This book is for mathematicians, scientists, and engineers who have learned the very basics of programming in Python, and who would like to become more capable programmers. In addition to covering higher level programming concepts such as objects, inheritance, and abstract data types, emphasis is placed on programming skills such as interpreting and debugging errors. If you find yourself baffled by the pages of error messages that Python emits, and would like to make sense of them, then this book is for you.

The book takes a mathematician's view of programming, introducing higher level programming abstractions by analogy with the abstract objects that make up higher mathematics. Examples and exercises are chosen from across mathematics, though the actual mathematical knowledge required to understand this book is limited to differentiating functions of one variable.

Dr David Ham is a Reader in Computational Mathematics at Imperial College London. He teaches the object-oriented programming course for undergraduate mathematicians on which this book is based, as well as an advanced course on the finite element method. His research focuses on high level abstractions and code generation for forward and inverse finite element simulation.

Object-oriented Programming in Python for Mathematicians

David A. Ham

Object-oriented Programming in Python for Mathematicians

Second Edition

David A. Ham

```
class TreeNode:
    """A basic tree implementation.

    A tree is simply a collection of connected TreeNodes.

    Parameters
    ----------
    value:
        An arbitra
    children:
        The TreeNo
    """

    def __init__(self, value, *children):
        self.value = value
        self.children = tuple(children)

    def __repr__(self):
        """Return the canonical string representation."""
        return f"{type(self).__name__}{(self.value,) + self.children}"

    def __str__(self):
        """Serialise the tree recursively as parent -> (children)."""
        childstring = ", ".join(map(str, self.children))
        return f"{self.value!s} -> ({childstring})"


def previsitor(tree, fn, fn_parent=None):
    """Traverse tree in preorder applying a function to every node.

    Parameters
    ----------
    tree: TreeNode
        The tree to be visited.
    fn: function(node, fn_parent)
        A function to be applied at each node. The function should take the
        node to be visited as its first argument, and the result of visiting
        its parent as the second.
    """
    fn_out = fn(tree, fn_parent)

    for child in tree.children:
        previsitor(child, fn, fn_out)


def postvisitor(tree, fn):
    """Traverse tree in postorder applying a function to every node.

    Parameters
    --
    tr

    fn
        A function to be applied at each node. The function should take the
        node to be visited as its first argument, and the results of visiting
        its children as any further arguments.
    """
    return fn(tree, *(postvisitor(c, fn) for c in tree.children))
```