CS 61A Structure and Interpretation of Computer Programs

Fall 2012

ALTERNATE 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
/16	/12	/14	/8	/50

THIS PAGE INTENTIONALLY LEFT BLANK

1. (16 points) Expressionism

(a) (8 pt) For each of the following expressions, write the `repr` string of the value to which the expression evaluates. Special cases: If an expression evaluates to a function, write FUNCTION. If evaluation would never complete, write FOREVER. None of these expressions cause an error.

Assume that the expressions are evaluated in order. Evaluating the first may affect the value of the second, etc.

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

```
def fruit(y):
    def ninja(angry):
        nonlocal y
        if y < len(angry):
            return angry
        y = y - 2
        return pig(angry)

    def pig(bird):
        bird.append(y)
        return ninja(bird)

    return pig

slingshot = fruit(5)
green = lambda x: fruit(x)([x])
```

Expression	Evaluates to
5*5	25
slingshot([1, 2, 3, 4])	
green(5)	
slingshot([6, 7])	
green(8)	

(b) **(8 pt)** For each of the following expressions, write the `repr` string of the value to which the expression evaluates. Special cases: If an expression evaluates to a function, write FUNCTION. If evaluation would never complete, write FOREVER. None of these expressions cause an error.

Assume that the expressions are evaluated in order. Evaluating the first may affect the value of the second, etc.

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

```
class Student(object):
    def __init__(self, s):
        if len(s) < 2:
            self.s = s
        else:
            self.s = Student(s[1:])

    def __repr__(self):
        return 'Student(' + repr(self.s) + ')'

    def learn(self):
        if hasattr(self, 'teach'):
            return self.teach()
        while type(self.s) == Student:
            self.s = self.s.s
        return self.s

class Teacher(Student):
    def teach(self):
        return 'Good Job'

luke = Student([1, 3])
yoda = Teacher([5, luke])
```

Expression	Evaluates to
`5*5`	25
`luke.learn()`	
`luke`	
`yoda`	
`Student.learn(yoda)`	

