CS 61A Structure and Interpretation of Computer Programs
Fall 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 two official 61A midterm study guides 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
/14	/14	/12	/10	/50

1. **(14 points) Classy Costumes**

For each of the following expressions, write the value to which it evaluates. The first two rows have been provided as examples. If evaluation causes an error, write Error. If evaluation never completes, write Forever.

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

```
class Monster:
    vampire = {2: ’scary’}
    def werewolf(self):
        return self.vampire[2]

class Blob(Monster):
    vampire = {2: ’night’}
    def __init__(self, ghoul):
        vampire = {2: ’frankenstein’}
        self.witch = ghoul.vampire
        self.witch[3] = self

spooky = Blob(Monster)
spooky.werewolf = lambda self: Monster.vampire[2]
```

Expression	Evaluates to
`square(5)`	25
`1/0`	Error
`[k+2 for k in range(4)]`	
`Monster.vampire[2][3]`	
`repr(len(spooky.witch))`	
`spooky.witch[3] is not spooky`	
`spooky.witch[2][0:4]`	
`spooky.werewolf()`	
`Monster.werewolf(spooky)`	

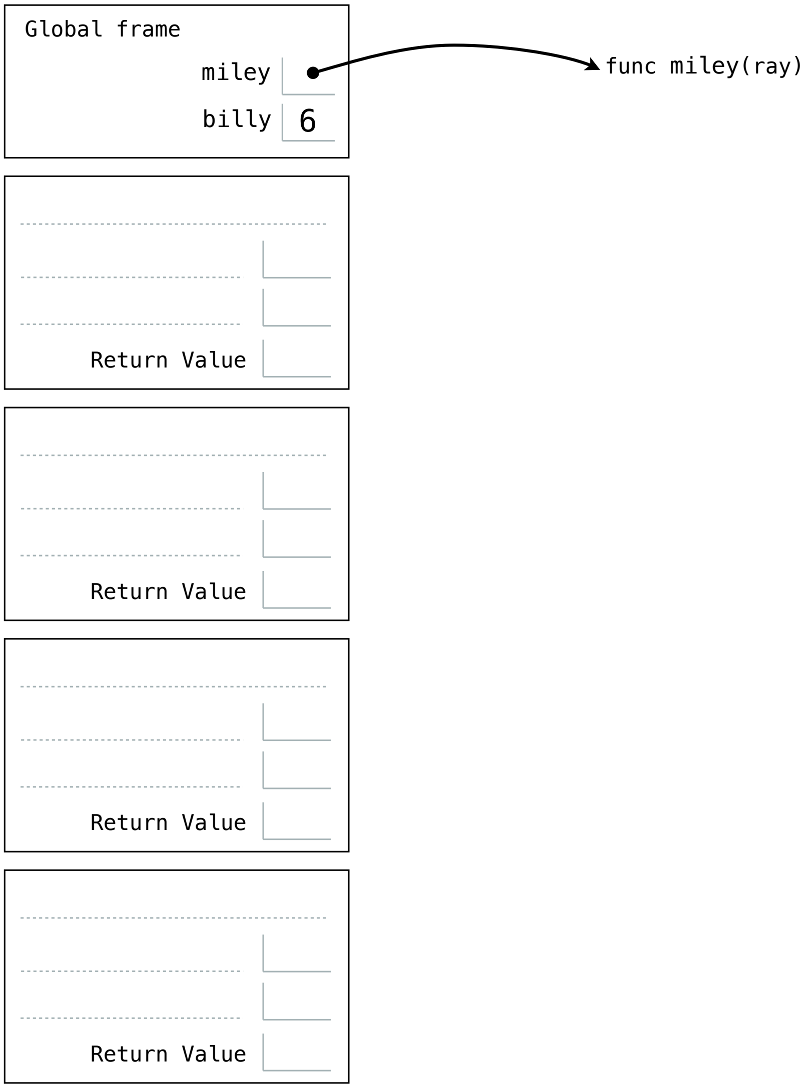
2. (14 points) Wreaking Ball

(a) (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.*

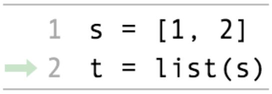
A complete answer will:

- 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 miley(ray):
    def cy():
        def rus(billy):
            nonlocal cy
            cy = lambda: billy + ray
            return (1, billy)
        if len(rus(2)) == 1:
            return (3, 4)
        else:
            return (cy(), 5)
    return cy()[1]
billy = 6
miley(7)
```

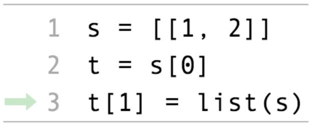


(b) **(6 pt)** Write the letter of the environment diagram that would result from executing each code snippet below, just after starting Python. The first blank is filled for you. Two different snippets may result in the same environment diagram. If none of the environment diagrams are correct, write N.

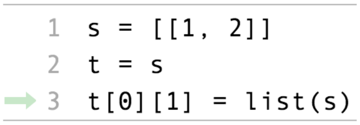


