Chapter 3
Hide contents

3.1 Introduction

3.1.1 Programming Languages

3.2 Functional Programming

3.2.1 Expressions
3.2.2 Definitions
3.2.3 Compound values
3.2.4 Symbolic Data
3.2.5 Turtle graphics

3.3 Exceptions

3.3.1 Exception Objects

3.4 Interpreters for Languages with Combination

3.4.1 A Scheme-Syntax Calculator
3.4.2 Expression Trees
3.4.3 Parsing Expressions
3.4.4 Calculator Evaluation

3.5 Interpreters for Languages with Abstraction

3.5.1 Structure
3.5.2 Environments
3.5.3 Data as Programs

Chapter 3: Interpreting Computer Programs

3.1 Introduction

Chapters 1 and 2 describe the close connection between two fundamental elements of programming: functions and data. We saw how functions can be manipulated as data using higher-order functions. We also saw how data can be endowed with behavior using message passing and an object system. We have also studied techniques for organizing large programs, such as functional abstraction, data abstraction, class inheritance, and generic functions. These core concepts constitute a strong foundation upon which to build modular, maintainable, and extensible programs.

This chapter focuses on the third fundamental element of programming: programs themselves. A Python program is just a collection of text. Only through the process of interpretation do we perform any meaningful computation based on that text. A programming language like Python is useful because we can define an *interpreter*, a program that carries out Python's evaluation and execution procedures. It is no exaggeration to regard this as the most fundamental idea in programming, that an interpreter, which determines the meaning of expressions in a programming language, is just another program.

To appreciate this point is to change our images of ourselves as programmers. We come to see ourselves as designers of languages, rather than only users of languages designed by others.

3.1.1 Programming Languages

Programming languages vary widely in their syntactic structures, features, and domain of application. Among general purpose programming languages, the constructs of function definition and function application are pervasive. On the other hand, powerful languages exist that do not include an object system, higher-order functions, assignment, or even control constructs such as `while` and `for` statements. As an example of a powerful language with a minimal set of features, we will introduce the Scheme programming language. The subset of Scheme introduced in this text does not allow mutable values at all.

In this chapter, we study the design of interpreters and the computational processes that they create when executing programs. The prospect of designing an interpreter for a general programming language may seem daunting. After all, interpreters are programs that can carry out any possible computation, depending on their input. However, many interpreters have an elegant common structure: two mutually recursive functions. The first evaluates expressions in environments; the second applies functions to arguments.

These functions are recursive in that they are defined in terms of each other: applying a function requires evaluating the expressions in its body, while evaluating an expression may involve applying one or more functions.

Continue: 3.2 Functional Programming