CS 61A Structure and Interpretation of Computer Programs

Fall 2012 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	/12	/14	/12	/50

1. (12 points) The Call Express is Delayed

For each of the following call expressions, write the value to which it evaluates *and* what would be output by the interactive Python interpreter. The first two rows have been provided as examples.

- In the **Evaluates to** column, write the value to which the expression evaluates. If evaluation causes an error, write Error.
- In the column labeled **Interactive Output**, write all output that would be displayed during an interactive session, after entering each call expression. This output may have multiple lines. Whenever the interpreter would report an error, write Error. You *should* include any lines displayed before an error.

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

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

def delay(arg):
    print('delayed')
    def g():
        return arg
    return g
```

Expression	Evaluates to	Interactive Output
`square(5)`	25	25
`1/0`	Error	Error
`print(square(4))`		
`square(square(print(2)))`		
`print(add(3, 4), print(5))`		
`delay(square)(3)`		
`add(delay(square)()(2), 1)`		
`delay(delay)()(6)()`		

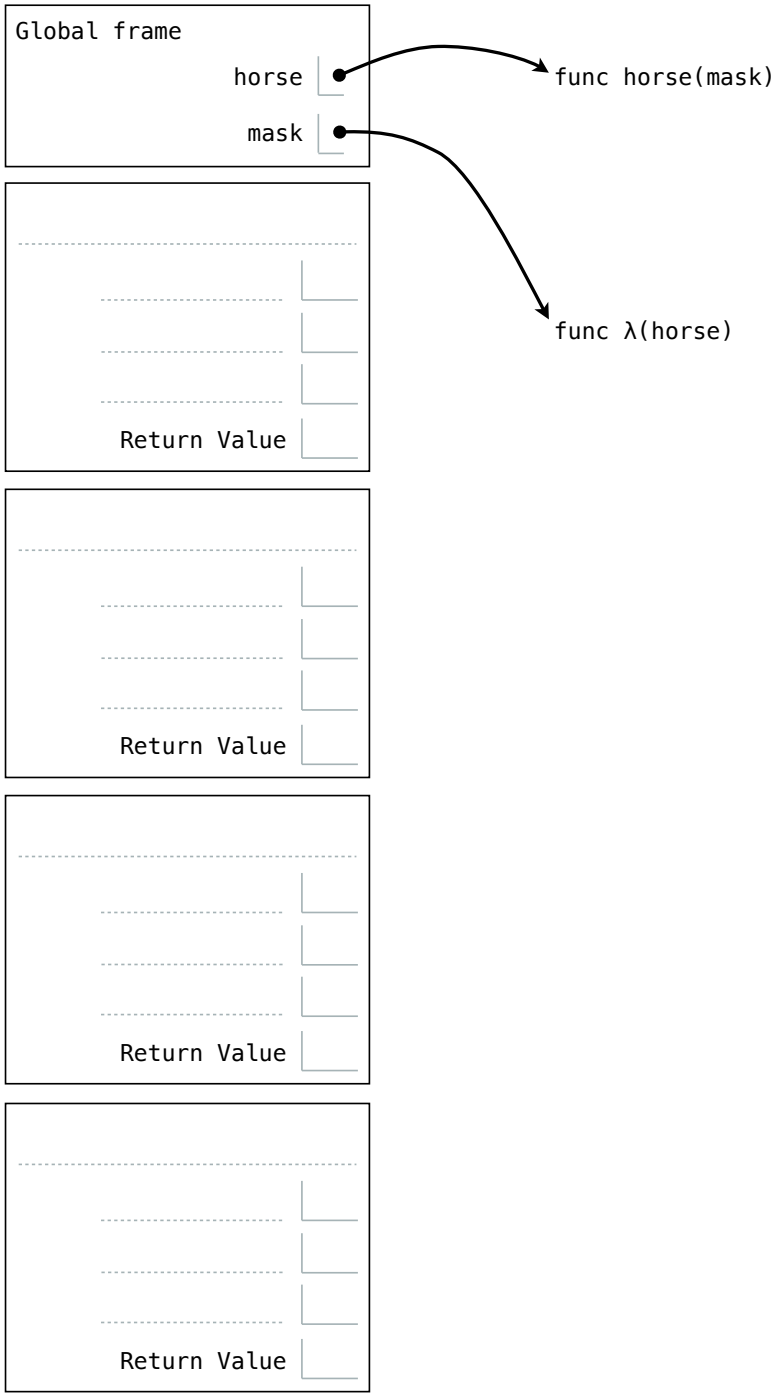
2. (12 points) Protect the Environment

(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.

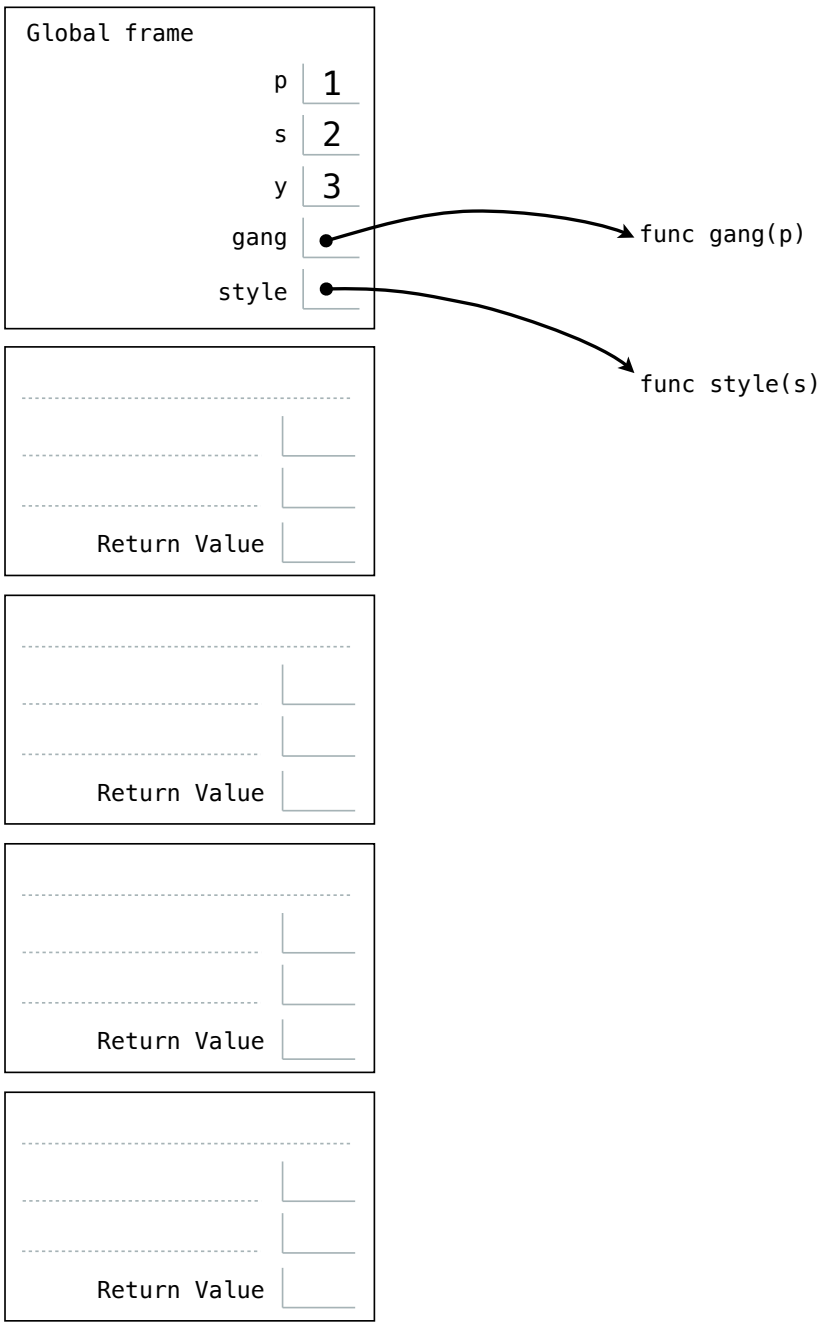
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```



(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 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.