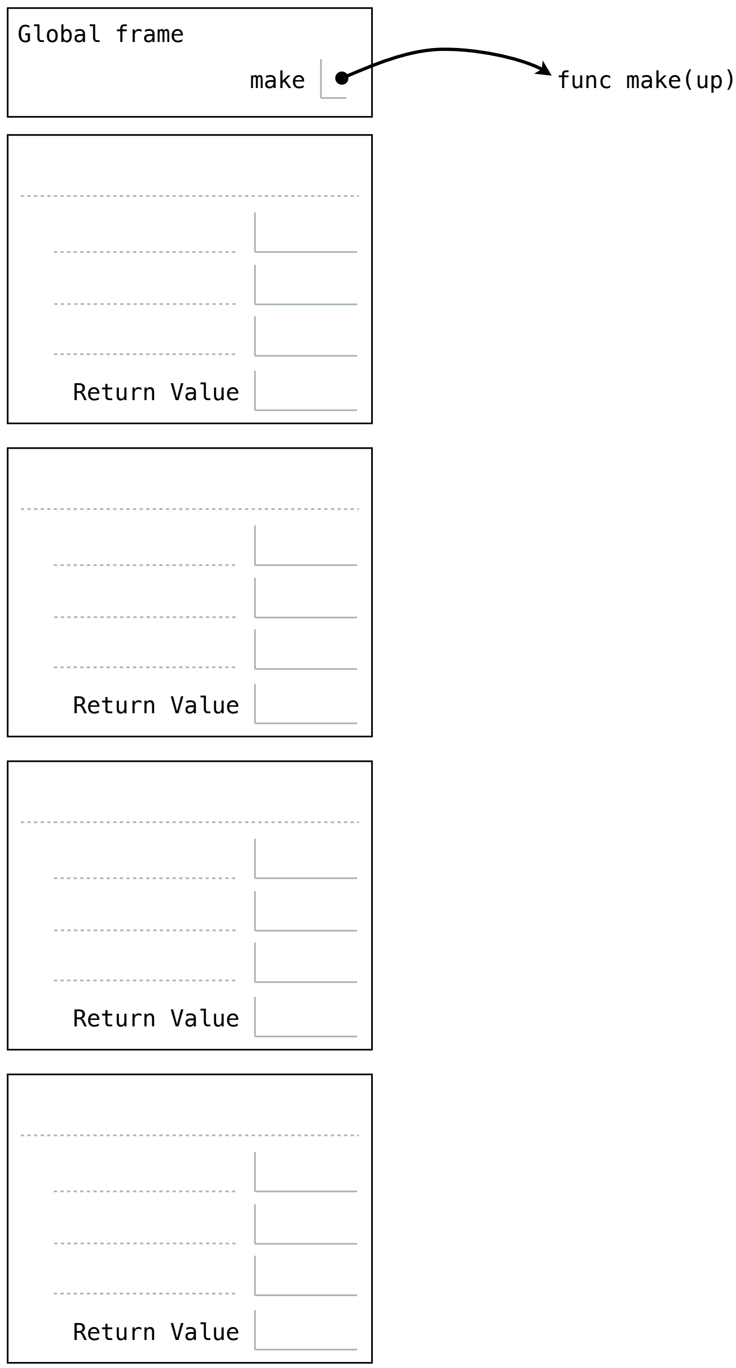
2. (12 points) Picture Frame

(a) (5 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 make(up):
    exam = [2012]
    def wed():
        nonlocal exam
        if exam[0] >= up:
            return list(exam)
        a = exam[0] + 1
        exam = [a, exam]
        return wed()
    return wed()

make(2014)
```

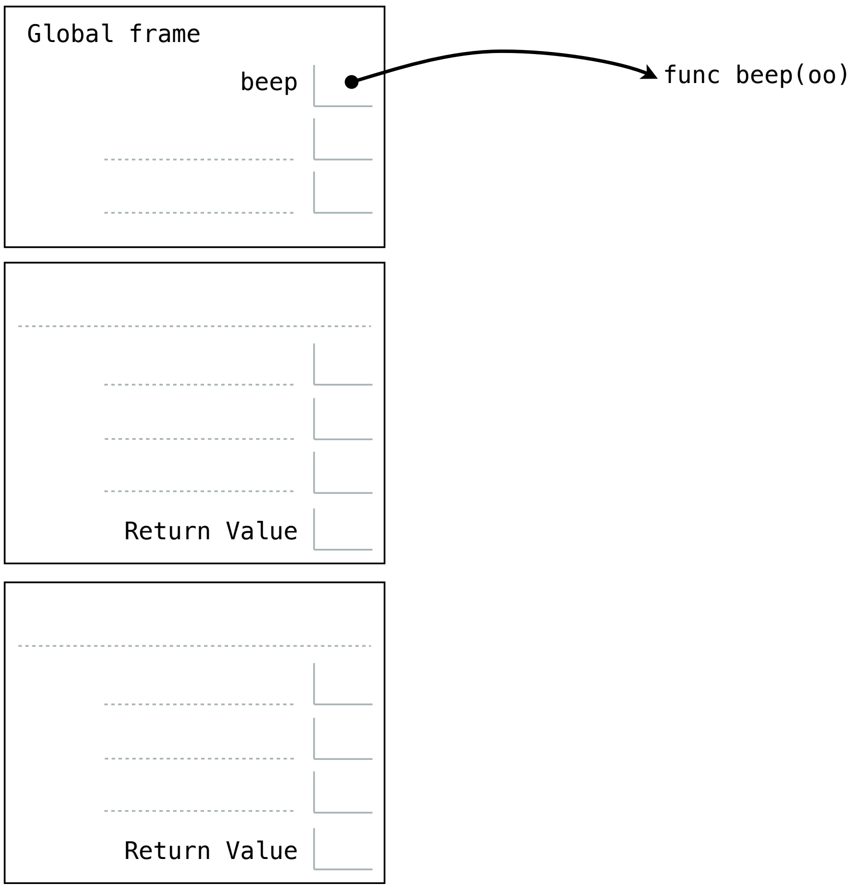


(b) (1 pt) Circle **True** or **False**: This diagram would change if the `nonlocal` statement were removed.

(c) **(5 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 beep(oo):
    b[b[oo]] = [b[ee], oo, b[ee]]
    return b[len(b)-1]

b = list(range(3, 7))
ee = 1
beep(0).append('not found')
```

