)

```
def make_prime_generator():
    """Return a function that computes the next prime number each time it
    is called.

    >>> gen = make_prime_generator()
    >>> gen(), gen(), gen()
    (2, 3, 5)
    >>> [gen() for _ in range(10)]
    [7, 11, 13, 17, 19, 23, 29, 31, 37, 41]
    """
```

5. (11 points) Mutation: It is the Key to Our Evolution

The following is an object-oriented recursive list implementation:

```
class Rlist(object):
    """A recursive list consisting of a first element and the rest."""
    class EmptyList(object):
        def __len__(self):
            return 0
    empty = EmptyList()

    def __repr__(self):
        f = repr(self.first)
        if self.rest is Rlist.empty:
            return ’Rlist({0})’.format(f)
        else:
            return ’Rlist({0}, {1})’.format(f, repr(self.rest))

    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest

    def __len__(self):
        return 1 + len(self.rest)

    def __getitem__(self, i):
        if i == 0:
            return self.first
        return self.rest[i - 1]
```

(a) Implement a `mutating_map` method that takes in a function and applies it to each element in an `Rlist`. This method should mutate the list in place, replacing each element with the result of applying the function to it. Do not create any new objects. You may assume that the input `Rlist` contains at least one element.

```
def mutating_map(self, fn):
    """Mutate this Rlist by applying fn to each element.

    >>> r = Rlist(1, Rlist(2, Rlist(3)))
    >>> r.mutating_map(lambda x: x + 1)
    >>> r
    Rlist(2, Rlist(3, Rlist(4)))
    """
```

(b) The *sieve of Eratosthenes* is an ancient algorithm for finding prime numbers. It starts with a sequence of numbers between 2 and n, in order. The first number is a prime, and the algorithm removes all larger multiples of that number from the sequence. Then the next remaining number is a prime, and the algorithm removes all larger multiples of that number from the sequence, and so on, until the end of the sequence is reached. At that point, all remaining numbers in the sequence are prime.

Here is a more concrete illustration of this process:

Initial sequence:	2, 3, 4, 5, 6, 7, 8, 9, 10
Remove larger multiples of 2:	2, 3, 5, 7, 9
Remove larger multiples of 3:	2, 3, 5, 7
Remove larger multiples of 5:	2, 3, 5, 7
Remove larger multiples of 7:	2, 3, 5, 7
Done.	

In this problem, you will implement this algorithm on `Rlist`s. Assume that you have `map_rlist` and `filter_rlist` functions with the following signatures and docstrings:

```
def map_rlist(s, fn):
    """Return an Rlist resulting from mapping fn over the elements of s.

    >>> map_rlist(Rlist(1, Rlist(2, Rlist(3))), lambda x: x * x)
    Rlist(1, Rlist(4, Rlist(9)))
    """

def filter_rlist(s, fn):
    """Filter the elements of s by predicate fn.

    >>> filter_rlist(Rlist(1, Rlist(2, Rlist(3))), lambda x: x % 2 == 1)
    Rlist(1, Rlist(3))
    """
```

i. First, write a function `sequence_to_rlist` that converts a Python sequence into an `Rlist`. Elements in the resulting `Rlist` should be in the same order as in the original sequence.

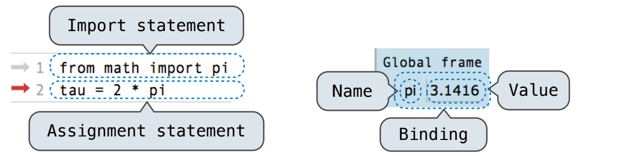
```
def sequence_to_rlist(seq):
    """Converts a sequence to an Rlist, preserving order.

    >>> sequence_to_rlist((3, 2, 1))
    Rlist(3, Rlist(2, Rlist(1)))
    """
```

ii. Now fill in the following function `prime_sieve` that implements the sieve of Eratosthenes algorithm. This function takes in an `Rlist` of numbers between 2 and n and removes all composite numbers from the `Rlist`. You may assume that the input `Rlist` has at least one element. You may leave the last line blank if you do not need it.

```
def prime_sieve(rlst):
    """Remove all composite numbers from the input Rlist. Assumes
    that the input contains the numbers from 2 to len(rlst), in
    order.

    >>> seq = sequence_to_rlist(range(2, 15))
    >>> prime_sieve(seq)
    >>> seq
    Rlist(2, Rlist(3, Rlist(5, Rlist(7, Rlist(11, Rlist(13))))))
    """

    while rlst.rest != ____________________________________________:

        func = lambda x: ___________________________________________

        rlst.rest = ________________________________________________

        rlst = rlst.rest

    ________________________________________________________________
```