```
s = [1, 2]
t = list(s)
```

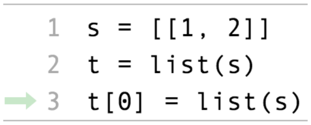
A



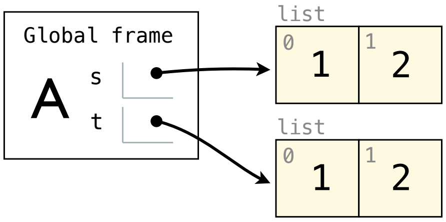
```
s = [[1, 2]]
t = s[0]
t[1] = list(s)
```

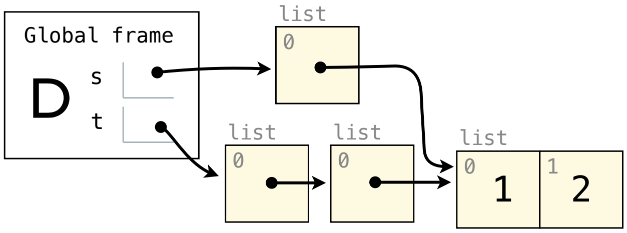


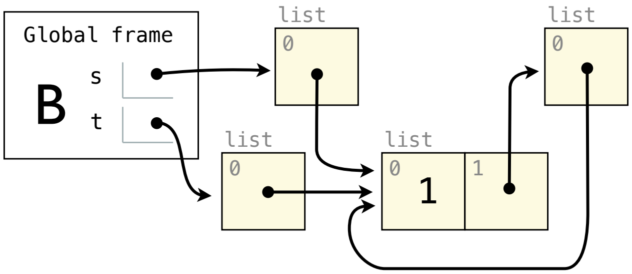
```
s = [[1, 2]]
t = s
t[0][1] = list(s)
```

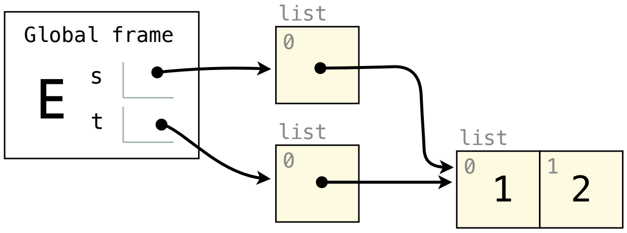


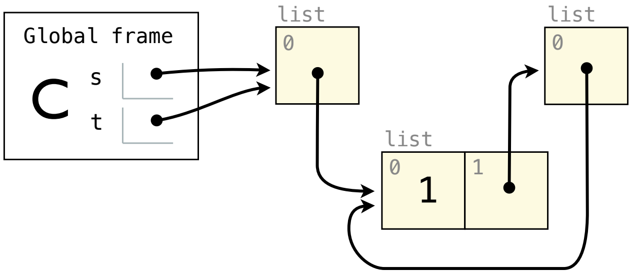
```
s = [[1, 2]]
t = list(s)
t[0] = list(s)
```

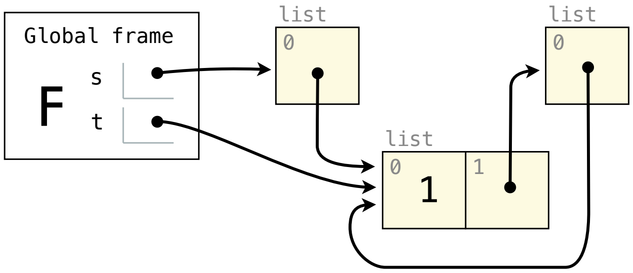












N: None of the above

3. (12 points) Mutants

(a) (4 pt) Given two `Rlist` arguments `a` and `b`, the function `merge(a, b)` changes `a` so that it also includes all elements of `b` at the end, but *does not* change `b`. After merging, changes to `b` *should not* affect `a`. Assume that `a` is not empty, but `b` may be empty. Complete the implementation by filling the blanks with expressions. The `Rlist` class is defined in your study guide.

```
def merge(a, b):
    """Add the elements of b to the end of a, mutating a but not b.

