Document Generated: 11/15/2026

Learning Style: Virtual Classroom

Technology:

Difficulty: Intermediate

Course Duration: 5 Days

Hands-on Rust Programming for Python Developers (TTRS2100)



About This Course:

Rust is a modern programming language designed for performance and safety, particularly in concurrent systems, offering a unique combination of efficiency and reliability. Learning Rust equips you with the ability to write high-performance, bug-free code, greatly enhancing your programming toolkit.

Geared for Python experienced developers, this five-day Rust programming course is designed to help you transition from Python to Rust with confidence. Led by an expert trainer, the course combines engaging lectures with hands-on labs, ensuring you grasp Rust's unique features and gain practical experience. You'll explore Rust's syntax, control flow, and module imports, and move on to advanced topics like memory management, concurrency, and pattern matching. By leveraging Cargo, you'll streamline project management and integrate popular crates like Serde for serialization, Tokio for asynchronous programming, and SQLx for database interactions, enabling you to build robust, scalable applications.

The workshop-style format dedicates half of the class time to hands-on labs, applying what you've learned in real-world scenarios. You'll gain expertise in writing concurrent programs using Rust's powerful tools, such as threads, Mutex, RwLock, and async/await, ensuring your applications handle multiple tasks efficiently. You'll also master pattern matching, creating readable and maintainable code. By the end of the course, you'll be confident in writing comprehensive tests and documentation, making you a highly effective programmer ready to tackle complex Rust projects and advance your career.

Course Objectives:

- Master Rust Syntax and Semantics: Get comfortable writing and understanding Rust code, including how to use control flow, functions, and module imports.
- Implement Memory Management: Learn to manage memory effectively in Rust by mastering concepts like ownership, borrowing, and lifetimes to ensure your code is both safe and efficient.
- Leverage Cargo and Crates: Become adept at using Cargo to manage Rust projects and dependencies, and explore how to integrate popular crates like Serde, Tokio, and SQLx to add powerful features to your applications.
- Utilize Rust's Concurrency Model: Discover how to write efficient and concurrent programs using Rust's concurrency tools, such as threads, Mutex, RwLock, and async/await.
- Employ Pattern Matching and Generics: Understand and apply pattern matching with enums and functions, and use generics to create flexible and reusable code.
- Create and Test Rust Applications: Develop complete Rust applications, including writing tests with Rust's testing framework and generating clear, comprehensive documentation using Rustdoc.

Audience:

- This course is designed for Python developers looking to expand their programming skills with Rust. It is ideal for software engineers, system

programmers, and developers interested in performance-critical and concurrent applications. Whether you are transitioning to a new role or enhancing your current one, this course provides the knowledge and skills to excel in Rust programming.

Prerequisites:

- Proficiency in Python Programming: A strong understanding of Python syntax, functions, and modules.
- Basic Knowledge of Programming Concepts: Familiarity with variables, expressions, control flow (loops and conditionals), and basic data structures.
- Experience with Version Control Systems: Basic experience using Git for version control, including committing, branching, and merging code.
- Understanding of Software Development Practices: Familiarity with project management tools and practices such as virtual environments, dependency management, and code testing.

Course Outline:

Getting Started with Rust

- What is Rust?
- Philosophy and Goals
- History and Motivation
- Differences between Rust and Python
- Rust Community and Ecosystem
- Exploring the Rust Playground

Install Rust (Optional)

- Installation Script
- Using macOS Homebrew
- Platform-Specific Installers

Rust Editors

- Setting up VSCode with Rust Extensions
- Using Rust Rover IDE
- Debugging Rust Code in VSCode
- Integrating GitHub Copilot for Rust

Hello World

- Creating a New Rust Project
- Writing the Main Function
- Printing to the Console
- Adding Comments to Code

Cargo

- Understanding What Cargo Is
- Comparing Cargo to Pip and Conda
- Rust Crates vs Python Packages
- Using Run, Build, and Release Commands
- Installing and Managing Third-Party Crates

