Basic operators

**True and False, True or False, not True**

\[
\begin{align*}
1 + 3 &= 4, 1 - 3 &= -2, 1 \times 3 &= 3 \\
5 / 2 &= 2.5, 5 \div 2 &= 2.5, 5 \mod 2 &= 1 \\
'hi' + 'bye' &= \text{ ''hi''bye''} \\
[1, 2, 3] + [4, 5, 6] &= [1, 2, 3, 4, 5, 6]
\end{align*}
\]

**List methods**

```python
lst = [1, 2, 3]
len(lst) # 3
lst[0] # 1
lst[0:2] # [1, 2]
```

```python
lst[0] = 'howdy' # lst == ['howdy', 2, 3]
lst.append(29) # lst == ['howdy', 3, 4, 5]
lst.insert(1, 100) # lst == ['howdy', 100, 3, 4, 5]
lst.extend([4, 5]) # lst == ['howdy', 100, 3, 4, 5]
```

**Dictionary methods**

```python
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**

```python
class MyCustomError(Exception):
    pass
raise MyCustomError
```

**Classes**

```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def size(self):
    return (self.x ** 2 + self.y ** 2) ** 0.5
```

```python
class MyWeirdClass(Point): # inheritance
    pass
```

**Linked List**

```python
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: object
    next: Optional[_Node]

def __init__(self, item: object) -> None:
    """Initialize a new node storing <item>, with no 'next' node."
    ""
```

```python
class LinkedList:
    """A linked list implementation of the List ADT."
    
    # === Private Attributes ===
    # _first: The first node in the linked list, or None if the list is empty.
    _first: Optional[_Node]

def __init__(self, items: list) -> None:
    """Initialize a linked list with the given items. The first node in the linked list contains the first item in <items>."
    ""
```

**Stack and Queues**

```python
s = Stack()
s.is_empty() # Returns True if the stack is empty.
s.push(10)
s.pop() # Raises an EmptyStackError if the stack is empty.
q = Queue()
q.is_empty() # Returns True if the Queue is empty.
q.enqueue(10)
q.dequeue() # Returns None if the Queue is empty
```
class Tree:
    # === Private Attributes ===
    # The item stored at this tree's root, or None if the tree is empty.
    _root: Optional[object]
    # 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 may be empty when self._root is not None.
    # This setting of attributes represents a tree consisting of just one
    # node.

def __init__(self, root: object, 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 True if this tree is empty."""

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

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

def __init__(self, root: Optional[object]) -> None:
    """Initialize a new BST with the given root value and no children.
    If <root> is None, make an empty tree, with subtrees that are None.
    If <root> is not None, make a tree with subtrees are empty trees.
    """

def is_empty(self) -> bool:
    """Return True if this BST is empty."""