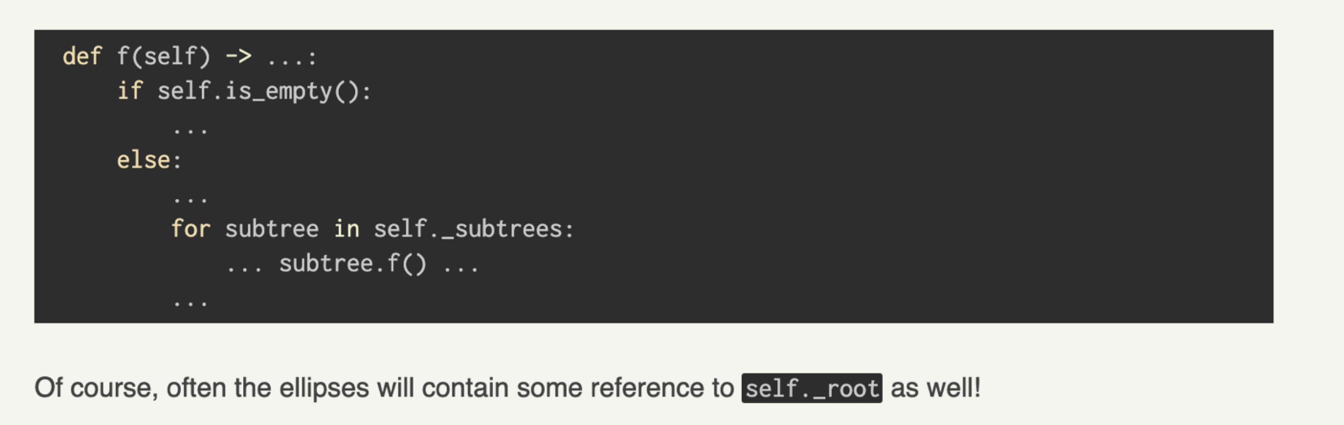
6.2 A Tree Implementation

RECURSION ON TREES :

- **T**he size of a non empty tree:
  - The *sum of the sizes of its subtrees plus 1* (root)
- The size of an empty tree
  - zero

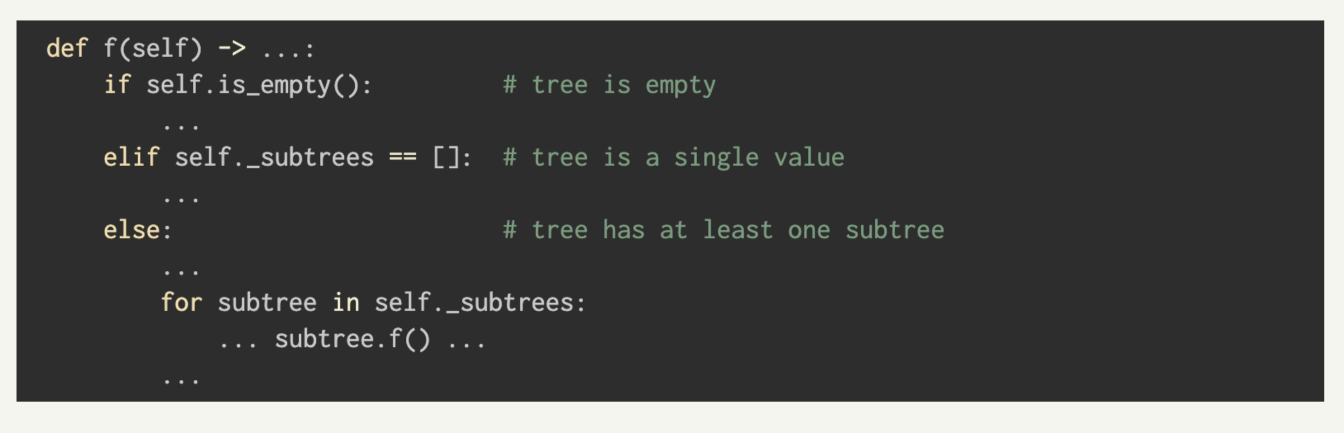
TEMPLATE FOR RECURSIVE METHODS ON TREES:



```
def f(self) -> ...:
    if self.is_empty():
        ...
    else:
        ...
        for subtree in self._subtrees:
            ... subtree.f() ...
        ...
```

Of course, often the ellipses will contain some reference to `self._root` as well!

