PiccodeScript

⟵ Language Reference Guide ⟶

Author: Gama Sibusiso Vincent

Preface

This book began with a simple question: what if a language could be expressive, predictable, and compact—with just the essentials?

PiccodeScript was created with that spirit in mind. It strips away the noise of modern development and gives you a sharp, clear tool to think with. No distractions. No bloat. Just clean, functional code that does what you ask.

This guide is not just about teaching syntax—it’s about introducing a mindset. One where you solve problems with precision. Where fewer lines say more. Where functions, not features, drive the code.

Whether you’re a curious beginner or a seasoned coder looking for a different pace, this guide is designed to get you going fast and thinking clearly.

Welcome to PiccodeScript.

Table of Contents

Introduction

PiccodeScript is a lightweight, expressive scripting language designed for clarity, composability, and fun. It combines the flexibility of Python with the structure of C and the conciseness of modern functional languages. PiccodeScript favors immutability, avoids hidden side effects, and encourages writing readable, maintainable code.

PiccodeScript is:

- **Interpreted** — You can run scripts directly without compilation.
- **Dynamically typed** — Variables don't need type declarations, and types are resolved at runtime.
- **Functional-first** — Functions are first-class citizens, and expressions dominate over statements.

This book is a fast-paced, practical introduction to PiccodeScript. You'll learn how to work with data structures, write functions, structure logic, and build small utilities. With clear syntax and powerful features, PiccodeScript is ideal for rapid scripting, data manipulation, and learning programming fundamentals.

Whether you're a beginner or a seasoned developer, this guide will show you how to get productive quickly.

Hello, World

```
1 | import std.io
2 |
3 | IO.println("Hello, world!")
4 |
5 |
```

Every PiccodeScript program begins with importing the modules it needs. Here, std.io provides basic input/output functions. The IO.println call sends text to standard output.

The language has a small set of core keywords: import, let, function, module, if, else, when, and is. There are no hidden rules or secret behaviors—what you see is what you run.

This simplicity makes it easy to understand and reason about code. Even complex programs remain approachable and predictable.

Variables & Expressions

```
1 |
2 | let n = 5 * 2
3 | let song = "Feel good Inc"
4 |
```

Variables are declared using **let**, and cannot be reassigned. This eliminates whole classes of bugs related to unexpected state changes.

Expressions are used everywhere: assignment, conditionals, function bodies. You can build logic with combinations of mathematical operators, string operations, and pipelines. Everything evaluates to a value.

There are no type annotations—types are inferred at runtime. This allows for fast iteration and less boilerplate.

Arrays

```
1 |
2 | let numbers = [1, 2, 3]
3 | let first = numbers[0]
4 | let joined = first:numbers
```

Arrays are zero-indexed collections. You access items using a `arr[<index>]` syntax.

The `:` operator allows concatenation, either prepending or appending values. Arrays can be any length and store any types, though consistency is often good practice.

Use arrays for list processing, pipelines, and batching data. They are fundamental tools in *PiccodeScript*.

Tuples

```
1 |
2 | let triplet = (1, 2, 3)
3 | let first = triplet[0]
4 |
```

Tuples are immutable, fixed-size groupings. They are ideal when you know exactly how many values you're dealing with and what each represents.

Access is positional, using the same dot-number syntax as arrays. While arrays are dynamic and flexible, tuples are structured and precise.

Tuples can be destructured in when blocks, passed to functions, and used as return values to cleanly package related data.

Conditionals

```
1 |
2 | if 1 > 1 { "Too big" } else { "All good" }
3 |
```

The `if` keyword introduces a conditional expression. Both branches must be present and enclosed in braces. Each branch returns a value.

PiccodeScript treats if as an expression, not a statement. That means you can assign or return its result directly:

```
1 |
2 | let message = if n > 0 { "Positive" }
3 | else { "Non-positive" }
4 |
```

This promotes a clear, declarative coding style where logic flows directly from data.

Pattern Matching

```
1 |
2 | when numbers {
3 |  is (a, b, c) → a + " " + b + " " + c
4 |  else → "Not a match"
5 | }
6 |
```

Use `when` to match and destructure values. It's powerful, concise, and expressive. Patterns are matched top-down, and the first match executes.

The is keyword allows destructuring into named bindings. If no pattern matches, the else clause acts as a fallback. The → separates the pattern from its result.

Use when to handle complex data and encode decision logic that remains readable.

Functions

```
1 |
2 | function add(x=1, y=1) = x + y
3 |
4 | let result = add(y=2)
5 |
```

Functions are declared using `function`, and support default arguments. Functions are also expressions, which means you can define them inline, assign them to variables, or return them from other functions.

Named arguments increase clarity, especially when using defaults or when order doesn't matter. The function body is a single expression, which reinforces the functional nature of the language.

This approach leads to cleaner, more predictable functions.

Pipelines

```
1 |
2 | let answer = 10 ▷ add(10) ▷ add(10)
3 |
4 |
```

The `▷` (**|>**) operator pipes a value into the function on the right. It reads left to right, forming a clean sequence of transformations.

Pipelines are perfect for chaining function calls and building readable flows without nesting or temporary variables. This encourages composability and reduces clutter.

Think of pipelines as the glue between functions. They let you express processes as a series of steps.

Modules

```
1  |
2  | import std.io
3  |
4  | module Foo {
5  |   function bar() = IO.println("Foo.Bar")
6  | }
7  |
8  | module Foo {
9  |   function baz() = IO.println("Foo.Baz")
10 | }
11 |
12 | Foo.bar()
13 | Foo.baz()
14 |
15 |
```

Modules group related functions and variables. Repeated declarations of the same module name merge their contents, allowing incremental definition.

You can also define modules inside other modules:

```
1  |
2  | import std.io
3  |
5  | module Foo {
6  |  module Foo2 {
7  |    function bar2() = IO.println("Foo.Foo2.bar2")
8  |  }
9  | }
10 |
11 | IO.println(Foo.Foo2)
12 | Foo.Foo2.bar2()
```

This enables deep structuring of APIs and code. Modules help separate logic and encapsulate responsibilities cleanly.

Annotations

Command Line usage

PiccodeScript compiler is a command-line interface tool called **picoc**. It's used to run scripts, build projects, manage libraries, and launch the interactive REPL.

Help and Version

```
$ picoc –h
$ picoc --version
```

Running Code

```
$ picoc run <file> # Run a specific script file
$ picoc run # Run the current project (main script)
```

Creating a New Project

```
$ picoc new # Create a new project in the current directory
```

Building

```
$ picoc build <file> # Compile a specific file
$ picoc build     # Build the whole project
```

Working with Libraries

```
$ picoc lib --new <file> # Create a new .jar library from a file
$ picoc lib --ls <file>  # List contents of a .jar library
```

REPL Mode

```
$ picoc run --repl   # Launch interactive REPL session
```

Use the CLI to experiment, automate builds, or script workflows. It’s your main interface to executing and packaging PiccodeScript project

The End

You’ve reached the end of this mini guide, but you’re just getting started with PiccodeScript.

What you’ve seen here is a language built on clarity and constraint. With only a few keywords, consistent syntax, and functional foundations, PiccodeScript gives you powerful tools without unnecessary complexity.

Use it to prototype ideas, automate tasks, write clean utilities, or explore functional programming. Build modular systems. Compose functions. Match patterns. Write code that feels solid, readable, and easy to reason about.

If something feels missing, that’s intentional. PiccodeScript is minimal on purpose. It’s not here to overwhelm—it’s here to empower.

Keep building. Keep refining. And if you’re inspired, contribute. The language will grow with its users.

Until then, write it in PiccodeScript.