(This page intentionally left blank)

CS 61A Midterm 1 Study Guide – Page 1



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

Code:

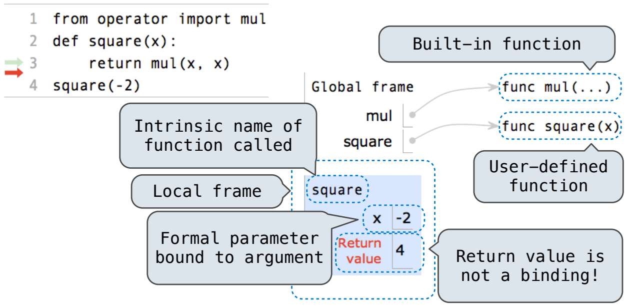
Statements and expressions

Red arrow points to next line.
Gray arrow points to the line just executed

Frames:

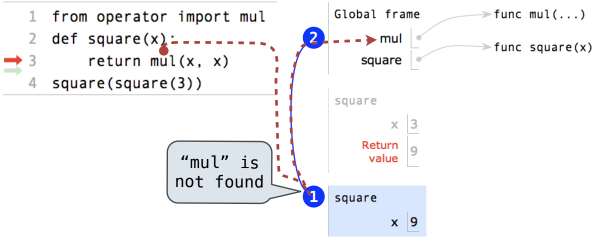
A name is bound to a value

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



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

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



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

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:

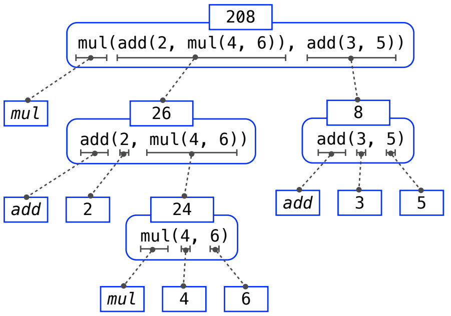
1. Evaluate the subexpression <left>.
2. If the result is a false value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

Evaluation rule for not expressions:

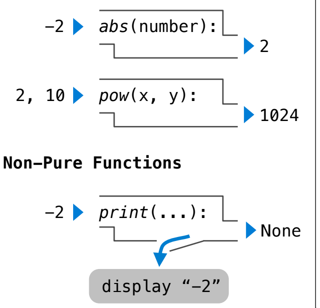
1. Evaluate <exp>; The value is True if the result is a false value, and False otherwise.

Execution rule for while statements:

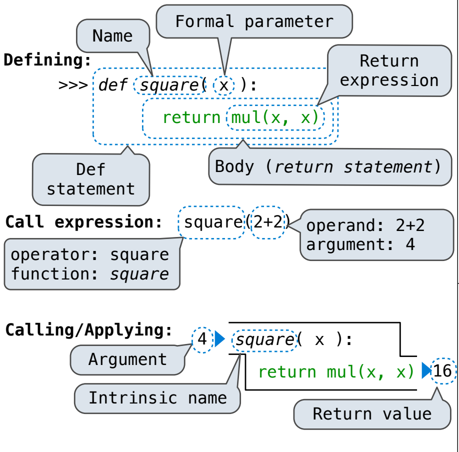
1. Evaluate the header's expression.
2. If it is a true value, execute the (*whole*) suite, then return to step 1.

