Lecture 01: Introduction

Vincenzo Ciancia

October 02, 2025

Section 1: What is a Programming Language?

A **programming language** is a formal language comprising a set of strings (instructions) that produce various kinds of machine output. Programming languages are used in computer programming to implement algorithms.

Key Characteristics of Programming Languages

- **Syntax**: The form or structure of the expressions, statements, and program units
- **Semantics**: The meaning of the expressions, statements, and program units
- **Type System**: The set of types and rules for how types are assigned to various constructs in the language
- **Runtime Model**: How the language executes on a computer, including memory management
- **Standard Library**: Common functionality provided out of the box

Programming Languages vs. Natural Languages

Programming languages differ from natural languages in several important ways:

1. **Precision**: Programming languages are designed to be precise and unambiguous
2. **Vocabulary**: Programming languages have a limited vocabulary defined by the language specification
3. **Grammar**: Programming languages have a strict, formal grammar with precise rules
4. **Evolution**: Programming languages evolve through explicit design decisions, not organic usage
5. **Purpose**: Programming languages are designed to instruct machines, not primarily for human communication

Section 2: The Importance of Programming Language Design

Why should we care about programming language design?

Programming Languages Shape How We Think

Programming languages are not just tools for instructing computers; they are frameworks for human thinking. Different languages emphasize different concepts and approaches:

- **Imperative languages** (C, Pascal) focus on step-by-step instructions
- **Functional languages** (Haskell, Lisp) emphasize expressions and function composition
- **Object-oriented languages** (Java, C++, Python) organize code around objects and their interactions
- **Logic languages** (Prolog) express programs as logical relations

Language Design Affects Software Quality

The design of a programming language can significantly impact:

- **Reliability**: How easy is it to write correct code?
- **Maintainability**: How easy is it to understand and modify existing code?
- **Performance**: How efficiently can the code be executed?
- **Security**: How easily can programmers avoid security vulnerabilities?
- **Developer Productivity**: How quickly can developers write and debug code?

The Evolution of Programming Languages

Programming languages have evolved dramatically over time, reflecting changes in hardware, software engineering practices, and problem domains:

- **1950s**: Assembly languages and early high-level languages (FORTRAN, LISP)
- **1960s**: ALGOL, COBOL, and structured programming concepts
- **1970s**: C, Pascal, and the rise of procedural programming
- **1980s**: C++, Ada, and the adoption of object-oriented programming
- **1990s**: Java, Python, Ruby, and the focus on portability and productivity
- **2000s**: C#, JavaScript frameworks, and web-centric languages
- **2010s**: Go, Rust, Swift, and the focus on safety and concurrency
- **2020s**: Continued evolution with AI assistance, type inference improvements, and more

Section 3: Modern Language Design Principles

What principles guide the design of modern programming languages?

Abstraction

Abstraction is the process of removing details to focus on the essential features of a concept or object.

Examples in programming languages: - Functions abstract away implementation details - Classes abstract data and behavior - Interfaces abstract expected behaviors - Modules abstract related functionality

Expressiveness

Expressiveness refers to how easily and concisely a language can express computational ideas.

Factors that contribute to expressiveness: - Rich set of operators and built-in functions - Support for higher-order functions - Pattern matching - Concise syntax for common operations

Safety

Safety features help prevent programmers from making mistakes or make it easier to find and fix errors.

Safety mechanisms in modern languages: - Static type checking - Bounds checking - Memory safety guarantees - Exception handling systems - Null safety features

Performance

Performance considerations affect how efficiently a language can be implemented and executed.

Performance factors: - Compilation vs. interpretation - Memory management approach - Static vs. dynamic typing - Optimization opportunities - Support for concurrency and parallelism

Consistency

Consistency in language design makes languages easier to learn and use correctly.

Consistency principles: - Similar concepts should have similar syntax - Minimal special cases - Orthogonal features (features that can be used in any combination) - Principle of least surprise (intuitive behavior)

Section 4: Using Python to Explore Programming Language Concepts

Why use Python for studying programming language design?

Python's Suitability for Language Implementation

Python is well-suited for implementing language interpreters and exploring language concepts:

- **Readability**: Python's clean syntax makes interpreter code easier to understand
- **High-level constructs**: Python provides lists, dictionaries, and other structures useful for language implementation
- **Dynamic typing**: Simplifies working with diverse language constructs (NOT REALLY! left here for discussion)
- **Rich standard library**: Includes parsing tools, regular expressions, and other useful utilities
- **Interactive development**: Makes experimenting with language features easier

Python 3.10+ Features Relevant to Language Design

Recent Python versions have introduced features that make it particularly interesting for PL experiments:

- **Type hints**: Allows for static type checking while maintaining dynamic execution
- **Pattern matching**: Provides elegant structural decomposition similar to functional languages
- **Dataclasses**: Simplifies creating data-carrying classes with minimal boilerplate
- **Functional programming tools**: Map, filter, reduce, lambdas, and comprehensions
- **AST module**: Allows inspection and manipulation of Python's abstract syntax tree

Section 5: Implementing Language Features in Python

Let's explore how we can implement core language components in Python.

Representing Syntax: Abstract Syntax Trees (ASTs)

An abstract syntax tree (AST) is a tree representation of the abstract syntactic structure of source code. Here's a simple example of representing expressions:

```
from dataclasses import dataclass
from typing import Union, List


# Define the node types
@dataclass
class Number:
    value: float


@dataclass
class Variable:
    name: str
```

```python
@dataclass
class BinaryOp:
    left: 'Expr'
    operator: str
    right: 'Expr'

# Define the expression type
Expr = Union[Number, Variable, BinaryOp]

# Example: 2 + (x * 3)
expr = BinaryOp(
    left=Number(2.0),
    operator='+',
    right=BinaryOp(
        left=Variable('x'),
        operator='*',
        right=Number(3.0)
    )
)
```

Pattern Matching for AST Processing

Python 3.10's pattern matching provides a more elegant way to implement evaluators:

```
match expr:
    case Number(value):
        return value
    case Variable(name):
        if name not in environment:
            raise NameError(f"Variable '{name}' not defined")
        return environment[name]
    case BinaryOp(left, operator, right):
        left_val = evaluate_with_match(left, environment)
        right_val = evaluate_with_match(right, environment)

        match operator:
            case '+':
                return left_val + right_val
            case '-':
                return left_val - right_val
```

Section 6: Course Structure

This course will introduce you to programming language design concepts through hands-on implementation in Python.

Course Topics

Throughout this course, we will build a simple programming language interpreter while covering key topics:

1. **Language Syntax and Semantics**
    - Parsing and lexical analysis
    - Abstract syntax trees
2. **Language Features**
    - Environment (how to assign names to things?)
    - State (how to represent changing values?)
    - Control flow (if, while, for, etc.)
    - Static scoping (how do variable names related to nested environments)

Projects and Exercises

The course will include:

- Regular programming exercises to reinforce concepts
- Progressive development of a language interpreter
- Exploration of existing language implementations
- Analysis of language design trade-offs

Knowledge Prerequisites

To get the most out of this course, you should have:

- Basic Python programming experience
- Understanding of fundamental programming concepts (variables, functions, control flow)
- Familiarity with basic data structures (lists, dictionaries, trees)
- Interest in how programming languages work “under the hood”

No prior experience with compiler or interpreter development is required.

Additional Resources

Online Resources

- Python Documentation
- Python Type Hints
- Pattern Matching in Python 3.10
- The AST Module
- Building a Simple Interpreter