- **Sometimes we add the case where the tree is a single item... if we are doing stuff to that item.**
- **THE GO TO FIRST DRAFT :)**



```
def f(self) -> ...:
    if self.is_empty():         # tree is empty
        ...
    elif self._subtrees == []:  # tree is a single value
        ...
    else:                       # tree has at least one subtree
        ...
        for subtree in self._subtrees:
            ... subtree.f() ...
        ...
```

OPTIONAL PARAMETERS:

- You can give a function optional parameters by writing = default value beside the type.
- All optional parameters must appear after all of the required parameter sin the function header
- DON'T use mutable values for your optional parameters

THE IMPLEMENTATION:

```
from __future__ import annotations
from typing import Any, Optional, List
```

```python
class Tree:
    """ A recursive tree data structure.
    === Private Attributes ===
    _root:
    The item stored st this tree' root, or None if the tree is empty.
    _subtrees:
    The list of all subtrees of this tree.
    === Representation Invariants ===
    If the self._root is None, then self._subtrees is an empty list.
    This setting of attributes represnts an empty tree.

    Note: self._subtrees may be empty when self._root is not None.
    This setting of attributes represents a tree consisting of just
    one node.
    """

    _root: Optional[Any]
    _subtrees: List[Tree]

    def __init__(self, root: Optional[Any], subtrees: List[Tree]) -> None:
        """Initialize a new tree with the given root value and subtrees.
        If <root> is None, this tree is empty.
        Precondition: if <root> is None, then <subtree> is empty.
        """
        self._root = root
        self._subtrees = subtrees

    def is_empty(self) -> bool:
        """Return whether this tree is empty.
        >>> t1 = Tree(None, [])
        >>> t1.is_empty()
        True
        >>> t2 = Tree(3, [])
        >>> t2.is_empty()
        False
        """
        return self._root is None
        # EMPTY TREE: root is None
        # YOU CAN HAVE A ROOT BEING SMTH AND NO SUBTREES, The tree is composed of a single value

    ### ----------RECURSION ON TREES---------- ###
    # THE SIZE OF A NON EMPTY TREE :  The sum of the sizes of its subtrees plus 1 (root)
    # THE SIZE OF AN EMPTY TREE: 0

    def __len__(self) -> int:
        """Return the number of item contained in this tree.
        >>> t1 = Tree(None, [])
        >>> len(t1)
        0
        >>> t2 = Tree(3, [Tree(4, []), Tree(1, [])])
        >>> len(t2)
        3
        """
        if self.is_empty():  # tree is empty
            return 0
        elif self._subtrees == []:  # tree is a single item
            return 1
        else:  # Has at least one subtree
            size = 1  # We start at one because of the root
            for subtree in self._subtrees:
                size += len(subtree)  # could also write it as subtree.__len__()
            return size
    # IF WE AREN'T DOING ANYTHING DIFFERENT WITH THE ROOT, THEN THE SECOND CASE IS REDUNDANT
    # IT IS TECHNICALLY COVERED WHEN WE SET SIZE = 1
    def __str__(self) -> str:
        """Return a string representation of this tree!"""
        # if self.is_empty():
        #     return ""
        # else:
        #     # We use newlines (\n) to separate the different values.
        #     s = f'{self._root}\n'
        #     for subtree in self._subtrees:
        #         s += str(subtree)  # equivalent to subtree.__str__()
        #     return s
        # INSTEAD JUST CALL _str_indented
        return self._str_indented(0)  # the start depth is zero

    def _str_indented(self, depth: int = 0) -> str: # giving a default value to depth
        """Return an indented string representation of this tree.
        The indentation level is specified by the <depth> parameter.
        """
        if self.is_empty():
            return ''
        else:
            s = "- "* depth + str(self._root) + "\n"
            for subtree in self._subtrees:
            # Note that the 'depth' argument to the recursive call is modified.
                s += subtree._str_indented(depth + 1)
            return s


if __name__ == "__main__":
    print("hello")
    t1 = Tree(1, [])
    t2 = Tree(2, [])
    t3 = Tree(3, [])
    t4 = Tree(4, [t1, t2, t3])
    t5 = Tree(5, [])
    t6 = Tree(6, [t4, t5])
    print(t6)
```