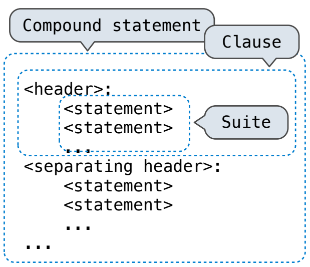


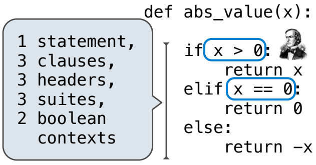
Pure Functions



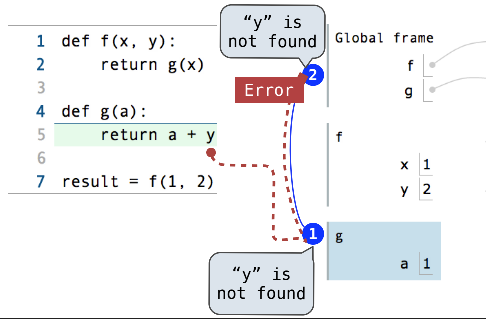
Non-Pure Functions





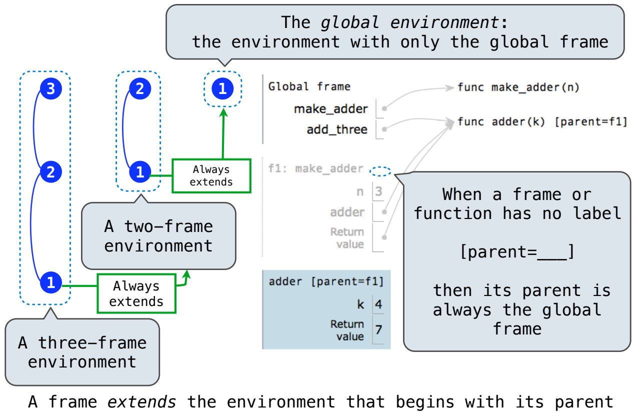


```
def abs_value(x):
    if x > 0:
        return x
    elif x == 0:
        return 0
    else:
        return -x
```



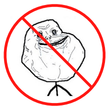
```
def f(x, y):
    return g(x)

def g(a):
    return a + y

result = f(1, 2)
```

- 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



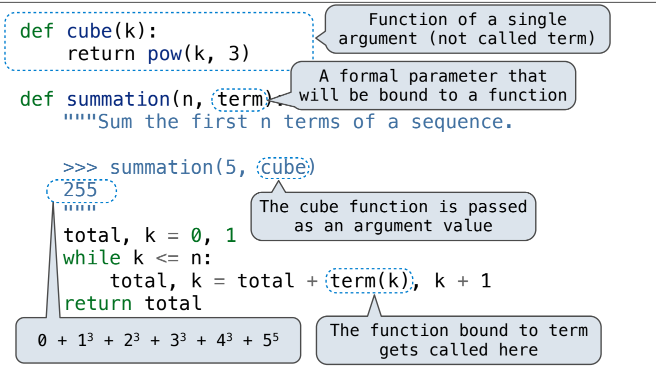
The global environment: the environment with only the global frame
3
2
1
2
1
1
Always extends
A two-frame environment
Always extends
A three-frame environment
Global frame
make_adder
add_three
func make_adder(n)
func adder(k) [parent=f1]
f1: make_adder
n 3
adder
Return value
adder [parent=f1]
k 4
Return value 7
When a frame or function has no label [parent=___] then its parent is always the global frame

A frame *extends* the environment that begins with its parent



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



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

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

    >>> summation(5, cube)
    255
    """
    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
0 + 1³ + 2³ + 3³ + 4³ + 5³
The function bound to term gets called here

CS 61A Midterm 1 Study Guide – Page 2

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

A function
with formal parameters x and y
and body "return x * y"

Must be a single expression

Facts about print
- Non-pure function
- Returns None
- Multiple arguments are printed with a space between them

```
>>> print(4, 2)
4 2
```

```
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

```
make_adder(1)(2)

make_adder(1)   (   2   )
```

Operator

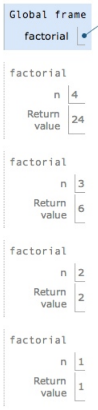
An expression that evaluates to a function value

Operand 0

An expression that evaluates to any value