(d) **(1 pt)** What will `print(b)` output after executing this code?

3. (14 points) Objets d'Art

(a) (6 pt) Cross out whole lines in the implementation below so that the doctests for `Vehicle` pass. In addition, cross out all lines that **have no effect**. Don't cross out docstrings, doctests, or decorators.

```
class Vehicle(object):
    """
    >>> c = Car('John', 'CS61A')
    >>> c.drive('John')
    John is driving
    >>> c.drive('Jack')
    Car stolen: John CS61A
    >>> c.pop_tire()
    3
    >>> c.pop_tire()
    2
    >>> c.fix()
    >>> c.pop_tire()
    3
    """
    def __init__(self, owner):
        self.owner = owner
    def move(self):
        print(self.owner + ' is driving')

class Car(Vehicle):
    tires = 4
    Car.tires = 4
    def __init__(self, owner, license_plate):
        Vehicle.__init__(owner)
        Vehicle.__init__(self, owner)
        self.plate = license_plate
        self.tires = tires
        self.tires = Car.tires
    def drive(self, person):
        if person != self.owner:
        if self.person != self.owner:
            print('Car stolen: ' + identification)
            print('Car stolen: ' + identification())
            print('Car stolen: ' + self.identification)
            print('Car stolen: ' + self.identification())
        else:
            Car.move(self)
    @property
    def identification(self):
        return self.owner + ' ' + self.plate
    def pop_tire(self):
        self.tires -= 1
        return self.tires
    def fix(self):
        setattr(Car, 'tires', self.tires)
        setattr(Car, 'tires', Car.tires)
        setattr(self, 'tires', self.tires)
        setattr(self, 'tires', type(self).tires)
        setattr(self, 'tires', self.Car.tires)
```

(b) **(4 pt)** The `max_path` function takes an instance of the `Tree` class from Study Guide 2. It is meant to return the maximal sum of internal `entry` values on a path from the *root* to a *leaf* of the tree.

```
def max_path(tree):
    """Return the sum of entries in a maximal path from the root to a leaf.

    >>> max_path(Tree(3, Tree(4), Tree(-2, Tree(8), Tree(3))))
    9
    >>> max_path(Tree(9, None, Tree(1, Tree(-2, Tree(5), Tree(2)), None)))
    13
    """
    paths = [0]
    if tree.right is not None:
        paths.append(max_path(tree.right))
    if tree.left is not None:
        paths.append(max_path(tree.left))
    if min(paths) < 0:
        return tree.entry + max([p for p in paths if p < 0])
    return tree.entry + max(paths)
```

Circle **True** or **False** to indicate whether each of the following statements about `max_path` is true.

i. (**True**/**False**) It returns the correct result for all doctests shown.

ii. (**True**/**False**) It returns the correct result for all valid trees with integer entries.

(c) **(4 pt)** Define a function `coerce` that takes a `Tree` instance and returns a dispatch dictionary representing the same tree using messages `left`, `right`, and `entry`. The base case has been provided for you.

```
def coerce(tree):
    """Return a dispatch dictionary representing the tree.

    >>> t = Tree(4, None, Tree(5, None, Tree(1, Tree(3, Tree(5), Tree(2)))))
    >>> d = coerce(t)
    >>> d['entry']
    4
    >>> d['left'] is None
    True
    >>> d['right']['right']['left']['entry']
    3
    """
    if tree is None:
        return None
```

4. (8 points) Form and Function

(a) (4 pt) You have been hired to work on AI at UnitedPusherElectric, the leading manufacturer of Pusher Bots. The latest model, PusherBot 5, keeps pushing people down stairs when it gets lost. Fix it!