```
p, s, y = 1, 2, 3

def gang(p):
    nam = style(p)
    return (nam(4), 5)

def style(s):
    return lambda y: (p, s, y)

gang(3)
```



3. (14 points) Sequences

(a) (2 pt) Fill in the blanks so that the final call expression below evaluates to a **tuple** value.

```
def pair(x):
    if x == 30:
        return lambda: (1, 2, 3)
    else:
        return lambda: 4


(lambda_____________, soda: hall_____________)(pair, "sequence")
```

(b) (2 pt) Draw a box and pointer diagram for the following `rlist`:

```
a = rlist(1, rlist(rlist(2, (3, empty_rlist)), rlist((4, 5, 6), empty_rlist)))
```

(c) (2 pt) What is the element at index 1 of this rlist, returned by `getitem_rlist(a, 1)`?

```
def getitem_rlist(s, i):
    """Return the element at index i of recursive list s."""
    while i > 0:
        s, i = rest(s), i - 1
    return first(s)
```

(d) (2 pt) What is the length of this rlist, returned by `len_rlist(a)`?

```
def len_rlist(s):
    """Return the length of recursive list s."""
    length = 0
    while s != empty_rlist:
        s, length = rest(s), length + 1
    return length
```

(e) (6 pt) When the `int` constructor is called on a `float` value, it "truncates toward zero," meaning that it returns the largest integer less than any positive argument, or the least integer greater than any negative argument. For example:

```
>>> int(2)
2
>>> int(2.7)
2
>>> int(-1.5)
-1
```

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

```
def alt(f, g, z):
    while g(z) > 0 and z != 5:
        f, g = g, f
        z = g(z)
    return z

def grow(x):
    return int((x * 3) / 2)

def shrink(x):
    return x - 2

def flip(x):
    return int(10 / (x-2))
```

For each of the following call expressions, write the value to which it evaluates. If evaluation causes an error, write ERROR. If evaluation would run forever, write FOREVER.

- `alt(grow, shrink, 3)`

- `alt(grow, shrink, 4)`

- `alt(flip, shrink, 3)`

4. (12 points) In Verse

The inverse of some function F	*There once was a rhyming device*
is a function of argument X	*That was built to make any sound, twice,*
that returns you the Y,	*But used orthography*
such that when you apply	*And not phonology*
F to Y you recover the X.	*To decide if a rhyme would suffice.*

An invertible function is a function that takes and returns a single numeric value, is differentiable, and never returns the same value for two different arguments. Some examples:

```
def double(y):
    """Return twice the value of y."""
    return 2 * y

def cube(y):
    """Return y raised to the third power."""
    return pow(y, 3)

def pow2(y):
    """Return 2 raised to the power of y."""
    return pow(2, y)
```

(a) (4 pt) Implement a function `invert` that takes an invertible function argument and returns its inverse. You may call `find_root`, `newton_update`, `approx_deriv`, and/or `iter_improve`. You **cannot** use any assignment, conditional, `while`, or `for` statements.

```
def invert(f):
    """Return the inverse of invertible function f.

    >>> halve = invert(double)
    >>> halve(12)
    6.0
    >>> cube_root = invert(cube)
    >>> cube_root(27)
    3.0
    >>> log2 = invert(pow2)
    >>> log2(32)
    5.0
    """
```

A sight rhyme is a pair of words that do not rhyme, but have the same endings, such as *device* and *office*. Two numbers that end in the same digit can be sight rhymes. For example:

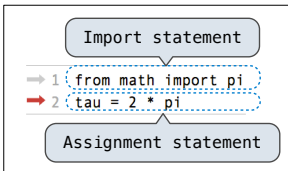
- `(13, 53)` are pronounced *thirteen* and *fifty-three*, despite both ending with the same one's digit `3`.
- `(0, 30)` are pronounced *zero* and *thirty*, despite both ending with the same one's digit `0`.

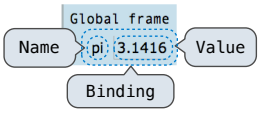
(b) (4 pt) A `numpair` is a pair of integers that have the same one's digit. Fill in the **two** missing expressions in the constructor below, which takes two non-negative integers less than 100, asserts that they have the same one's digit, and returns a numpair represented as a pair of tens digits and the shared one's digit.

```
from operator import floordiv, mod  # Use these functions or // and %

def numpair(first, second):
    """Return a numpair as a pair of ten's digits and a shared one's digit.

    >>> numpair(23, 53)
    ((2, 5), 3)
    >>> numpair(67, 7)
    ((6, 0), 7)
    """
    assert _____________________________________________, "different one's"

    return ____________________________________________________________
```

(c) (4 pt) Fill in **four** missing expressions below so that `sight_rhyme` returns whether the numbers in a `numpair p` **do not** end with the same sound when *pronounced*. Your implementation **cannot** depend on the *representation* of a `numpair`; use selector functions. You **cannot** use the boolean operators `and` or `or`.

```
def ones(p):
    return p[1]
def first_tens(p):
    return p[0][0]
def second_tens(p):
    return p[0][1]
def sight_rhyme(p):
    """Return whether the two numbers in a numpair do not rhyme.