```
def square(x):            def sum_squares(x, y):
    return mul(x, x)          return square(x)+square(y)
```

What does sum_squares need to know about square?
- Square takes one argument. **Yes**
- Square has the intrinsic name square. **No**
- Square computes the square of a number. **Yes**
- Square computes the square by calling mul. **No**



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

factorial(4)
```

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

Recursive functions have two important components:

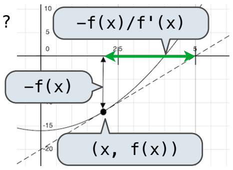
1. *Base case(s)*, where the function directly computes an answer without calling itself
2. *Recursive case(s)*, where the function calls itself as part of the computation

```
square = lambda x: x * x        VS        def square(x):
                                              return x * x
```

- Both create a function with the same arguments & behavior
- Both of those functions are associated with the environment in which they are defined
- Both bind that function to the name "square"
- Only the def statement gives the function an intrinsic name

How to find the square root of 2?

```
>>> f = lambda x: x*x - 2
>>> find_zero(f, 1)
1.4142135623730951
```



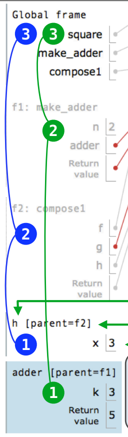
Begin with a function f and an initial guess x

1. Compute the value of f at the guess: f(x)
2. Compute the derivative of f at the guess: f'(x)
3. Update guess to be: $x - \frac{f(x)}{f'(x)}$

```
def iter_improve(update, done, guess=1, max_updates=1000):
    """Iteratively improve guess with update until done returns a true value.

    >>> iter_improve(golden_update, golden_test)
    1.618033988749895
    """
    k = 0
    while not done(guess) and k < max_updates:
        guess = update(guess)
        k = k + 1
    return guess

def newton_update(f):
    """Return an update function for f using Newton's method."""
    def update(x):
        return x - f(x) / approx_derivative(f, x)
    return update

def approx_derivative(f, x, delta=1e-5):
    """Return an approximation to the derivative of f at x."""
    df = f(x + delta) - f(x)
    return df/delta

def find_root(f, guess=1):
    """Return a guess of a zero of the function f, near guess.

    >>> from math import sin
    >>> find_root(lambda y: sin(y), 3)
    3.141592653589793
    """
    return iter_improve(newton_update(f), lambda x: f(x) == 0, guess)
```

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

def make_adder(n):
    def adder(k):
        return k + n
    return adder

def compose1(f, g):
    def h(x):
        return f(g(x))
    return h

compose1(square, make_adder(2))(3)
```

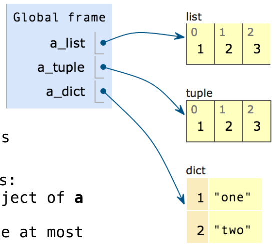
- Every user-defined function has a parent frame
- The parent of a function is the frame in which it was defined
- Every local frame has a parent frame
- The parent of a frame is the parent of the function called



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

CS 61A Midterm 2 Study Guide — Page 1

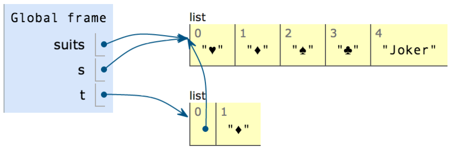
```
a_list = [1, 2, 3]
a_tuple = (1, 2, 3)
a_dict = {1: 'one', 2: 'two'}
```



- Tuples are immutable sequences.
- Lists are mutable sequences.
- Dictionaries are **unordered** collections of key-value pairs.

Dictionary keys do have two restrictions:
- A key of a dictionary **cannot be** an object of **a mutable built-in** type.
- Two **keys cannot be equal.** There can be at most one value for a key.

