def rational(n, d):
    def select(name):
        if name == 'n':
            return n
        elif name == 'd':
            return d
    return select

def numerator(x):
    return x['n']

def denominator(x):
    return x['d']

Rational implementation using functions:

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

A combined expression that evaluates to a list using this evaluation procedure:
1. Add a new frame with the current frame as its parent
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

The result of calling repr on a value is what Python prints using the
repr function. The result list is a string.
```
### Python object system

**Idea:** All bank accounts have a balance and an account holder;

The Account class should add those attributes to each of its instances.

A new instance is created by calling a class

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

When a class is called:

1. A new instance of that class is created.
2. The `__init__` method 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.

```python
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, withdraw_fee=0.04):
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance
```

Method invocation: One object before the dot and other arguments within parentheses

```python
>>> Account.deposit(a, 5)
```

The `<expression>`, `<name>`, 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>`.

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.
3. If an attribute with that name exists, its value is returned.
4. If not, `<name>` is looked up in the class, which yields a class attribute value.
5. 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 the attributes of the object for 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

```python
class CheckingAccount(Account):
    """A bank account that charges for withdrawals.""
    withdraw_fee = 0.05
    interest = 0.08

    def withdraw(self, amount, withdraw_fee=0.04):
        return Account.withdraw(self, amount + amount * withdraw_fee)

    def deposit(self, amount):
        Account.deposit(self, amount + amount * withdraw_fee)
```

To lookup 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.

```python
>>> ch = CheckingAccount('Tom')  # Calls Account.__init__
>>> ch.withdraw(20)  # Found in Account
20
```

### Recursive description

- A tree has a root label and a list of branches.
- Each branch is a tree.
- A tree with zero branches is a leaf.

### Relative description

- Each location is a node.
- Each node has a label.
- One node can be the parent/child of another node.
- Each branch has three possibilities:
  - A leaf
  - A new instance
  - A list of branches

### Python code

```
def tree(label, branches=[]):
    # Verifies the tree definition
    return [label] + list(branches)

def branches(tree):
    return tree[1:]

def is_tree(tree):
    if type(tree) != list or len(tree) < 1:
        return False
    for branch in branches(tree):
        if not is_tree(branch):
            return False
    return True

def is_leaf(tree):
    return not branches(tree)

def leaves(tree):
    """The leaf values in t.""
    return (tree[1],) if is_leaf(tree) else
    sum((leaves(b) for b in branches(tree)), (tree[1],))

class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        for branch in branches:
            assert is_tree(branch)
        self.branches = list(branches)

    def __repr__(self):
        rest = ', ' + repr(self.rest) if self.rest else ''
        return 'Tree(' + repr(self.label) + rest + ')

    def __str__(self):
        string = '<' if self.rest is not None else '>
        return string + self.label + str(self.rest) + '>'

    def __len__(self):
        return len(self.label) + sum((len(b) for b in branches(self)), 0)

    def __and__(self):
        return Link(self.label, self.rest)
```

### Example

```
>>> s = tree(3, [tree(1), ...]
>>> >>> leaves(s)
(3, 1, 0, 1, 1, 1, 1)
```

### A binary search tree is a binary tree where each root is larger than all values in its left branch and smaller than all values in its right branch

- `BTree.empty()` returns `None`
- `BTree.empty()` returns `None`
- `BTree.empty()` returns `None`