tree.py

```python
# Copyright 2013, Michael H. Goldwasser
#
# Developed for use with the book:
#
#        Data Structures and Algorithms in Python
#        Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser
#        John Wiley & Sons, 2013
#
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program.  If not, see <http://www.gnu.org/licenses/>.

import collections

class Tree:
    """Abstract base class representing a tree structure."""

    #------------------------------- nested Position class -------------------------------
    class Position:
        """An abstraction representing the location of a single element within a tree.

        Note that two position instaces may represent the same inherent location in a tree.
        Therefore, users should always rely on syntax 'p == q' rather than 'p is q' when testing
        equivalence of positions.
        """

        def element(self):
            """Return the element stored at this Position."""
            raise NotImplementedError('must be implemented by subclass')

        def __eq__(self, other):
            """Return True if other Position represents the same location."""
            raise NotImplementedError('must be implemented by subclass')

        def __ne__(self, other):
            """Return True if other does not represent the same location."""
            return not (self == other)                       # opposite of __eq__

    # ---------- abstract methods that concrete subclass must support ----------
    def root(self):
        """Return Position representing the tree's root (or None if empty)."""
        raise NotImplementedError('must be implemented by subclass')

    def parent(self, p):
        """Return Position representing p's parent (or None if p is root)."""
        raise NotImplementedError('must be implemented by subclass')

    def num_children(self, p):
        """Return the number of children that Position p has."""
        raise NotImplementedError('must be implemented by subclass')
```

```python
    def children(self, p):
        """Generate an iteration of Positions representing p's children."""
        raise NotImplementedError('must be implemented by subclass')

    def __len__(self):
        """Return the total number of elements in the tree."""
        raise NotImplementedError('must be implemented by subclass')

    # ---------- concrete methods implemented in this class ----------
    def is_root(self, p):
        """Return True if Position p represents the root of the tree."""
        return self.root() == p

    def is_leaf(self, p):
        """Return True if Position p does not have any children."""
        return self.num_children(p) == 0

    def is_empty(self):
        """Return True if the tree is empty."""
        return len(self) == 0

    def depth(self, p):
        """Return the number of levels separating Position p from the root."""
        if self.is_root(p):
            return 0
        else:
            return 1 + self.depth(self.parent(p))

    def _height1(self):                               # works, but O(n^2) worst-case time
        """Return the height of the tree."""
        return max(self.depth(p) for p in self.positions() if self.is_leaf(p))

    def _height2(self, p):                                 # time is linear in size of
subtree
        """Return the height of the subtree rooted at Position p."""
        if self.is_leaf(p):
            return 0
        else:
            return 1 + max(self._height2(c) for c in self.children(p))

    def height(self, p=None):
        """Return the height of the subtree rooted at Position p.

        If p is None, return the height of the entire tree.
        """
        if p is None:
            p = self.root()
        return self._height2(p)                # start _height2 recursion

    def __iter__(self):
        """Generate an iteration of the tree's elements."""
        for p in self.positions():      # use same order as positions()
            yield p.element()           # but yield each element

    def positions(self):
        """Generate an iteration of the tree's positions."""
        return self.preorder()          # return entire preorder iteration

    def preorder(self):
        """Generate a preorder iteration of positions in the tree."""
        if not self.is_empty():
            for p in self._subtree_preorder(self.root()):     # start recursion
                yield p
```

```
    def _subtree_preorder(self, p):
        """Generate a preorder iteration of positions in subtree rooted at p."""
        yield p                           # visit p before its subtrees
        for c in self.children(p):                    # for each child c
            for other in self._subtree_preorder(c):             # do preorder of c's subtre
                yield other                                          # yielding each tc
ur caller

    def postorder(self):
        """Generate a postorder iteration of positions in the tree."""
        if not self.is_empty():
            for p in self._subtree_postorder(self.root()):     # start recursion
                yield p

    def _subtree_postorder(self, p):
        """Generate a postorder iteration of positions in subtree rooted at p."""
        for c in self.children(p):        # for each child c
            for other in self._subtree_postorder(c):  # do postorder of c's subtree
                yield other       # yielding each to our caller
        yield p             # visit p after its subtrees
```

