### Write recursive evaluate method

**first...**

Read over the _init_ method for class BTNode:

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

def __init__(self, value, left=None, right=None):
    """
    Create BinaryTree self with value and children left and right.
    
    @param BinaryTree self: this binary tree
    @param object value: value of this node
    @param BinaryTree|None left: left child
    @param BinaryTree|None right: right child
    @rtype: None
    """
    self.value, self.left, self.right = value, left, right
```

**next...**

Now, read the header and docstring for the function evaluate, and then answer the questions that follow it.

```python
def evaluate(b):
    """
    Evaluate the expression rooted at b. If b is a leaf, return its float value. Otherwise, evaluate b.left and b.right and combine them with b.value.
    
    Assume: -- b is a non-empty binary tree
    -- interior nodes contain value in {"+", "-", "+", "/"}
    -- interior nodes always have two children
    -- leaves contain float value
    
    @param BinaryTree b: binary tree representing arithmetic expression
    @rtype: float
    """

>>> b = BinaryTree(3.0)
>>> evaluate(b)
3.0
>>> b = BinaryTree("*", BinaryTree(3.0), BinaryTree(4.0))
>>> evaluate(b)
12.0

1. One of the examples in evaluate docstring is simple enough not to require recursion (a base case). Write an if... expression that checks for this case, and then returns the correct thing. Include an else... for when the tree is _less_ easy to deal with.
2. Another docstring examples is a typical one which can benefit from recursion. Write code that returns the correct value for this case. Hint: it may be helpful to use the built-in `eval` function, which takes a string Python expression and evaluates it.

Now implement the body of `evaluate`