    >>> a = Rlist(1, Rlist(2, Rlist(3)))
    >>> b = Rlist(4, Rlist(5, Rlist(6)))
    >>> merge(a, b)
    >>> a  # a should be modified
    Rlist(1, Rlist(2, Rlist(3, Rlist(4, Rlist(5, Rlist(6))))))
    >>> b  # b should not be modified
    Rlist(4, Rlist(5, Rlist(6)))
    >>> b.first = 7  # modify the elements of b
    >>> b  # b should be modified
    Rlist(7, Rlist(5, Rlist(6)))
    >>> a  # a should not be modified
    Rlist(1, Rlist(2, Rlist(3, Rlist(4, Rlist(5, Rlist(6))))))
    """

    assert a is not Rlist.empty

    if b is Rlist.empty:

        return  # No entries to add to a.


    elif ________________________________________________________:


        __________________________________________________________


        __________________________________________________________


    else:


        __________________________________________________________
```

(b) (2 pt) Define a mathematical function $f(m, n)$ such that evaluating a correct and efficient implementation of `merge(a, b)` on Rlist `a` of length m and Rlist `b` of length n requires $\Theta(f(m, n))$ function calls.

$f(m, n) =$

(c) (6 pt) The function `fold_tree` takes in a three-argument function, a zero value, and a `Tree`. It returns the value of replacing `Tree` with the function and empty branches with the zero value. For each of `size`, `reverse`, and `repeated`, complete the inner function `f`. Each `f` *cannot* be recursive.

```
def fold_tree(fn, zero, tree):
    """Replaces the tree constructor with a 3-argument function.

    >>> t = Tree(3, Tree(5, None, Tree(3)), Tree(2))
    >>> f = lambda a, b, c: a + b + c
    >>> fold_tree(f, 0, t)  # is equivalent to the expression...
    13
    >>> f(3, f(5, 0, f(3, 0, 0)), f(2, 0, 0))
    13
    """
    if tree is None:
        return zero
    return fn(tree.entry, fold_tree(fn, zero, tree.left),
                          fold_tree(fn, zero, tree.right))

def size(tree):
    """Return the number of trees contained in tree.
    >>> size(Tree(3, Tree(5, None, Tree(3)), Tree(2)))
    4
    """
    def f(entry, left, right):




    return fold_tree(f, 0, tree)

def reverse(tree):
    """Return a new tree swapping all left and right branches of tree.
    >>> reverse(Tree(3, Tree(5, None, Tree(3)), Tree(2)))
    Tree(3, Tree(2), Tree(5, Tree(3), None))
    """
    def f(entry, left, right):




    return fold_tree(f, None, tree)

def repeated(tree):
    """Return how many times the root entry of tree appears in tree.
    >>> repeated(Tree(3, Tree(5, None, Tree(3)), Tree(2))) # 3 appears twice
    2
    """
    def f(entry, left, right):







    return fold_tree(f, 0, tree)
```

4. **(10 points) Expansion Mansion**

For a fraction `n/d` with `n < d`, its decimal expansion is written as a series of digits following a decimal point.

For example, 5/8 expands to 0.625. We can compute this result recursively. The numerator 5 times 10 is 50. 50 divided by 8 is **6** with remainder 2. 20 divided by 8 is **2** with remainder 4. 40 divided by 8 is **5** with remainder 0. The quotients in bold are the digits of the expansion. Each subsequent digit is the quotient of dividing 10 times the remainder of the previous digit by the denominator of the fraction.

(a) **(4 pt)** Assume that the decimal expansion of `n/d` is finite, `n` is positive, and `n < d`. The function `expand_finite` returns the digits of the decimal expasion as an `Rlist`. Complete it by filling in each blank with an expression. The `Rlist` class is defined in your study guide.

```
def expand_finite(n, d):
    """Return the finite decimal expansion of n/d as an Rlist. Assume n<d.

    >>> expand_finite(1, 2) # 1/2 = 0.5
    Rlist(5)
    >>> expand_finite(5, 8) # 5/8 = 0.625
    Rlist(6, Rlist(2, Rlist(5)))
    >>> expand_finite(3, 40) # 3/40 = 0.075
    Rlist(0, Rlist(7, Rlist(5)))
    """

    dividend = n * 10

    quotient, remainder = dividend // d, dividend % d


    if ________________________________________________________________:

        return ________________________________________________________

    else:

        return ________________________________________________________
```

(b) **(2 pt)** The function `coerce_to_float` returns the float equal to an input `Rlist` representing the series of digits following the decimal point in a finite decimal expansion. Complete its implementation below.

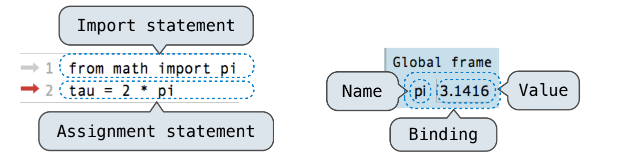
```
def coerce_to_float(s):
    """Return a float equal to an Rlist encoding a series of digits.

