Tree class

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

    === Attributes ===
    value - value  of root node
    children - root nodes of children
    """
    value: object
    children: List["Tree"]

    def __init__(self, value: object, children: List["Tree"]=None) -> None:
        """
        Create Tree self with content value and 0 or more children
        """
        self.value = value
        # copy children if not None
        self.children = children[:] if children is not None else []
```

BTNode class

```
class BTNode:
    """Binary Tree node.

    === Attributes ===
    data - data this node represents
    left - left child
    right - right child
    """
    data: object
    left: Union["BTNode", None]
    right: Union["BTNode", None]

    def __init__(self, data: object,
                 left: Union["BTNode", None]=None,
                 right: Union["BTNode", None]=None) -> None:
        """ Create BTNode (self) with data and children left and right.

        An empty BTNode is represented by None.
        """
        self.data, self.left, self.right = data, left, right

    def __repr__(self) -> str:
        """ Represent BTNode (self) as a string that can be evaluated to
        produce an equivalent BTNode.

        >>> BTNode(1, BTNode(2), BTNode(3))
        BTNode(1, BTNode(2, None, None), BTNode(3, None, None))
        """
        return ’BTNode({}, {}, {})’.format(self.data, repr(self.left), repr(self.right))
```

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) -> None
    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.pop(i)
    Remove and return L[i]
  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
  {1, 2, 3}.union({2, 4}) -> {1, 2, 3, 4}
  {1, 2, 3}.intersection({2, 4}) -> {2}
  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>
```