    >>> sight_rhyme(numpair(13, 53))
    True
    >>> sight_rhyme(numpair(0, 30))
    True
    >>> sight_rhyme(numpair(53, 23))
    False
    """
    if ____________________________________________:

        return ____________________________________________
    elif ones(p) == 0:
        if first_tens(p) == 0:

            return ____________________________________________
        else:

            return ____________________________________________
    else:
        return False
```





Code (left):

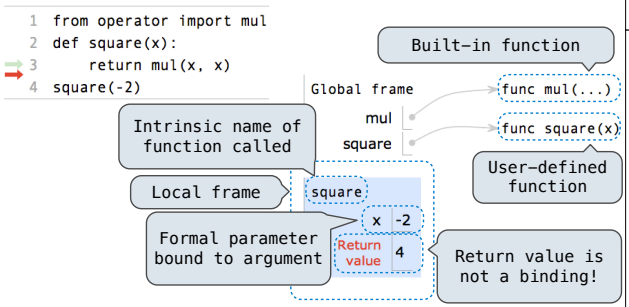
Statements and expressions

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

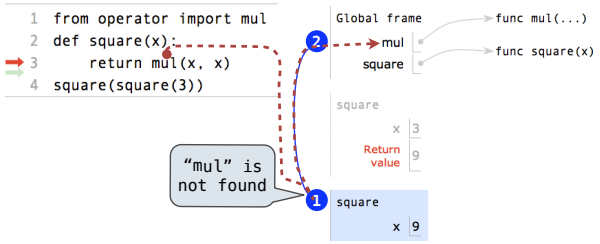
Frames (right):

A name is bound to a value

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



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



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:

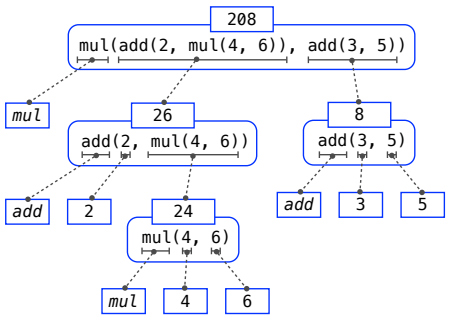
1. Evaluate the subexpression <left>.
2. If the result is a false value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

Evaluation rule for not expressions:

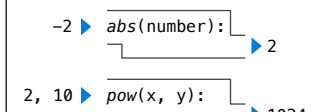
1. Evaluate <exp>; The value is True if the result is a false value, and False otherwise.

Execution rule for while statements:

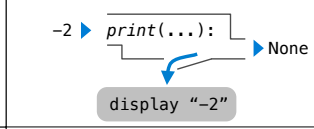
1. Evaluate the header's expression.
2. If it is a true value, execute the (*whole*) suite, then return to step 1.

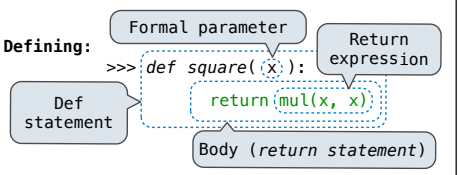


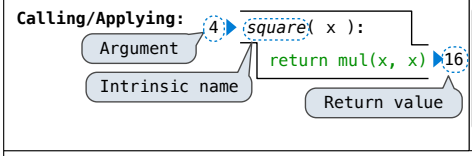
Pure Functions

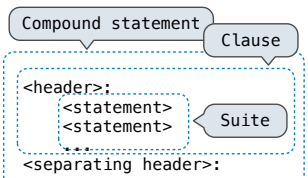


Non-Pure Functions



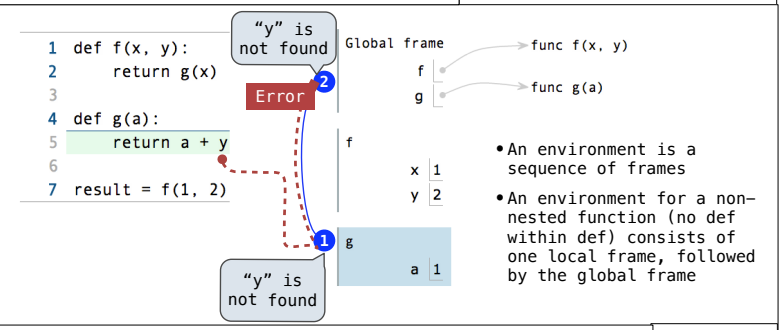




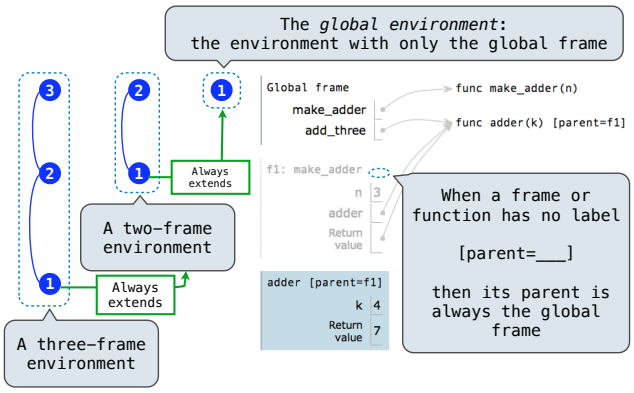


1 statement,
3 clauses,
3 headers,
3 suites,
2 boolean contexts

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



- 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

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)
    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

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

The function bound to term gets called here

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

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

Must be a single expression