    >>> coerce_to_float(expand_finite(1, 2))
    0.5
    >>> coerce_to_float(expand_finite(3, 40))
    0.075
    """

    if s is Rlist.empty:

        return 0

    else:

        return _______________________________________________________
```

Repeating Decimal Expansions. The decimal expansion of a rational number may be infinite, but can always be described by a finite (and possibly repeating) series of digits. We can represent a series of digits as a recursive list, which may contain itself.

(c) **(4 pt)** Assume that `n` is positive and `n < d`. The `expand` function returns representations of both finite and infinite decimal expansions. Complete it by filling in each blank with an expression or assignment statement.

```
def expand(n, d):
    """Return the decimal expansion of n/d as an Rlist. Assume n < d.

    >>> expand(1, 2) # 1/2 = 0.5
    Rlist(5)
    >>> expand(5, 8) # 5/8 = 0.625
    Rlist(6, Rlist(2, Rlist(5)))
    >>> third = expand(1, 3)  # 1/3 = 0.333333...
    >>> [third[i] for i in range(10)]
    [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
    >>> third.rest is third  # There is only one unique digit in 1/3
    True
    >>> fourteenth = expand(1, 14)  # 1/14 = 0.0714285714285...
    >>> [fourteenth[i] for i in range(10)]
    [0, 7, 1, 4, 2, 8, 5, 7, 1, 4]
    """

    return expand_using(n, d, {})

def expand_using(n, d, known):
    """Return the decimal expansion of n/d as an Rlist.
    known -- a dictionary from integer k to the decimal expansion of k/d.
    """

    if n in known:

        return known[n]

    else:

        dividend = n * 10

        quotient, remainder = dividend // d, dividend % d


        digits = ____________________________________________


        ________________________________________________________


        if _____________________________________________________:


            ____________________________________________________

        return digits
```

CS 61A Midterm 1 Study Guide – Page 1



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

Code (left):

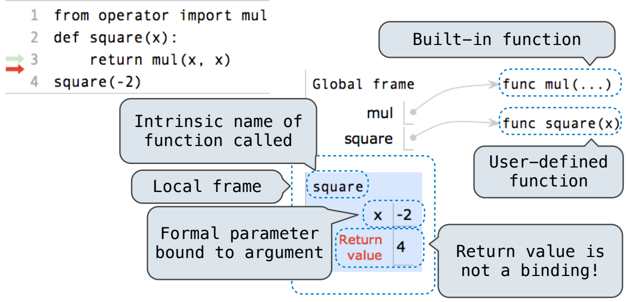
Statements and expressions

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

Frames (right):

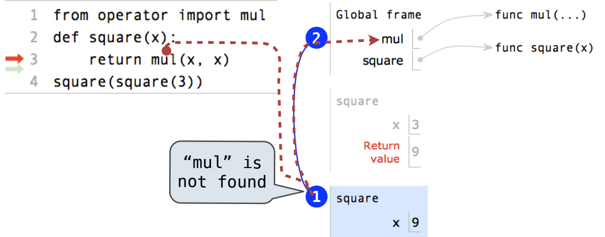
A name is bound to a value

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



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

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:

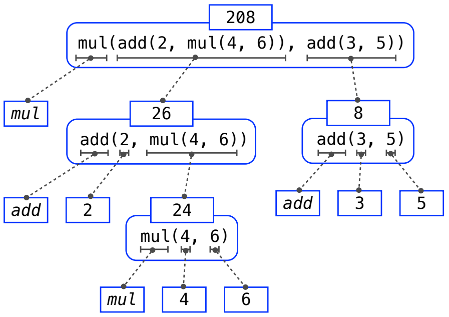
1. Evaluate the subexpression <left>.
2. If the result is a false value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

Evaluation rule for not expressions:

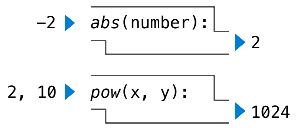
1. Evaluate <exp>; The value is True if the result is a false value, and False otherwise.

Execution rule for while statements:

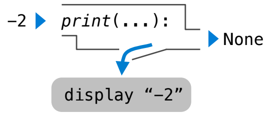
1. Evaluate the header’s expression.
2. If it is a true value, execute the (*whole*) suite, then return to step 1.

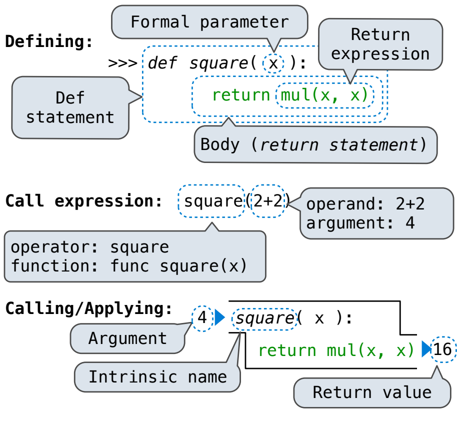


Pure Functions



Non-Pure Functions





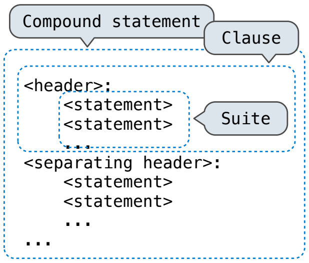
Defining:

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

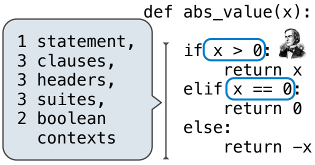
Call expression: square(2+2)

Calling/Applying:

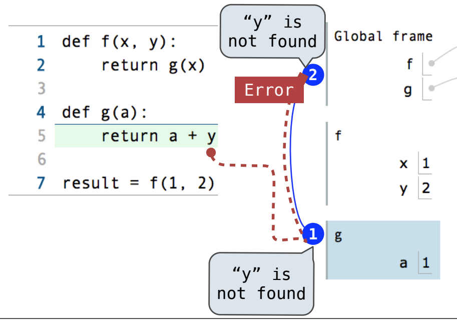
```
square( x ):
    return mul(x, x)
```



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

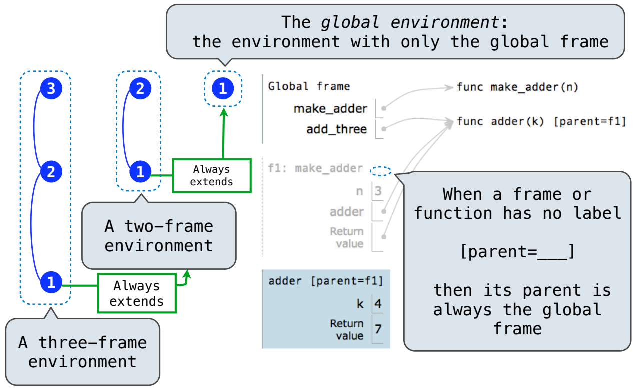


