Basic operators

True and False, True or False, not True
1 + 3, 1 - 3, 1 * 3
5 / 2 == 2.5, 5 // 2 == 2, 5 % 2 == 1
'hi' + 'bye'  # 'hbye'
[1, 2, 3] + [4, 5, 6]  # [1, 2, 3, 4, 5, 6]

List methods

lst = [1, 2, 3]
len(lst)  # 3
lst[0]  # 1
lst[0:2]  # [1, 2]
lst[0] = 'howdy'  # lst == ['howdy', 2, 3]
lst.append(29)  # lst == ['howdy', 2, 3, 29]
lst.pop()  # lst == ['howdy', 2, 3], returns 29
lst.pop(1)  # lst == ['howdy', 3], returns 2
lst.insert(1, 100)  # lst == ['howdy', 100, 3]
lst.extend([4, 5])  # lst == ['howdy', 100, 3, 4, 5]
3 in lst  # returns True

Dictionary methods

d = {'hi': 4, 'bye': 100}
d['hi']  # 4
d[100]  # raises KeyError!
'hi' in d  # True
4 in d  # False
d['howdy'] = 15  # adds new key-value pair
d['hi'] = -100  # changes a key-value pair

Exceptions

class MyCustomError(Exception):
    # Override __str__ to customize the error message.
raise MyCustomError

Stacks and Queues

s = Stack()
s.is_empty()  # Returns False if stack is empty.
s.push(10)
s.pop()  # Raises an EmptyStackError if stack is empty.
q = Queue()
q.is_empty()  # Returns False if queue is empty.
q.enqueue(10)
q.dequeue()  # Returns None if queue is empty.

Classes

class Point:
x: float
y: float
def __init__(self, x: float, y: float) -> None:
    self.x = x
    self.y = y
def size(self) -> float:
    return (self.x ** 2 + self.y ** 2) ** 0.5

p = Point(3, 4)  # create an instance
p.x  # attribute access: evaluates to 3
p.size()  # method call: returns 5.0

class MyWeirdClass(Point):  # inheritance
    pass

Linked Lists

class _Node:
    # A node in a linked list.
    # Attributes
    # item: The data stored in this node.
    # next:
    #   The next node in the list, or None if there are
    #   no more nodes in the list.

    item: Any
    next: Optional[_Node]
def __init__(self, item: Any) -> None:
    # Initialize a new node storing <item>,
    # with no 'next' node.

class LinkedList:
    # A linked list implementation of the List ADT.

    _first: Optional[_Node]
def __init__(self, items: list) -> None:
    # Initialize a linked list with the given items.
    # The first node in the linked list, or None if the list is empty.

Summation identities

\[
\sum_{i=1}^{n} i = \frac{n(n+1)}{2}
\]
Trees

class Tree:
    # === Private Attributes ===
    _root: The item stored at this tree's root, or None if this tree is empty.
    _root: Optional[Any]
    _subtrees: The list of all subtrees of this tree.
    _subtrees: List[Tree]

    # === Representation Invariants ===
    # - If self._root is None then self._subtrees is an empty list.
    # This setting of attributes represents an empty tree.
    # - self._subtrees does not contain any empty trees.

def __init__(self, root: Optional[Any], subtrees: List[Tree]) -> None:
    """Initialize a new Tree with the given root value and subtrees.
    
    If <root> is None, the tree is empty.
    Precondition: if <root> is None, then <subtrees> is empty.
    ""

def is_empty(self) -> bool:
    """Return whether this tree is empty.""

Binary Search Trees

class BinarySearchTree:
    # === Private Attributes ===
    _root: The item stored at the root of the tree, or None if the tree is empty.
    _root: Optional[Any]
    _left: The left subtree, or None if the tree is empty.
    _left: Optional[BinarySearchTree]
    _right: The right subtree, or None if the tree is empty.
    _right: Optional[BinarySearchTree]

    # === Representation Invariants ===
    # - If self._root is None, then so are self._left and self._right.
    # This represents an empty BST.
    # - If self._root is not None, then self._left and self._right are BinarySearchTrees.
    # - (BST Property) If self is not empty, then
    #   all items in self._left are <= self._root, and
    #   all items in self._right are >= self._root.

    def __init__(self, root: Optional[Any]) -> None:
        """Initialize a new BST containing only the given root value.

        If <root> is None, initialize an empty tree.
        ""

Abstract Syntax Trees

class Statement:
    """An abstract class representing a Python statement.""

def evaluate(self, env: Dict[str, Any]) -> Optional[Any]:
    """Evaluate this statement with the given environment.""

class Expr(Statement):
    """An abstract class representing a Python expression.""