Every object is an arrow pointing to a box

A recursive list is a pair

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

for <name> in <expression>:
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, ...

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.

{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.

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.


Call to fib
Found in cache

from operator import add, sub, mul, truediv

def adder(a, b, c):
    """The constraint that a + b = c."""
    return make_ternary_constraint(a, b, c, add, sub, sub)
def multiplier(a, b, c):
    """The constraint that a * b = c."""
    return make_ternary_constraint(a, b, c, mul, div, div)

def make_ternary_constraint(a, b, c, add, sub, sub):
    """The constraint that add(a, b) = c."""
    new_constraint = constraint
    def new_value():
        return add(new_value(), new_value())
    return make_ternary_constraint(a, b, c, add, sub, sub)
To evaluate a dot expression: to evaluate an expression in the dot, which yields the object of the dot expression.
2. (name) is matched against the instance attributes of that object; if no 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.

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')
>>> jim_account.balance = 0
>>> jim_account.balance
0
```

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

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

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

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

class CheckingAccount(Account):
    withdraw_fee = 1
    interest = 0.02

def __init__(self, account_holder):
    super().__init__(account_holder)
    self.withdraw_fee = 0

    return Account.withdraw(self, amount) + self.withdraw_fee

class SavingsAccount(Account):
    deposit_fee = 2

def __init__(self, account_holder):
    super().__init__(account_holder)

def deposit(self, amount):
    return Account.deposit(self, amount) + self.deposit_fee
```

```
class AsSeenOnTVAccount(CheckingAccount, SavingsAccount):
    def __init__(self, account_holder):
        self.balance = 1

        # A free dollar!

    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'
```

```
class RList:
    class EmptyList:
        def __len__(self):
            return 0

        empty = EmptyList()  # The base case

    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]

    def map_rlist(s, f):
        if s is RList.empty:
            return RList.empty
        rest = map_rlist(s.rest, f)
        return RList(f(s.first), rest)

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

```
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
```

Type dispatching: Define a different function for each possible combination of types for which an operation is valid.
- `iscomplex(z)`
- `return type(z) in (ComplexRI, ComplexMA)`
- `isrational(z)`
- `return type(z) == Rational`