```
def abs_value(x):
    if x > 0:
        return x
    elif x == 0:
        return 0
    else:
        return -x
```

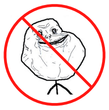


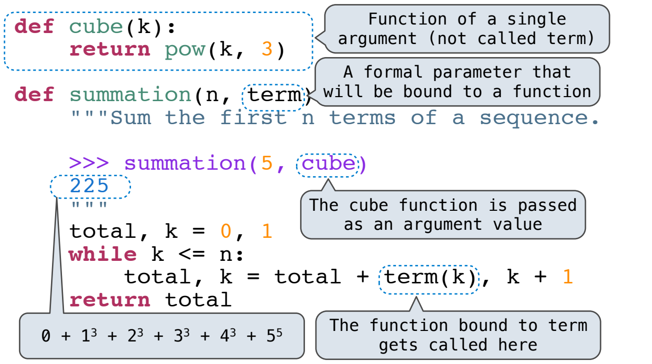
```
def f(x, y):
    return g(x)

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



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




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

```
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
```

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

CS 61A Midterm 1 Study Guide – Page 2

```
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 * x"

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
```

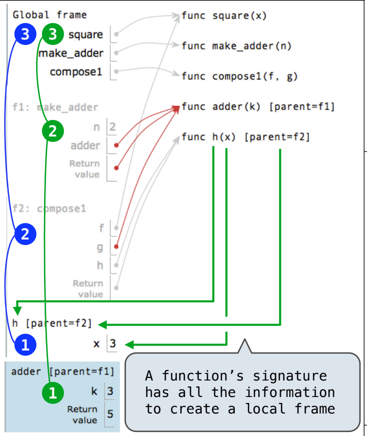
A function that returns a function

The name add_three is bound to a function

A local def statement

Can refer to names in the enclosing function

```
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)
```



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

- 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*

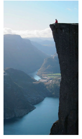
```
def curry2(f):
    """Returns a function g such that g(x)(y) returns f(x, y)."""
    def g(x):
        def h(y):
            return f(x, y)
        return h
    return g
```

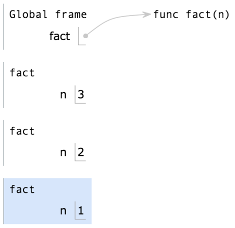
Currying: Transforming a multi-argument function into a single-argument, higher-order function.