binary_tree.py

```
# Copyright 2013, Michael H. Goldwasser
#
# Developed for use with the book:
#
#        Data Structures and Algorithms in Python
#        Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser
#        John Wiley & Sons, 2013
#
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program.  If not, see <http://www.gnu.org/licenses/>.

from tree import Tree

class BinaryTree(Tree):
    """Abstract base class representing a binary tree structure."""

    # --------------------- additional abstract methods ---------------------
    def left(self, p):
        """Return a Position representing p's left child.

        Return None if p does not have a left child.
        """
        raise NotImplementedError('must be implemented by subclass')

    def right(self, p):
        """Return a Position representing p's right child.

        Return None if p does not have a right child.
        """
        raise NotImplementedError('must be implemented by subclass')
```

```
    # ---------- concrete methods implemented in this class ----------
    def sibling(self, p):
        """Return a Position representing p's sibling (or None if no sibling)."""
        parent = self.parent(p)
        if parent is None:                                    # p must be the root
            return None                                         # root has no siblin
        else:
            if p == self.left(parent):
                return self.right(parent)                 # possibly None
            else:
                return self.left(parent)                    # possibly None

    def children(self, p):
        """Generate an iteration of Positions representing p's children."""
        if self.left(p) is not None:
            yield self.left(p)
        if self.right(p) is not None:
            yield self.right(p)

    def inorder(self):
        """Generate an inorder iteration of positions in the tree."""
        if not self.is_empty():
            for p in self._subtree_inorder(self.root()):
                yield p

    def _subtree_inorder(self, p):
        """Generate an inorder iteration of positions in subtree rooted at p."""
        if self.left(p) is not None:                   # if left child exists, traverse its
subtree
            for other in self._subtree_inorder(self.left(p)):
                yield other
        yield p                                                     # visit p betwee
ts subtrees
        if self.right(p) is not None:                 # if right child exists, traverse its
ubtree
            for other in self._subtree_inorder(self.right(p)):
                yield other

    # override inherited version to make inorder the default
    def positions(self):
        """Generate an iteration of the tree's positions."""
        return self.inorder()                                # make inorder the default
```