Assume that you have an abstract data type `position` that combines `x` and `y` coordinates (in meters).

```
>>> pos = position(3, 4)
>>> x(pos)
3
>>> y(pos)
4
```

`pathfinder` should return a `visit` function that takes a `position` argument. `visit` returns `True` unless:

i. Its argument position is more than 6 meters from `position(0, 0)`, or

ii. Its argument position has been visited before.

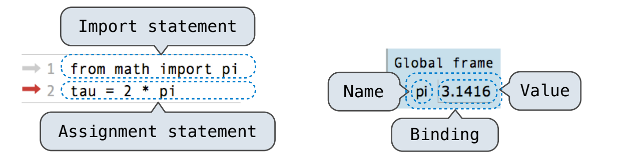
The implementation below is incorrect. Cross out each line (or part of a line) that must change and write a revised version next to it, so that `pathfinder` is correct **and** does not depend on the implementation of `position`. Assume your corrections have the same indentation as the lines they replace. You may not add or remove lines. Make as few changes as necessary.

```
from math import sqrt
def equal(position, other):
    return x(position) == x(other) and y(position) == y(other)

def pathfinder():
    """Return a visit function to help with path-finding.
    >>> visit1, visit2 = pathfinder(), pathfinder()
    >>> visit1(position(3, 4))
    True
    >>> visit1(position(5, 12)) # Too far away
    False
    >>> visit1(position(3, 4))  # Already visited
    False
    >>> visit2(position(3, 4))
    True
    """
    visited = ()

    def visit(pos):

        if sqrt( x(pos)*x(pos) + y(pos)*y(pos) ) > 6:

            return False

        for p in visit:

            if p == pos:

                return True

        visited.append(pos)

        return True

    return visited
```

(b) (4 pt) Fill in missing expressions in the implementation for `subwords`, which lists all of the `n` letter subwords of a given word `w`. A subword consists of some subset of the letters in a word, in their original order. You may assume that the word has no repeated letters. Some hints about string slicing appear in the doctest.

```
def subwords(w, n):
    """List all subwords of word w that have length n.

    >>> 'conc' + 'atenate'
    'concatenate'
    >>> w = 'spot'
    >>> w[len(w):]
    ''
    >>> subwords(w, 2)
    ['sp', 'so', 'st', 'po', 'pt', 'ot']
    >>> subwords(w, 3)
    ['spo', 'spt', 'sot', 'pot']
    """
    if n == 0:



        return ______________________________________________________

    if w == '':


        return ______________________________________________________



    r = [w[0] + x for x in __________________________________________]



    return r + ______________________________________________________
```

(c) (0 pt) Draw a picture of PusherBot 5.



Code (left):

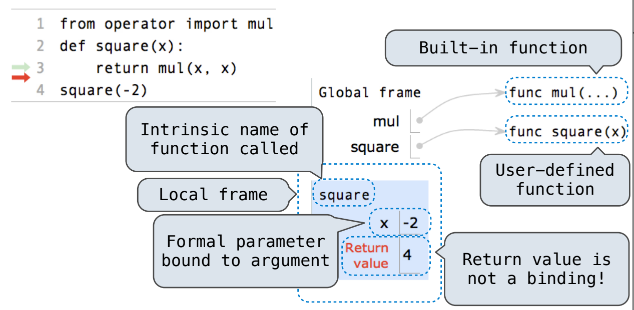
Statements and expressions

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

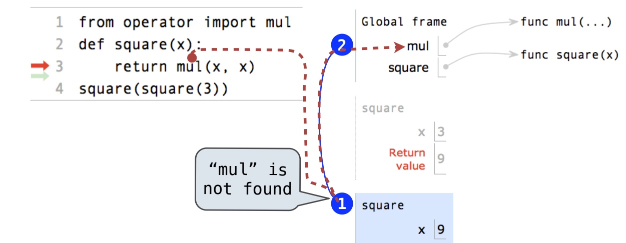
Frames (right):