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

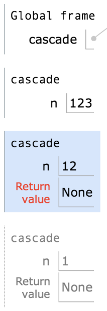




Is fact implemented correctly?

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 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.

```
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. If the **parent frame** of that function is not the global frame, add matching labels to the **parent frame** and the **function value** (such as *f1*, *f2*, or *f3*).

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

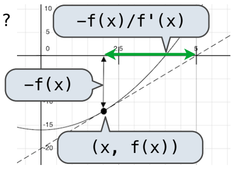
3. Bind *<name>* to the **function value** in the first frame of the current environment.

When a function is called:

1. Add a **local frame**, titled with the *<name>* of the function being called.
2. If the function has a parent label, copy it 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**.

How to find the square root of 2?

```
>>> f = lambda x: x*x - 2
>>> df = lambda x: 2*x
>>> find_zero(f, df)
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 improve(update, close, guess=1):
    """Iteratively improve guess with update until close(guess) is true."""
    while not close(guess):
        guess = update(guess)
    return guess

def approx_eq(x, y, tolerance=1e-15):
    return abs(x - y) < tolerance

def find_zero(f, df):
    """Return a zero of the function f with derivative df."""
    def near_zero(x):
        return approx_eq(f(x), 0)
    return improve(newton_update(f, df), near_zero)

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

def power(x, n):
    """Return x * x * x * ... * x for x repeated n times."""
    product, k = 1, 0
    while k < n:
        product, k = product * x, k + 1
    return product

def nth_root_of_a(n, a):
    """Return the nth root of a."""
    def f(x):
        return power(x, n) - a
    def df(x):
        return n * power(x, n-1)
    return find_zero(f, df)
```

- Recursive decomposition: finding simpler instances of the problem: **partition(6, 4)**
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
- Solve two simpler problems:
  - **partition(2, 4)**
  - **partition(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
```

```
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
    """
    return floordiv(n, d), mod(n, d)
```

Multiple assignment to two names

Multiple return values, separated by commas

CS 61A Midterm 2 Study Guide — Page 1

Numeric types in Python:

```
>>> type(2)
<class 'int'>

>>> type(1.5)
<class 'float'>

>>> type(1+1j)
<class 'complex'>
```

Represents integers exactly

Represents real numbers approximately

User-defined complex type:

```
>>> z = ComplexRI(-1, 0)
>>> (z.real, z.imag)
(-1, 0)
>>> z.magnitude
1
>>> z.angle
3.141592653589793
```

```
class ComplexRI:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag
    @property
    def magnitude(self):
        return (self.real
    @property
    def angle(self):
        return atan2(self.imag, self.real)
    def __repr__(self):
        return 'ComplexRI({0}, {1})'.format(self.real,
                                            self.imag)
```

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

math.atan2(y,x): Angle between x-axis and the point (x,y)

Type dispatching: Look up a cross-type implementation of an operation based on the types of its arguments
Data-directed programming: Look up a cross-type implementation based on both the operator and types of its arguments
Type coercion: Look up a function for converting one type to another, then apply a type-specific implementation.

Rational number: $\frac{\text{numerator}}{\text{denominator}}$

- Exact representation of fractions
- A pair of integers
- As soon as division occurs, the exact representation may be lost!
- Assume we can compose and decompose rational numbers:

Constructor
- rational(n, d) *returns a rational number x*

Selectors
- numer(x) *returns the numerator of x*
- denom(x) *returns the denominator of x*

These functions implement an *abstract data type* for rational numbers

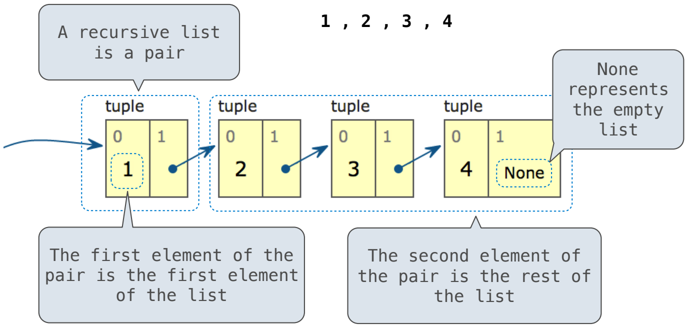
There isn't just one sequence class or abstract data type (in Python or in general).

The sequence abstraction is a collection of behaviors:

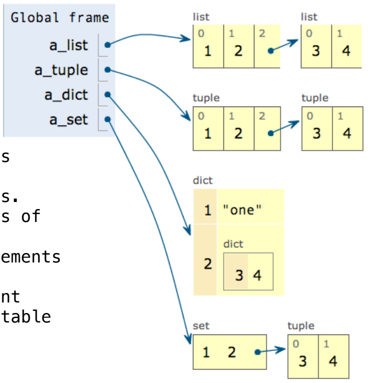
Length. A sequence has a finite length.

