Programming for QA

Python, JavaScript, and TypeScript from Zero to Fluent

modernQAcourse.com

THE MODERN QA ENGINEER SKILLSET

Programming for QA: Python, JavaScript, and TypeScript from Zero to Fluent

Three Languages, One Mission: Better Testing

Yuri Syuganov

Modern QA Course

https://modernQAcourse.com

Programming for QA: Python, JavaScript, and TypeScript from Zero to Fluent

Three Languages, One Mission: Better Testing

Book 12 of 26 in the series *The Modern QA Engineer Skillset*

First Edition · Version 1.0.0 · 2026-07-11

Set in Charter, IBM Plex Sans, and IBM Plex Mono. Typeset with Pandoc and XeLaTeX.

Modern QA Course · https://modernQAcourse.com · support@modernQAcourse.com

Contents

Introduction

A QA engineer who cannot code is limited to tools that other people build. A QA engineer who can code builds the tools, automates the checks, parses the logs, and scripts the pipelines. Programming fluency is the single biggest force multiplier in a testing career.

This book teaches you programming from absolute zero. You do not need any prior coding experience. By the final chapter, you will be writing test scripts, building automation frameworks, parsing logs, making API calls, and scripting CI/CD pipelines – confidently, in Python, JavaScript, and TypeScript.

Every concept in this book is taught three times: once in Python, once in JavaScript, and once in TypeScript. This triple-track approach is deliberate. In the real world, QA engineers work across all three. Python dominates backend and API testing. JavaScript powers browser automation and quick scripting. TypeScript adds type safety to JavaScript and is the default for modern frameworks like Playwright. Fluency in all three makes you versatile, hireable, and effective.

All examples are drawn from real QA work. You will not sort abstract number arrays or build toy calculators. Instead, you will parse test results, validate API responses, process log files, build page objects, and automate test environments.

Who This Book Is For

This book is for:

- **Manual testers** who want to transition into test automation
- **Junior QA engineers** who need to build programming confidence

- **Career changers** entering QA from non-technical backgrounds
- **QA leads** who want a structured resource for training their teams
- **Anyone** who has tried to learn programming before and found the examples irrelevant to their actual work

You do not need:

- A computer science degree
- Prior programming experience
- Advanced math skills

You do need:

- A computer with Python 3.10+ and Node.js 24+ (active LTS) installed
- TypeScript installed globally (`npm install -g typescript tsx`)
- A text editor or IDE (Visual Studio Code is recommended)
- Willingness to type out every example yourself – do not copy and paste
- About 200 hours of dedicated study time

How to Use This Book

Read actively. Type every code example yourself. The physical act of typing builds muscle memory and forces you to notice syntax differences between the three languages.

Do every exercise. Each chapter ends with three exercises graded Easy, Medium, and Hard. Start with Easy. If you get stuck on Hard, move on and come back later.

Pick a primary language. Although this book teaches all three, pick one as your primary and focus on it first. A good rule: if your team uses pytest, start with Python. If your team uses Playwright or Cypress, start with TypeScript. Read the other tracks for comparison.

Use the quizzes. Each chapter ends with a five-question quiz. If you cannot answer most questions correctly, re-read the chapter before moving on.

The book is structured in seven parts:

Part	Topic	Chapters
1	Language Fundamentals	Chapters 1-5
2	Data Structures	Chapters 6-9
3	Object-Oriented Programming	Chapters 10-13
4	Functional Programming	Chapters 14-16
5	Async Programming	Chapters 17-19
6	Practical Skills	Chapters 20-23
7	Bash Scripting	Chapters 24-25

Conventions Used in This Book

Throughout the book, you will encounter these callout boxes:

> **Why This Matters for QA**: These sections explain how a programming concept connects directly to real testing work. Never skip these.

> **Pro Tip**: Practical advice from experienced QA engineers.

> **Common Mistake**: Warnings about errors that beginners make frequently.

Code examples always appear in three blocks. Python first, then JavaScript, then TypeScript:

```python
# Python
test_name = "Login Test"
print(f"Running: {test_name}")
```

```javascript
// JavaScript
const testName = "Login Test";
console.log(`Running: ${testName}`);
```

```
// TypeScript
const testName: string = "Login Test";
console.log(`Running: ${testName}`);
```

When a concept is identical in JavaScript and TypeScript, the TypeScript block will still appear but will highlight what type annotations or TypeScript-specific features add to the picture.

PART 1

LANGUAGE FUNDAMENTALS

CHAPTER 1

Your First Program

Every programmer starts here. You will write your first program, understand what happens when it runs, and learn to use the tools that execute your code.

