Short Python function/method descriptions, and classes

```
__builtins__:
  len(x) -> integer
    Return the length of the list, tuple, dict, or string x.
  max(L) -> value
    Return the largest value in L.
  min(L) -> value
    Return the smallest value in L.
  range([start], stop, [step]) -> list of integers
    Return a list containing the integers starting with start and
    ending with stop - 1 with step specifying the amount to increment
    (or decrement). If start is not specified, the list starts at 0.
    If step is not specified, the values are incremented by 1.
  sum(L) -> number
    Returns the sum of the numbers in L.

dict:
  D[k] -> value
    Return the value associated with the key k in D.
  k in d -> boolean
    Return True if k is a key in D and False otherwise.
  D.get(k) -> value
    Return D[k] if k in D, otherwise return None.
  D.keys() -> list of keys
    Return the keys of D.
  D.values() -> list of values
    Return the values associated with the keys of D.
  D.items() -> list of (key, value) pairs
    Return the (key, value) pairs of D, as 2-tuples.

float:
  float(x) -> floating point number
    Convert a string or number to a floating point number, if
    possible.

int:
  int(x) -> integer
    Convert a string or number to an integer, if possible. A floating
    point argument will be truncated towards zero.

list:
  x in L -> boolean
    Return True if x is in L and False otherwise.
  L.append(x)
    Append x to the end of list L.
  L1.extend(L2)
    Append the items in list L2 to the end of list L1.
  L.index(value) -> integer
    Return the lowest index of value in L.
```

```
  L.insert(index, x)
    Insert x at position index.
  L.pop()
    Remove and return the last item from L.
  L.remove(value)
    Remove the first occurrence of value from L.
  L.sort()
    Sort the list in ascending order.

Module random:
  randint(a, b)
    Return random integer in range [a, b], including both end points.

str:
  x in s -> boolean
    Return True if x is in s and False otherwise.
  str(x) -> string
    Convert an object into its string representation, if possible.
  S.count(sub[, start[, end]]) -> int
    Return the number of non-overlapping occurrences of substring sub
    in string S[start:end]. Optional arguments start and end are
    interpreted as in slice notation.
  S.find(sub[,i]) -> integer
    Return the lowest index in S (starting at S[i], if i is given)
    where the string sub is found or -1 if sub does not occur in S.
  S.split([sep]) -> list of strings
    Return a list of the words in S, using string sep as the separator
    and any whitespace string if sep is not specified.

set:
  {1, 2, 3, 1, 3} -> {1, 2, 3}
  s.add(...)
    Add an element to a set
  set()
    Create a new empty set object
  x in s
    True iff x is an element of s

list comprehension:
   [<expression with x> for x in <list or other iterable>]

functional if:
   <expression 1> if <boolean condition> else <expression 2>
   -> <expression 1> if the boolean condition is True,
      otherwise <expression 2>
```

```
class LinkedListNode:
    """
    Node to be used in linked list

    === Attributes ===
    @param LinkedListNode next_: successor to this LinkedListNode
    @param object value: data this LinkedListNode represents
    """

    def __init__(self, value, next_=None):
        """
        Create LinkedListNode self with data value and successor next_.

        @param LinkedListNode self: this LinkedListNode
        @param object value: data of this linked list node
        @param LinkedListNode|None next_: successor to this LinkedListNode.
        @rtype: None
        """
        self.value, self.next_ = value, next_

    def __str__(self):
        """
        Return a user-friendly representation of this LinkedListNode.

        @param LinkedListNode self: this LinkedListNode
        @rtype: str

        >>> n = LinkedListNode(5, LinkedListNode(7))
        >>> print(n)
        5 -> 7 ->|
        """
        # start with a string s to represent current node.
        s = "{} ->".format(self.value)
        # create a reference to "walk" along the list
        current_node = self.next_
        # for each subsequent node in the list, build s
        while current_node is not None:
            s += " {} ->".format(current_node.value)
            current_node = current_node.next_
        # add "|" at the end of the list
        assert current_node is None, "unexpected non_None!!!"
        s += "|"
        return s
```

```
class LinkedList:
    """
    Collection of LinkedListNodes

    === Attributes ==
    @param: LinkedListNode front: first node of this LinkedList
    @param LinkedListNode back: last node of this LinkedList
    @param int size: number of nodes in this LinkedList
                        a non-negative integer
    """

    def __init__(self):
        """
        Create an empty linked list.

        @param LinkedList self: this LinkedList
        @rtype: None
        """
        self.front, self.back = None, None
        self.size = 0

    def append(self, value):
        """
        Insert a new LinkedListNode with value after self.back.

        @param LinkedList self: this LinkedList.
        @param object value: value of new LinkedListNode
        @rtype: None

        >>> lnk = LinkedList()
        >>> lnk.append(5)
        >>> lnk.size
        1
        >>> print(lnk.front)
        5 ->|
        >>> lnk.append(6)
        >>> lnk.size
        2
        >>> print(lnk.front)
        5 -> 6 ->|
        """
        new_node = LinkedListNode(value)
        if self.front is None:
            # append to an empty LinkedList
            self.front = self.back = new_node
        else:
            # self.back better not be None
            assert self.back, 'Unexpected None node'
            self.back.next_ = new_node
            self.back = new_node
        self.size += 1
```

```
class Tree:
    """
    A bare-bones Tree ADT that identifies the root with the entire tree.

    === Attributes ===
    @param object value: value stored in a Tree node
    @param list[Tree] children: list of children
    """

    def __init__(self, value=None, children=None):
        """
        Create Tree self with content value and 0 or more children

        @param Tree self: this tree
        @param object value: value contained in this tree
        @param list[Tree] children: possibly-empty list of children
        @rtype: None
        """
        self.value = value
        # copy children if not None
        self.children = children.copy() if children else []


# helper function
def gather_lists(list_):
    """
    Concatenate all the sublists of L and return the result.

    @param list[list[object]] list_: list of lists to concatenate
    @rtype: list[object]

    >>> gather_lists([[1, 2], [3, 4, 5]])
    [1, 2, 3, 4, 5]
    >>> gather_lists([[6, 7], [8], [9, 10, 11]])
    [6, 7, 8, 9, 10, 11]
    """
    new_list = []
    for l in list_:
        new_list += l
    return new_list
```

```
class BinaryTree:
    """
    A Binary Tree, i.e. arity 2.

    === Attributes ===
    @param object data: data in this Tree node
    @param BinaryTree|None left: left child
    @param BinaryTree|None right: right child
    """

    def __init__(self, data, left=None, right=None):
        """
        Create BinaryTree self with data and children left and right.

        None represents an empty tree.

        @param BinaryTree self: this binary tree
        @param object data: data of this node
        @param BinaryTree|None left: left child
        @param BinaryTree|None right: right child
        @rtype: None
        """
        self.data, self.left, self.right = data, left, right
```