```
@trace1
def triple(x):
    return 3 * x
```

is identical to

```
def triple(x):
    return 3 * x
triple = trace1(triple)
```

```
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

```
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

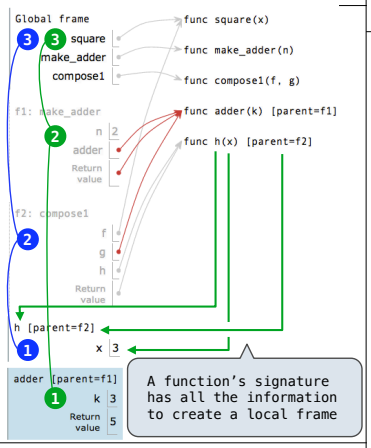
The name add_three is bound to a function

A local def statement

Can refer to names in the enclosing function

```
1  def square(x):
2      return x * x
3
4  def make_adder(n):
5      def adder(k):
6          return k + n
7      return adder
8
9  def compose1(f, g):
10     def h(x):
11         return f(g(x))
12     return h
13
14 compose1(square, make_adder(2))(3)
```

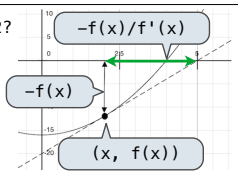
- Every user-defined function has a parent frame
- The parent of a function is the frame in which it was defined
- Every local frame has a parent frame
- The parent of a frame is the parent of the function called



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

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)
```

- Compound objects combine objects together
- An *abstract data type* lets us manipulate compound objects as units
- Programs that use data isolate two aspects of programming:
  - How data are represented (as parts)
  - How data are manipulated (as units)
- Data abstraction: A methodology by which functions enforce an abstraction barrier between ***representation*** and ***use***

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

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

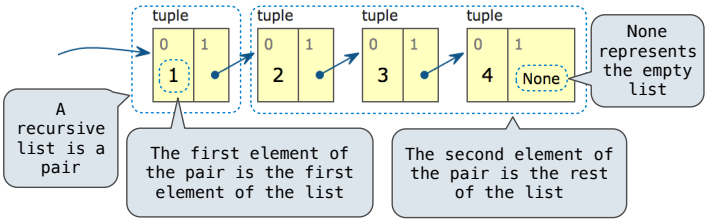
What does sum_squares need to know about square?

- 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 mul_rational(x, y):
    return rational(numer(x) * numer(y), denom(x) * denom(y))

def add_rational(x, y):
    nx, dx = numer(x), denom(x)
    ny, dy = numer(y), denom(y)
    return rational(nx * dy + ny * dx, dx * dy)
def eq_rational(x, y):
    return numer(x) * denom(y) == numer(y) * denom(x)
```

Constructor

Selectors




```
def rational(n, d):
    """Construct a rational number x that represents n/d."""
    return (n, d)

from operator import getitem
def numer(x):
    """Return the numerator of rational number x."""
    return getitem(x, 0)

def denom(x):
    """Return the denominator of rational number x."""
    return getitem(x, 1)
```

```
empty_rlist = None
def rlist(first, rest):
    """Make a recursive list from its first element and the rest."""
    return (first, rest)
def first(s):
    """Return the first element of a recursive list s."""
    return s[0]
def rest(s):
    """Return the rest of the elements of a recursive list s."""
    return s[1]
```

If a recursive list s is constructed from a first element f and a recursive list r, then

- first(s) returns f, and
- rest(s) returns r, which is a recursive list.

```
def pair(x, y):
    """Return a functional pair."""
    def dispatch(m):
        if m == 0:
            return x
        elif m == 1:
            return y
    return dispatch
def getitem_pair(p, i):
    """Return the element at index i of pair p."""
    return p(i)
```

This function represents a pair

```
def len_rlist(s):
    """Return the length of recursive list s."""
    length = 0
    while s != empty_rlist:
        s, length = rest(s), length + 1
    return length

def getitem_rlist(s, i):
    """Return the element at index i of rlist s."""
    while i > 0:
        s, i = rest(s), i - 1
    return first(s)
```

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.

```
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