Popular Cargo Crates

- Overview of Serde for Serialization
- Introduction to Tokio for Asynchronous Programming
- Using Reqwest for HTTP Requests
- Working with SQLx for Database Interactions
- Error Handling with Anyhow

Rust and Python Differences

- Static Typing in Rust vs Dynamic Typing in Python
- Memory Management Techniques
- Error Handling Approaches
- Control Flow: Sequence, Selection, and Iteration
- Structs vs Classes, and Traits vs Protocols

Scalar Types and Data

- Comparing Rust Types with Python Types
- Defining Constants in Rust
- Using Immutable Variables
- Utilizing Mutable Variables

Code Logic

- Writing If Statements
- Using Loops with Break
- Implementing While Loops

Functions

- Defining and Calling Functions
- Specifying Parameter and Return Types
- Creating and Using Closure Functions

Modules

- Importing Modules from the Standard Library
- Importing Modules from Third-Party Crates
- Defining Custom Modules
- Importing and Using Custom Modules

Built-In Macros

- Using print!, println!, and format! Macros
- Working with vec!, include_str!, and include_bytes!
- Employing cfg!, env!, and panic! Macros

Memory Management

- Understanding Ownership & Borrowing Concepts
- Working with References
- Differentiating Immutable vs Mutable References
- Managing Lifetimes in Rust
- Heap Allocation with Box and Rc

Strings

- Using String Slices and String Objects
- Converting Between Slices and Strings
- Parsing Numbers from Strings
- Trimming Strings and Printing with Interpolation

Tuples

- Understanding What a Tuple Is
- Using Heterogeneous Elements
- Accessing and Destructuring Tuple Elements
- Immutability of Tuples

Enums

- Defining and Using Enums
- Exploring Enum Variants and Methods

- Pattern Matching with Enums
- Working with Result and Option Enums

Structs

- Creating and Using Struct Instances
- Using Field Initialization Shorthand
- Implementing Methods and Associated Functions
- Employing the Constructor Pattern

Vectors

- Creating and Managing Vectors
- Adding and Removing Elements
- Accessing and Iterating Over Elements
- Slicing, Checking Length, and Capacity

Collections and Iterators

- Using Vectors, Arrays, and Slices
- Working with HashMaps and Hash Sets
- Implementing Iteration with Iterators

Traits

- Defining and Implementing Traits
- Using Default Trait Implementations
- Passing Traits as Parameters and Return Types

Generics

- Defining and Implementing Generics in Rust
- Using Generic Bounds and Multiple Types

- Writing Where Clauses for Generics

Pattern Matching

- Understanding Pattern Matching Concepts
- Using Match, If Let, and While Let Statements
- Destructuring Structs and Tuples
- Pattern Matching with Enums and Functions

Concurrent Programming

- Introduction to Concurrent Programming Concepts
- Using Multiple Threads in Rust
- Working with Mutex, RwLock, and Arc
- Message Passing with Channels
- Futures and Async/Await for Concurrency

Unsafe Rust

- Understanding the Need for Unsafe Rust
- Working with Raw Pointers and Dereferencing
- Calling Unsafe Functions and Creating Safe Abstractions
- Using Unsafe Traits and Blocks

Macros and Metaprogramming

- Defining and Using Macros with macro_rules!
- Pattern Matching in Macros
- Defining and Expanding Custom Macros
- Tests
- Writing and Organizing Test Functions

- Using Test Attributes and Coverage Tools
- Assertions with assert!, assert_eq!, and assert_ne!

Documentation with Rustdoc

- Generating Documentation with Rustdoc
- Adding Triple-Slash Comments and #[doc] Attributes
- Linking and Cross-Referencing Documentation

Python Extension written in Rust (overview)

- Creating and Running Python Extensions in Rust
- Compiling and Debugging Rust Extensions
- Running Parallel Code Outside of the GIL
- Managing GIL and Rust Lifetimes
- Creating Python Classes in Rust

Wrapping It Up

- Recap of Key Rust Concepts
- Next Steps for Further Learning and Practice