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:

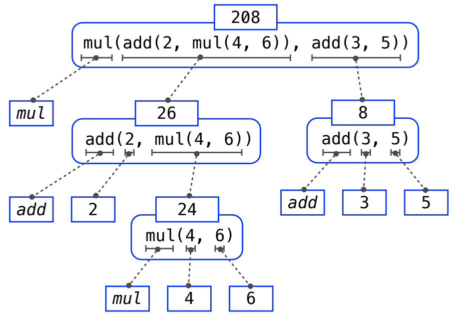
1. Evaluate the subexpression <left>.
2. If the result is a false value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

Evaluation rule for not expressions:

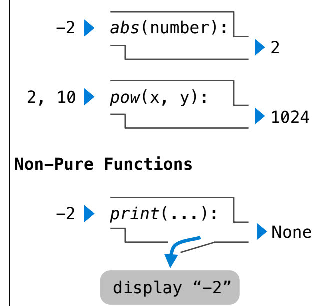
1. Evaluate <exp>; The value is True if the result is a false value, and False otherwise.

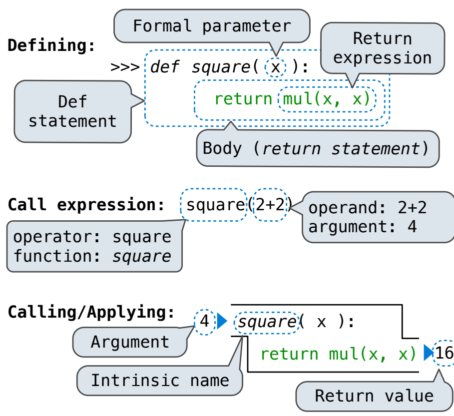
Execution rule for while statements:

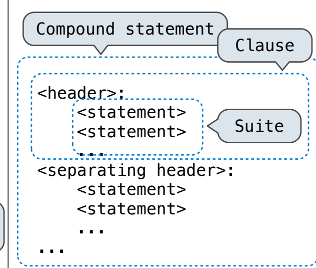
1. Evaluate the header's expression.
2. If it is a true value, execute the (*whole*) suite, then return to step 1.

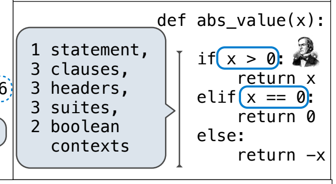


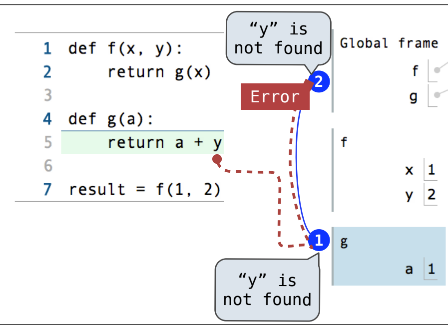
Pure Functions



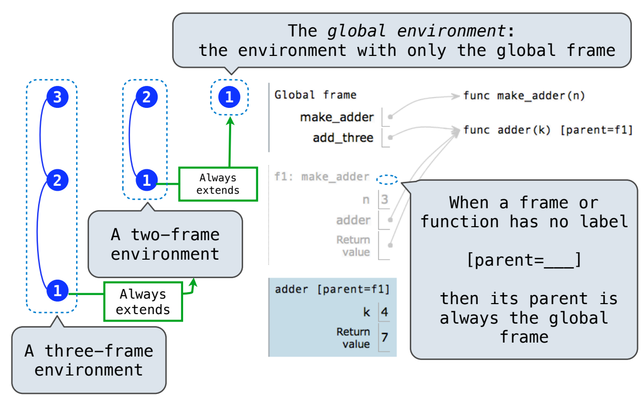




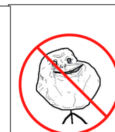




- An environment is a sequence of frames
- An environment for a non-nested function (no def within def) consists of one local frame, followed by the global frame