```
suits = ['♥', '♦']
s = suits
t = list(suits)
suits += ['♠', '♣']
t[0] = suits
suits.append('Joker')
```



```
for <name> in <expression>:
    <suite>
```

1. Evaluate the header <expression>, which must yield an iterable value.
2. For each element in that sequence, in order:
   A. Bind <name> to that element in the local environment.
   B. Execute the <suite>.

A range is a sequence of consecutive integers.*

..., -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, ...

range(-2, 2)

```
>>> city = 'Berkeley'
>>> len(city)
8
>>> city[3]
'k'
```

An element of a string is itself a string!

Length. A sequence has a finite length.

Element selection. A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

Generator expressions

(<map exp> for <name> in <iter exp> if <filter exp>)

- Evaluates to an iterable object.
- <iter exp> is evaluated when the generator expression is evaluated.
- Remaining expressions are evaluated when elements are accessed.

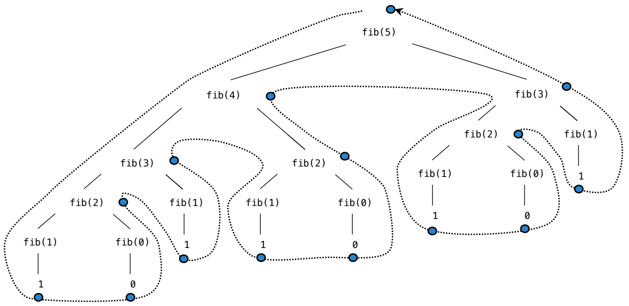
List comprehensions

[<map exp> for <name> in <iter exp> if <filter exp>]

Short version: [<map exp> for <name> in <iter exp>]

Unlike generator expressions, the map expression is evaluated when the list comprehension is evaluated.

```
>>> suits = ['heart', 'diamond', 'spade', 'club']
>>> from unicodedata import lookup
>>> [lookup('WHITE ' + s.upper() + ' SUIT') for s in suits]
['♡', '♢', '♤', '♧']
```



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

Every object that is an instance of a user-defined class has a unique identity:

```
>>> a = Account('Jim')
>>> b = Account('Jack')
```

Identity testing is performed by "is" and "is not" operators. Binding an object to a new name using assignment **does not** create a new object:

```
>>> a is a          >>> c = a
True                >>> c is a
>>> a is not b      True
True
```

nonlocal <name>, <name 2>, ...

Effect: Future assignments to that name change its pre-existing binding in the **first non-local frame** of the current environment in which that name is bound.

Python Docs: an "enclosing scope"

From the Python 3 language reference:

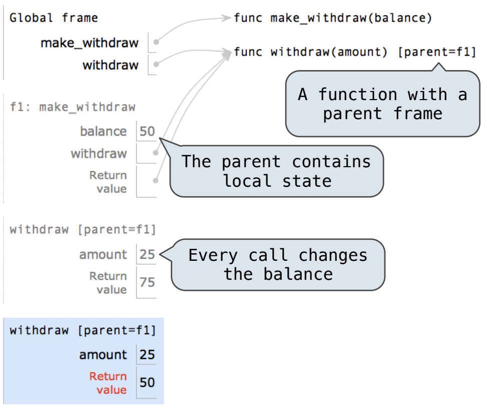
Names listed in a nonlocal statement must refer to pre-existing bindings in an enclosing scope.

Names listed in a nonlocal statement must not collide with pre-existing bindings in the local scope.

x = 2

Status	Effect
•No nonlocal statement •"x" **is not** bound locally	Create a new binding from name "x" to object 2 in the first frame of the current environment.
•No nonlocal statement •"x" **is** bound locally	Re-bind name "x" to object 2 in the first frame of the current env.
•nonlocal x •"x" **is** bound in a non-local frame (but not the global frame)	Re-bind "x" to 2 in the first non-local frame of the current environment in which it is bound.
•nonlocal x •"x" **is not** bound in a non-local frame	SyntaxError: no binding for nonlocal 'x' found
•nonlocal x •"x" **is** bound in a non-local frame •"x" also bound locally	SyntaxError: name 'x' is parameter and nonlocal