linked_binary_tree.py

```
# Copyright 2013, Michael H. Goldwasser
#
# Developed for use with the book:
#
#       Data Structures and Algorithms in Python
#       Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser
#       John Wiley & Sons, 2013
#
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
```

```python
#
# You should have received a copy of the GNU General Public License
# along with this program.    If not, see <http://www.gnu.org/licenses/>.

from binary_tree import BinaryTree

class LinkedBinaryTree(BinaryTree):
    """Linked representation of a binary tree structure."""

    #-------------------------- nested _Node class --------------------------
    class _Node:
        """Lightweight, nonpublic class for storing a node."""
        __slots__ = '_element', '_parent', '_left', '_right' # streamline memory usage

        def __init__(self, element, parent=None, left=None, right=None):
            self._element = element
            self._parent = parent
            self._left = left
            self._right = right

    #-------------------------- nested Position class --------------------------
    class Position(BinaryTree.Position):
        """An abstraction representing the location of a single element."""

        def __init__(self, container, node):
            """Constructor should not be invoked by user."""
            self._container = container
            self._node = node

        def element(self):
            """Return the element stored at this Position."""
            return self._node._element

        def __eq__(self, other):
            """Return True if other is a Position representing the same location."""
            return type(other) is type(self) and other._node is self._node

    #------------------------------- utility methods -------------------------------
    def _validate(self, p):
        """Return associated node, if position is valid."""
        if not isinstance(p, self.Position):
            raise TypeError('p must be proper Position type')
        if p._container is not self:
            raise ValueError('p does not belong to this container')
        if p._node._parent is p._node:            # convention for deprecated nodes
            raise ValueError('p is no longer valid')
        return p._node

    def _make_position(self, node):
        """Return Position instance for given node (or None if no node)."""
        return self.Position(self, node) if node is not None else None

    #-------------------------- binary tree constructor --------------------------
    def __init__(self):
        """Create an initially empty binary tree."""
        self._root = None
        self._size = 0

    #-------------------------- public accessors --------------------------
    def __len__(self):
        """Return the total number of elements in the tree."""
        return self._size

    def root(self):
        """Return the root Position of the tree (or None if tree is empty)."""
        return self._make_position(self._root)
```

```
    def parent(self, p):
        """Return the Position of p's parent (or None if p is root)."""
        node = self._validate(p)
        return self._make_position(node._parent)

    def left(self, p):
        """Return the Position of p's left child (or None if no left child)."""
        node = self._validate(p)
        return self._make_position(node._left)

    def right(self, p):
        """Return the Position of p's right child (or None if no right child)."""
        node = self._validate(p)
        return self._make_position(node._right)

    def num_children(self, p):
        """Return the number of children of Position p."""
        node = self._validate(p)
        count = 0
        if node._left is not None:          # left child exists
            count += 1
        if node._right is not None:         # right child exists
            count += 1
        return count

    #-------------------------- nonpublic mutators --------------------------
    def _add_root(self, e):
        """Place element e at the root of an empty tree and return new Position.

        Raise ValueError if tree nonempty.
        """
        if self._root is not None:
            raise ValueError('Root exists')
        self._size = 1
        self._root = self._Node(e)
        return self._make_position(self._root)

    def _add_left(self, p, e):
        """Create a new left child for Position p, storing element e.

        Return the Position of new node.
        Raise ValueError if Position p is invalid or p already has a left child.
        """
        node = self._validate(p)
        if node._left is not None:
            raise ValueError('Left child exists')
        self._size += 1
        node._left = self._Node(e, node)                                    # node is its pa
nt
        return self._make_position(node._left)

    def _add_right(self, p, e):
        """Create a new right child for Position p, storing element e.

        Return the Position of new node.
        Raise ValueError if Position p is invalid or p already has a right child.
        """
        node = self._validate(p)
        if node._right is not None:
            raise ValueError('Right child exists')
        self._size += 1
        node._right = self._Node(e, node)                                 # node is its pare
        return self._make_position(node._right)
```

```python
    def _replace(self, p, e):
        """Replace the element at position p with e, and return old element."""
        node = self._validate(p)
        old = node._element
        node._element = e
        return old

    def _delete(self, p):
        """Delete the node at Position p, and replace it with its child, if any.

        Return the element that had been stored at Position p.
        Raise ValueError if Position p is invalid or p has two children.
        """
        node = self._validate(p)
        if self.num_children(p) == 2:
            raise ValueError('Position has two children')
        child = node._left if node._left else node._right    # might be None
        if child is not None:
            child._parent = node._parent      # child's grandparent becomes parent
        if node is self._root:
            self._root = child                          # child becomes root
        else:
            parent = node._parent
            if node is parent._left:
                parent._left = child
            else:
                parent._right = child
        self._size -= 1
        node._parent = node                             # convention for deprecated node
        return node._element

    def _attach(self, p, t1, t2):
        """Attach trees t1 and t2, respectively, as the left and right subtrees of the exter
nal Position p.

        As a side effect, set t1 and t2 to empty.
        Raise TypeError if trees t1 and t2 do not match type of this tree.
        Raise ValueError if Position p is invalid or not external.
        """
        node = self._validate(p)
        if not self.is_leaf(p):
            raise ValueError('position must be leaf')
        if not type(self) is type(t1) is type(t2):       # all 3 trees must be same type
            raise TypeError('Tree types must match')
        self._size += len(t1) + len(t2)
        if not t1.is_empty():                 # attached t1 as left subtree of node
            t1._root._parent = node
            node._left = t1._root
            t1._root = None                         # set t1 instance to empty
            t1._size = 0
        if not t2.is_empty():                 # attached t2 as right subtree of node
            t2._root._parent = node
            node._right = t2._root
            t2._root = None                         # set t2 instance to empty
            t2._size = 0
```