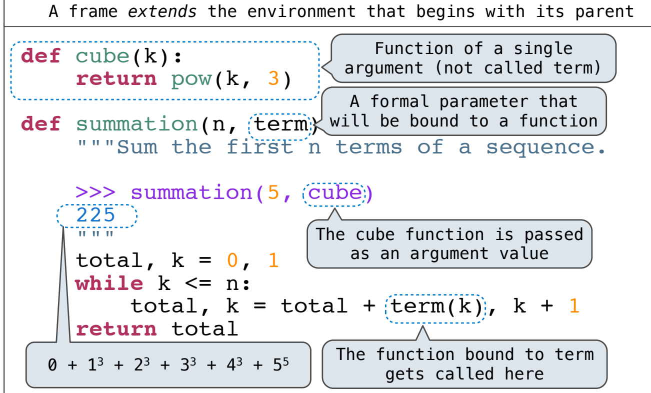


A frame *extends* the environment that begins with its parent



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
```

```
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

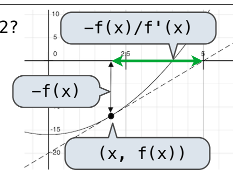
The name add_three is bound to a function

A local def statement

Can refer to names in the enclosing function

How to find the square root of 2?

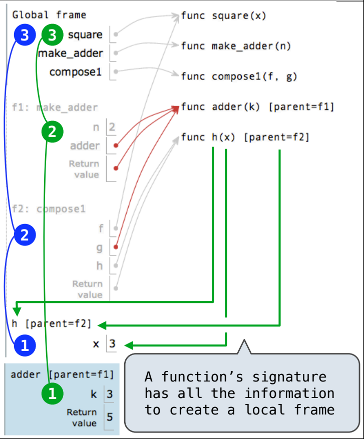
```
>>> f = lambda x: x*x - 2
>>> find_zero(f, 1)
1.4142135623730951
```



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 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

```
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?

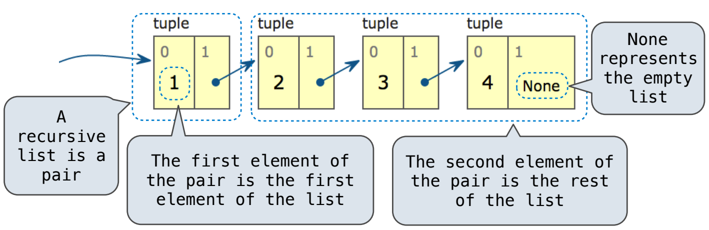
- Square takes one argument. **Yes**
- Square has the intrinsic name square. **No**
- Square computes the square of a number. **Yes**
- Square computes the square by calling mul. **No**

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

Constructor

Selectors

```
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)
```




```
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.

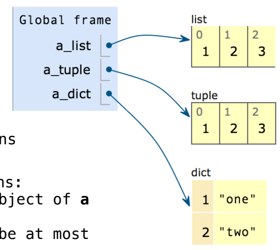
Element selection. A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

```
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

```
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.

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.