```
def make_withdraw(balance):
    def withdraw(amount):
        nonlocal balance
        if amount > balance:
            return 'No funds'
        balance -= amount
    return withdraw

withdraw = make_withdraw(100)
withdraw(25)
withdraw(25)
```



A function with a parent frame

The parent contains local state

Every call changes the balance

Python pre-computes which frame contains each name before executing the body of a function.

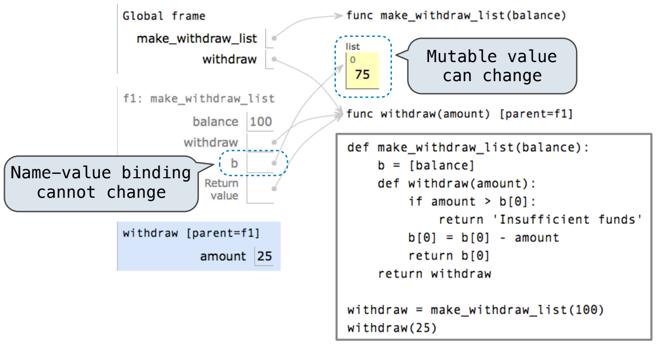
Therefore, within the body of a function, all instances of a name must refer to the same frame.

```
def make_withdraw(balance):
    def withdraw(amount):
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount
        return balance
    return withdraw

wd = make_withdraw(20)
wd(5)
```

Local assignment

UnboundLocalError: local variable 'balance' referenced before assignment

Mutable values can be changed *without* a nonlocal statement.



Mutable value can change

Name-value binding cannot change

```
def make_withdraw_list(balance):
    b = [balance]
    def withdraw(amount):
        if amount > b[0]:
            return 'Insufficient funds'
        b[0] = b[0] - amount
        return b[0]
    return withdraw

withdraw = make_withdraw_list(100)
withdraw(25)
```

```
def pig_latin(w):
    if starts_with_a_vowel(w):
        return w + 'ay'
    return pig_latin(w[1:] + w[0])

def starts_with_a_vowel(w):
    return w[0].lower() in 'aeiou'
```

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

CS 61A Midterm 2 Study Guide — Page 2

```
class <name>(<base class>):
    <suite>
```

- A class statement **creates** a new class and **binds** that class to <name> in the first frame of the current environment.
- Statements in the <suite> create attributes of the class.

To evaluate a dot expression: <expression> . <name>

1. Evaluate the <expression> to the left of the dot, which yields the object of the dot expression.
2. <name> is matched against the instance attributes of that object; **if an attribute with that name exists**, its value is returned.
3. If not, <name> is looked up in the class, which yields a class attribute value.
4. That value is returned **unless it is a function**, in which case a *bound method* is returned instead.

To look up a name in a class.

1. If it names an attribute in the class, return the attribute value.
2. Otherwise, look up the name in the base class, if there is one.

```
class Account(object):
    interest = 0.02
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance

    def withdraw(self, amount):
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance
```

Class attribute

Constructor

Methods

Assignment statements with a dot expression on their left-hand side affect attributes for the object of that dot expression

- If the object is an instance, then assignment sets an instance attribute
- If the object is a class, then assignment sets a class attribute

```
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
>>> tom_account.interest
0.02
>>> jim_account.interest
0.02
>>> tom_account.interest
0.02
>>> Account.interest = 0.04
>>> tom_account.interest
0.04
```

```
>>> jim_account.interest = 0.8
>>> jim_account.interest
0.8
>>> tom_account.interest
0.04
>>> Account.interest = 0.05
>>> tom_account.interest
0.05
>>> jim_account.interest
0.8
```

Instance Attribute Assignment :

```
tom_account.interest = 0.08
```

This expression evaluates to an object

But the name ("interest") is **not** looked up

Attribute assignment statement adds or modifies the "interest" attribute of tom_account

```
class CheckingAccount(Account):
    withdraw_fee = 1
    interest = 0.01
    def withdraw(self, amount):
        return Account.withdraw(self, amount + self.withdraw_fee)
```

Base class

To look up a name in a class:

1. If it names an attribute in the class, return the attribute value.
2. Otherwise, look up the name in the base class, if there is one.