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.

We can implement recursive lists as pairs. We'll use two-element tuples to encode pairs.


1 , 2 , 3 , 4
A recursive list is a pair
None represents the empty list
tuple
tuple
tuple
tuple
0
1
1
2
3
4
None
The first element of the pair is the first element of the list
The second element of the pair is the rest of the list

```
a_list = [1, 2, [3, 4]]
a_tuple = (1, 2, (3, 4))
a_dict = {1: 'one', 2: {3: 4}}
a_set = {1, 2, (3, 4)}
```



- Lists are mutable sequences
- Tuples are immutable sequences
- Dictionaries are unordered collections of key-value pairs.
- Sets are unordered collections of values.
- Two dictionary keys or set elements cannot be equal.
- A dictionary key or set element cannot be an instance of a mutable built-in type.

Executing a for statement:

```
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 first frame of the current 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)

Length: ending value - starting value

Element selection: starting value + index

```
>>> tuple(range(-2, 2))
(-2, -1, 0, 1)

>>> tuple(range(4))
(0, 1, 2, 3)
```

Tuple constructor

With a 0 starting value

List comprehensions: `[<map exp> for <name> in <iter exp> if <filter exp>]`
Short version: `[<map exp> for <name> in <iter exp>]`

A combined expression that evaluates to a list by this procedure:
1. Add a new frame extending the current frame.
2. Create an empty *result list* that is the value of the expression.
3. For each element in the iterable value of `<iter exp>`:
   A. Bind `<name>` to that element in the new frame from step 1.
   B. If `<filter exp>` evaluates to a true value, then add the value of `<map exp>` to the result list.

Strings are sequences too:

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

An element of a string is itself a string, but with only one character!

The "in" and "not in" operators match substrings

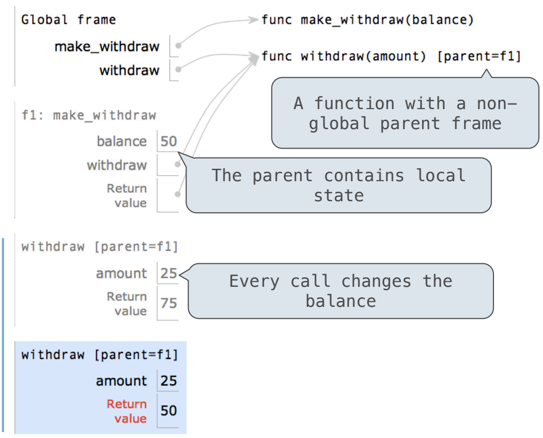
```
>>> 'here' in "Where's Waldo?"
True
>>> 234 in (1, 2, 3, 4, 5)
False
```

```
def make_withdraw(balance):
    """Return a withdraw function with a starting balance."""
    def withdraw(amount):
        nonlocal balance
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount
        return balance
    return withdraw
```

Declare the name "balance" nonlocal at the top of the body of the function in which it is re-assigned

Re-bind balance in the first non-local frame in which it was bound previously

```
>>> withdraw(25)
75

>>> withdraw(25)
50
```



All calls to the same function have the same parent

Status	x = 2	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		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

CS 61A Midterm 2 Study Guide – Page 2

```
class <name>:
    <suite>
```

The suite is executed when a class statement is evaluated.

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.

As soon as an instance is created, it is passed to `__init__`, which is a class attribute called the *constructor method*.

Objects receive messages via dot notation.

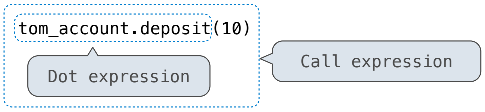
Dot notation accesses attributes of the instance **or** its class.

`<expression> . <name>`

The `<expression>` can be any valid Python expression.

The `<name>` must be a simple name.

Evaluates to the value of the attribute **looked up** by `<name>` in the object that is the value of the `<expression>`.



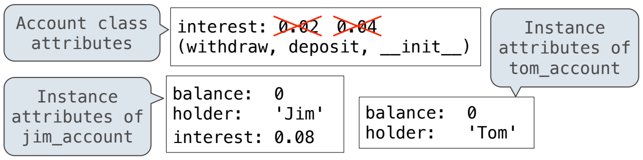
`tom_account.deposit(10)`

To evaluate a dot expression:

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 (see inheritance below).
4. That value is returned **unless it is a function**, in which case a *bound method* is returned instead.

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

- For an instance, then assignment sets an instance attribute
- For a class, then assignment sets a class attribute



Account class attributes
interest: ~~0.02~~ ~~0.04~~
(withdraw, deposit, __init__)
Instance attributes of tom_account
Instance attributes of jim_account
balance: 0
holder: 'Jim'
interest: 0.08
balance: 0
holder: 'Tom'

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