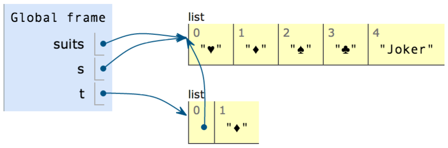
Python Docs: an "enclosing scope"

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.

```
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>]

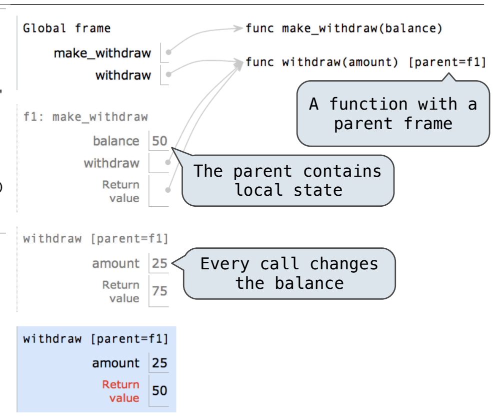
Short version: [<map exp> for <name> in <iter exp>]

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]
['♡', '♢', '♤', '♧']
```

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 balance
    return withdraw

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



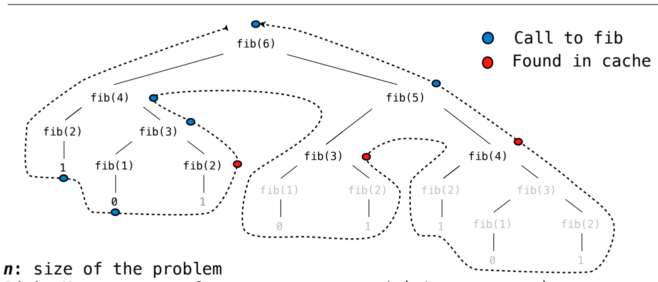
Python pre-computes which frame contains each name before executing the body of a function.

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



n: size of the problem
R(n): Measurement of some resource used (time or space)

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

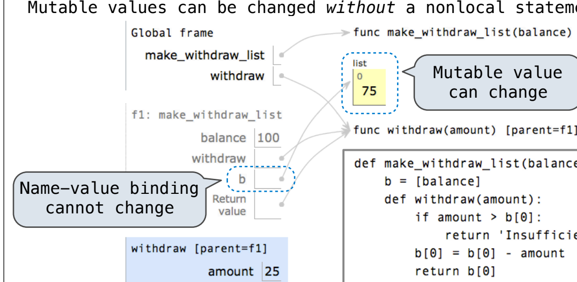
means that there are constants k_1 and k_2 such that

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

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

$\Theta(b^n)$ $\cdots$ $\Theta(n^3)$ $\Theta(n^2)$ $\Theta(n)$ $\Theta(\log n)$ $\Theta(1)$

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




```
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)
```

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

```
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**

```
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
```

```
def make_instance(cls):
    def get_value(name):
        if name in attributes:
            return attributes[name]
        else:
            value = cls['get'](name)
            return bind_method(value, instance)
    def set_value(name, value):
        attributes[name] = value
    attributes = {}
    instance = {'get': get_value, 'set': set_value}
    return instance
def bind_method(value, instance):
    if callable(value):
        def method(*args):
            return value(instance, *args)
        return method
    else:
        return value
def make_class(attributes={}, base_class=None):
    def get_value(name):
        if name in attributes:
            return attributes[name]
        elif base_class is not None:
            return base_class['get'](name)
    def set_value(name, value):
        attributes[name] = value
    def new(*args):
        return init_instance(cls, *args)
    cls = {'get': get_value, 'set': set_value, 'new': new}
    return cls
def init_instance(cls, *args):
    instance = make_instance(cls)
    init = cls['get']('__init__')
    if init is not None:
        init(instance, *args)
    return instance
def make_account_class():
    interest = 0.02
    def __init__(self, account_holder):
        self['set']('holder', account_holder)
        self['set']('balance', 0)
    def deposit(self, amount):
        new_balance = self['get']('balance') + amount
        self['set']('balance', new_balance)
        return self['get']('balance')
    ...
    return make_class(locals())
Account = make_account_class()
```

The class of the instance

Match name against instance attributes

Look up the name in the class

Assignment affects instance attributes

Class attribute lookup

Common dispatch dictionary pattern

Dispatch dictionary

Special constructor name is fixed here

```
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)

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)
```

```
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_leaves(tree):
    if type(tree) != tuple:
        return 1
    return sum(map(count_leaves, tree))
```