```
>>> ch = CheckingAccount('T')
>>> ch.interest
0.01
>>> ch.deposit(20)
20
>>> ch.withdraw(5)
14
```

```
class SavingsAccount(Account):
    deposit_fee = 2
    def deposit(self, amount):
        return Account.deposit(self, amount - self.deposit_fee)

class AsSeenOnTVAccount(CheckingAccount, SavingsAccount):
    def __init__(self, account_holder):
        self.holder = account_holder
        self.balance = 1            # A free dollar!
```

```
>>> a = Account('Jim')
```

When a class is called:

1. A new instance of that class is created:
2. The constructor __init__ of the class is called with the new object as its first argument (called self), along with additional arguments provided in the call expression.

```
class Account(object):
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
```

A mutable Rlist implementation using message passing

```
def mutable_rlist():
    contents = empty_rlist
    def dispatch(message, value=None):
        nonlocal contents
        if message == 'len':
            return len_rlist(contents)
        elif message == 'getitem':
            return getitem_rlist(contents, value)
        elif message == 'push_first':
            contents = make_rlist(value, contents)
        elif message == 'pop_first':
            f = first(contents)
            contents = rest(contents)
            return f
        elif message == 'str':
            return str(contents)
    return dispatch
```

A bank account implemented using dispatch dictionaries

```
def account(balance):
    def withdraw(amount):
        if amount > dispatch['balance']:
            return 'Insufficient funds'
        dispatch['balance'] -= amount
        return dispatch['balance']
    def deposit(amount):
        dispatch['balance'] += amount
        return dispatch['balance']
    dispatch = {'balance': balance, 'withdraw': withdraw,
                'deposit': deposit}
    return dispatch
```

A simple container implemented using two accessor methods

```
def container(contents):
    def get():
        return contents
    def put(value):
        nonlocal contents
        contents = value
    return put, get
```

```
class ComplexRI(object):
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag
    @property
    def magnitude(self):
        return (self.real ** 2 + self.imag ** 2) ** 0.5
```

Special decorator: "Call this function on attribute look-up"

Type dispatching: Define a different function for each possible combination of types for which an operation is valid

```
def iscomplex(z):
    return type(z) in (ComplexRI, ComplexMA)

def isrational(z):
    return type(z) == Rational

def add_complex_and_rational(z, r):
    return ComplexRI(z.real + r.numer/r.denom, z.imag)

def add_by_type_dispatching(z1, z2):
    """Add z1 and z2, which may be complex or rational."""
    if iscomplex(z1) and iscomplex(z2):
        return add_complex(z1, z2)
    elif iscomplex(z1) and isrational(z2):
        return add_complex_and_rational(z1, z2)
    elif isrational(z1) and iscomplex(z2):
        return add_complex_and_rational(z2, z1)
    else:
        add_rational(z1, z2)
```

Converted to a real number (float)

```
class Rlist(object):

    class EmptyList(object):
        def __len__(self):
            return 0

    empty = EmptyList()

    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest

    def __len__(self):
        return 1 + len(self.rest)

    def __getitem__(self, i):
        if i == 0:
            return self.first
        return self.rest[i-1]
```

The base case

A recursive call

```
class Tree(object):
    def __init__(self, entry,
                 left=None,
                 right=None):
        self.entry = entry
        self.left = left
        self.right = right

def map_rlist(s, fn):
    if s is Rlist.empty:
        return s
    rest = map_rlist(s.rest, fn)
    return Rlist(fn(s.first),rest)

def count_entries(tree):
    if tree is None:
        return 0
    left = count_entries(tree.left)
    right = count_entries(tree.right)
    return 1 + left + right
```

A valid tree has no cycles!

Valid if these are None or a Tree instance

1. Attempt to coerce arguments into values of the same type
2. Apply type-specific (not cross-type) operations

```
def coerce_apply(operator_name, x, y):
    tx, ty = type_tag(x), type_tag(y)
    if tx != ty:
        if (tx, ty) in coercions:
            tx, x = ty, coercions[(tx, ty)](x)
        elif (ty, tx) in coercions:
            ty, y = tx, coercions[(ty, tx)](y)
        else:
            return 'No coercion possible.'
    key = (operator_name, tx)
    return coerce_apply.implementations[key](x, y)
```