In [1]:

```python
# Copyright 2013, Michael H. Goldwasser
#
# Developed for use with the book:
#
#       Data Structures and Algorithms in Python
#       Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser
#       John Wiley & Sons, 2013
#
```

```python
#
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.    See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program.    If not, see <http://www.gnu.org/licenses/>.

from linked_binary_tree import LinkedBinaryTree

class ExpressionTree(LinkedBinaryTree):
    """An arithmetic expression tree."""

    def __init__(self, token, left=None, right=None):
        """Create an expression tree.

        In a single parameter form, token should be a leaf value (e.g., '42'),
        and the expression tree will have that value at an isolated node.

        In a three-parameter version, token should be an operator,
        and left and right should be existing ExpressionTree instances
        that become the operands for the binary operator.
        """
        super().__init__()                                  # LinkedBinaryTree initia
zation
        if not isinstance(token, str):
            raise TypeError('Token must be a string')
        self._add_root(token)                               # use inherited, nonpublic
method
        if left is not None:                                # presumably three-paramete
form
            if token not in '+-*x/':
                raise ValueError('token must be valid operator')
            self._attach(self.root(), left, right)    # use inherited, nonpublic method

    def __str__(self):
        """Return string representation of the expression."""
        pieces = []                                # sequence of piecewise strings to compose
        self._parenthesize_recur(self.root(), pieces)
        return ''.join(pieces)

    def _parenthesize_recur(self, p, result):
        """Append piecewise representation of p's subtree to resulting list."""
        if self.is_leaf(p):
            result.append(str(p.element()))                           # leaf value as
string
        else:
            result.append('(')                                                 # op
ng parenthesis
            self._parenthesize_recur(self.left(p), result)         # left subtree
            result.append(p.element())                                     # operator
            self._parenthesize_recur(self.right(p), result)        # right subtree
            result.append(')')                                                 # cl
ng parenthesis

    def evaluate(self):
        """Return the numeric result of the expression."""
        return self._evaluate_recur(self.root())

    def _evaluate_recur(self, p):
        """Return the numeric result of subtree rooted at p."""
        if self.is_leaf(p):
            return float(p.element())             # we assume element is numeric
        else:
            op = p.element()
            left_val = self._evaluate_recur(self.left(p))
            right_val = self._evaluate_recur(self.right(p))
            if op == '+':
                return left_val + right_val
            elif op == '-':
                return left_val - right_val
            elif op == '/':
```

```
            elif op == '/':
                return left_val / right_val
            else:                                                       # treat 'x' or '*' as
multiplication
                return left_val * right_val


def tokenize(raw):
    """Produces list of tokens indicated by a raw expression string.

    For example the string '(43-(3*10))' results in the list
    ['(', '43', '-', '(', '3', '*', '10', ')', ')']
    """
    SYMBOLS = set('+-x*/() ')        # allow for '*' or 'x' for multiplication

    mark = 0
    tokens = []
    n = len(raw)
    for j in range(n):
        if raw[j] in SYMBOLS:
            if mark != j:
                tokens.append(raw[mark:j])     # complete preceding token
            if raw[j] != ' ':
                tokens.append(raw[j])              # include this token
            mark = j+1                                  # update mark to being at next index
    if mark != n:
        tokens.append(raw[mark:n])             # complete preceding token
    return tokens

def build_expression_tree(tokens):
    """Returns an ExpressionTree based upon by a tokenized expression.

    tokens must be an iterable of strings representing a fully parenthesized
    binary expression, such as ['(', '43', '-', '(', '3', '*', '10', ')', ')']
    """
    S = []                                          # we use Python list as stack
    for t in tokens:
        if t in '+-x*/':                            # t is an operator symbol
            S.append(t)                                         # push the operator
symbol
        elif t not in '()':                                     # consider t to be a li
ral
            S.append(ExpressionTree(t))                     # push trivial tree storing
value
        elif t == ')':              # compose a new tree from three constituent parts
            right = S.pop()                                         # right subtree as
per LIFO
            op = S.pop()                                                # operator symb
            left = S.pop()                                            # left subtree
            S.append(ExpressionTree(op, left, right)) # repush tree
        # we ignore a left parenthesis
    return S.pop()
```

In [2]:

```
unit = build_expression_tree(tokenize('(1+2)'))
print(unit, '=', unit.evaluate())
```

```
(1+2) = 3.0
```

In [3]:

```
small = build_expression_tree(tokenize('((2x(5-1))+(3x10))'))
print(small, '=', small.evaluate())
```

```
((2x(5-1))+(3x10)) = 38.0
```

In [4]:

```
big = build_expression_tree(tokenize('((((3+1)x3)/((9-5)+2))-((3x(7-4))+6))'))
print(big, '=', big.evaluate())
```

```
((((3+1)x3)/((9-5)+2))-((3x(7-4))+6)) = -13.0
```

```
((((3+1)x3)/((9-5)+2))-((3x(7-4))+6)) = -13.0
```

In [5]:

```python
for item in small.preorder():
    print(item.element())
```

```
+
x
2
-
5
1
x
3
10
```

In [6]:

```python
for item in small.postorder():
    print(item.element())
```

```
2
5
1
-
x
3
10
x
+
```

In [7]:

```python
for item in small.inorder():
    print(item.element())
```

```
2
x
5
-
1
+
3
x
10
```

In [ ]:

```python

```