Data types

None
3, -4, 1.01, -2.0
True, False
'Hello, world!\n'
[1, 2.0, 'hi']
{'hi': 3, 'bye': 100}

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' # 'hibye'
[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]
"hello" # lst == ['hello', 2, 3]
lst.append(29) # lst == ['hello', 2, 3, 29]
lst.pop() # lst == ['hello', 2, 3], returns 29
lst.pop(1) # lst == ['hello', 3], returns 2
lst.insert(1, 100) # lst == ['hello', 100, 3]
lst.extend([4, 5]) # lst == ['hello', 100, 3, 4, 5]
3 in lst # returns True

Control flow

if x == 5:
    y = 1
elif 4 <= 100:
    z = 2
else:
    y = 100

for i in [0, 1, 2, 3]:  # or, "for i in range(4):
    print(i)

j = 1
while j < 10:
    print(j)
    j = j * 2

Exceptions

raise IndexError()

Class syntax

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

p = Point(3, 4)  # constructor
p.x # attribute access: returns 3
p.size() # method call: returns 5.0

class MyWeirdClass(Point):  # inheritance
    pass

Linked List (iterative)

class _Node:
    
    """A node in a linked list."
    
    === Attributes ===
    @type item: object
    The data stored in this node.
    @type next: _Node | None
    The next node in the list, or None if there are
    no more nodes in the list.
    """

    def __init__(self, item):
        """Initialize a new node storing <item>,
        with no 'next' node.
        @type self: _Node
        @type item: object
        @rtype: None"

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

    # === Private Attributes ===
    # @type _first: _Node | None
    # The first node in the list, or None if the list is empty.

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

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

        raise IndexError()
class LinkedListRec:
    """A recursive linked list.""
    # === Private Attributes ===
    # @type _first: object | None
    #    The first item in the list, or None
    # if the linked list is empty.
    # @type _rest: LinkedListRec | None
    #    A list containing the other items after the
    #    first one, or None if the linked list is empty.
    # === Representation Invariants ===
    # - _first is None if and only if _rest is None.
    # This situation represents an empty list.

def __init__(self, items):
    """Initialize a new linked list containing the
    given items.
    
    The first item in the linked list is the
    first item in <items>.
    
    @type self: LinkedListRec
    @type items: list
    @rtype: None"

    @type self: LinkedListRec
    @rtype: None"

def is_empty(self):
    """Return whether this linked list is empty.
    
    @type self: LinkedListRec
    @rtype: bool"

---

Stack and Queues

s = Stack()
s.is_empty()
s.push(10)
s.pop()
q = Queue()
q.is_empty()
q.enqueue(10)
q.dequeue()

---

Tree

class Tree:
    # === Private Attributes ===
    # @type _root: object | None
    #    The tree’s root item, or None.
    # @type _subtrees: list[Tree]
    #    A list of all subtrees of the tree.
    #    
    # === Representation Invariants ===
    # - If _root is None then _subtrees is empty.
    # This represents an empty Tree.
    # - _subtrees doesn’t contain any empty trees

def __init__(self, root):
    """Initialize a new Tree with the given root value.
    If <root> is None, the tree is empty.
    
    @type self: Tree
    @type root: object | None
    @rtype: None"

    @type self: Tree
    @rtype: None"

def is_empty(self):
    """Return whether this tree is empty.
    
    @type self: Tree
    @rtype: bool"

---

BinarySearchTree class

class BinarySearchTree:
    # === Private Attributes ===
    # @type _root: object | None
    #    The BST’s root value, or None.
    # @type _left: BinarySearchTree | None
    #    The left subtree, or None.
    # @type _right: BinarySearchTree | None
    #    The right subtree, or None.
    # === Representation Invariants ===
    # - If _root is None, then so are _left and _right.
    # This represents an empty BST.
    # - If _root is not None, then _left, _right are BSTs.
    # - Every item in _left is <= _root, and
    # every item in _right is >= _root.

def __init__(self, root):
    """Initialize a new BST with a given root value.
    If <root> is None, the BST is empty.
    
    @type self: BinarySearchTree
    @type root: object | None
    @rtype: None"

    @type self: BinarySearchTree
    @rtype: None"

def is_empty(self):
    """Return whether this tree is empty.
    
    @type self: BinarySearchTree
    @rtype: bool"""