```
>>> jim_account.interest = 0.08
>>> jim_account.interest
0.08
>>> tom_account.interest
0.04
>>> Account.interest = 0.05
>>> tom_account.interest
0.05
>>> jim_account.interest
0.08
```

Inheritance: To look up a name in a class.
1. If it names an attribute in the class, return its value.
2. Otherwise, look up the name in the base class, if it exists.

```
class Account:
    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 CheckingAccount(Account):
    """A bank account that charges for withdrawals."""
    withdraw_fee = 1
    interest = 0.01
    def withdraw(self, amount):
        return Account.withdraw(self, amount + self.withdraw_fee)

 >>> ch = CheckingAccount('Tom')  # Calls Account.__init__
 >>> ch.interest     # Found in CheckingAccount
 0.01
 >>> ch.deposit(20)  # Found in Account
 20
 >>> ch.withdraw(5)  # Found in CheckingAccount
 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!
```

When a class is called: `>>> a = Account('Jim')`

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 (named `self`), along with any additional arguments provided in the call expression.

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

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

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

Every call to Account creates a new Account instance. There is only one Account class.

Identity testing is performed by "is" and "is not" operators:

```
>>> a is a
True
>>> a is not b
True
```

Binding an object to a new name using assignment **does not** create a new object:

```
>>> c = a
>>> c is a
True
```

All invoked methods have access to the object via the `self` parameter, and so they can all access and manipulate the object's state.

```
class Account:
    ...
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
```

Defined with two arguments

Dot notation automatically supplies the first argument to a method.

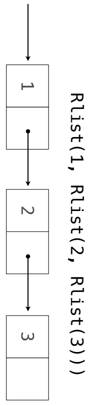
```
>>> tom_account = Account('Tom')
>>> tom_account.deposit(100)
100
```

Invoked with one argument

```
class Rlist:
    class EmptyList:
        def __len__(self):
            return 0
    empty = EmptyList()
    def __init__(self, first, rest=empty):
        assert type(rest) is Rlist or rest is Rlist.empty
        self.first = first
        self.rest = rest
    def __getitem__(self, index):
        if index == 0:
            return self.first
        else:
            return self.rest[index-1]
    def __len__(self):
        return 1 + len(self.rest)
    def __repr__(self):
        rest = ''
        if self.rest is not Rlist.empty:
            rest = ', ' + repr(self.rest)
        return 'Rlist({0}{1})'.format(self.first, rest)
def extend_rlist(s1, s2):
    if s1 is Rlist.empty:
        return s2
    return Rlist(s1.first, extend_rlist(s1.rest, s2))
```

There's the base case!

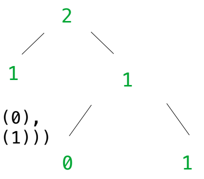
Yes, this call is recursive



```
class Tree:
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right
def sum_entries(t):
    if t is None:
        return 0
    else:
        return t.entry + sum_entries(t.left) + sum_entries(t.right)
```

left and right are Trees or None

```
Tree(2, Tree(1),
        Tree(1, Tree(0),
                Tree(1)))
```



```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

n: size of the problem

R(n): Measurement of some resource

$$R(n) = \Theta(f(n))$$

means that there are positive constants k_1 and k_2 such that

$$k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n)$$

for sufficiently large values of ***n***.

$\Theta(b^n)$ Exponential growth! Incrementing the problem scales R(n) by a factor.

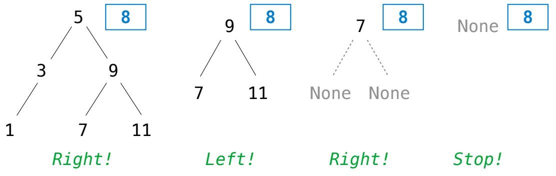
$\Theta(n^2)$ Quadratic growth. Incrementing n increases R(n) by the problem size n.

$\Theta(n)$ Linear growth. Resources scale with the problem size.

$\Theta(\log n)$ Logarithmic growth. Doubling the problem only increments R(n).

$\Theta(1)$ Constant. Independent of problem size.

Tree set: A set is a Tree. Each entry is:
- Larger than all entries in its left branch, and
- Smaller than all entries in its right branch



5
8
3
9
1
7
11
Right!
9
8
7
11
Left!
7
8
None
None
Right!
None
8
Stop!