1.1 Setting Up

Install the following before continuing:

- **Python 3.10+** from https://python.org
- **Node.js 24+** (active LTS) from https://nodejs.org
- **TypeScript** via `npm install -g typescript tsx`
- **Visual Studio Code** from https://code.visualstudio.com with the Python, ESLint, and Prettier extensions

Verify all three:

```
python --version    # Python 3.10.x or later
node --version      # v24.x.x or later
tsc --version       # Version 5.x.x or later
```

1.2 Hello World

Create a file, run it, see output. That is the entire cycle.

```
# hello.py
print("Hello, QA world!")
```

```
// hello.js
console.log("Hello, QA world!");
```

```
// hello.ts
const greeting: string = "Hello, QA world!";
console.log(greeting);
```

Run each one:

```
python hello.py       # Python
node hello.js         # JavaScript
npx tsx hello.ts      # TypeScript (tsx runs .ts files directly)
node hello.ts         # Also works on Node.js 24+ (type-stripping)
```

As of July 2026, Node.js 24 is the active LTS release, and it runs TypeScript files directly through stable native type-stripping: `node hello.ts` strips the type annotations and executes the underlying JavaScript. The catch: type-stripping does not type-check. Node will happily run code with type errors, so keep `tsc --noEmit` in CI to catch them. (Node.js 26 is the Current release; starting with Node 27 the project ships one major version per year.)

1.3 The REPL

A REPL (Read-Eval-Print Loop) lets you type code interactively and see results immediately. Use it to experiment.

```
# Python REPL
python
>>> 2 + 2
4
>>> "test" + "_login"
'test_login'
>>> exit()
```

```
# Node.js REPL (works for both JS and quick TS experiments)
node
> 2 + 2
4
> "test" + "_login"
'test_login'
> .exit
```

TypeScript does not ship a built-in REPL, but `npx tsx` can run `.ts` files instantly. For interactive exploration, use the Node REPL and add types when you move code into a `.ts` file.

1.4 Comments

Comments are notes for humans. The interpreter ignores them.

```
# This is a Python comment
# Every line needs its own # character
```

```
// This is a JavaScript comment
/* This is a multi-line
   JavaScript comment */
```

```
// TypeScript uses the same comment syntax as JavaScript
/* Multi-line comments
   work identically */
```

1.5 Your First QA Script

A script that prints a test result summary. This is closer to real work than "hello world."

```python
# test_summary.py
test_name = "Login API"
status_code = 200
passed = status_code == 200

print(f"Test: {test_name}")
print(f"Status: {status_code}")
print(f"Result: {'PASS' if passed else 'FAIL'}")
```

```javascript
// test_summary.js
const testName = "Login API";
const statusCode = 200;
const passed = statusCode === 200;

console.log(`Test: ${testName}`);
console.log(`Status: ${statusCode}`);
console.log(`Result: ${passed ? "PASS" : "FAIL"}`);
```

```typescript
// test_summary.ts
const testName: string = "Login API";
const statusCode: number = 200;
const passed: boolean = statusCode === 200;

console.log(`Test: ${testName}`);
console.log(`Status: ${statusCode}`);
console.log(`Result: ${passed ? "PASS" : "FAIL"}`);
```

> **Why This Matters for QA**: Every test automation framework produces output like this. Understanding how to construct strings, evaluate conditions, and print results is the foundation of every test report, every CI log, and every Slack notification your pipeline sends.

1.6 Errors Are Your Friend

When you make a mistake, the interpreter tells you what went wrong. Read the message carefully.

```
print("Hello)
# SyntaxError: unterminated string literal

print(hello)
# NameError: name 'hello' is not defined
```

```
console.log("Hello)
// SyntaxError: Invalid or unexpected token

console.log(hello)
// ReferenceError: hello is not defined
```

```
// TypeScript catches errors BEFORE you run the code
const msg: string = 42;
// Type 'number' is not assignable to type 'string'.
```

> **Pro Tip**: TypeScript's compile-time errors are one of its biggest advantages for QA. Bugs caught before execution never reach production.

Exercises: Chapter 1

Easy: Write a script in all three languages that prints your name, your role ("QA Engineer"), and today's date as three separate lines.

Medium: Write a script that stores a test case name, an expected HTTP status, and an actual HTTP status in variables. Print whether the test passed or failed by comparing expected to actual.

Hard: Write a script that prints a mini test report for three test cases. Each test case has a name and a pass/fail status. The final line prints the total number of passes and failures.

Quiz: Chapter 1

1. What command runs a TypeScript file named `test_login.ts` without compiling it first?
2. What is the difference between `print()`, `console.log()`, and how does TypeScript's output differ from JavaScript's?
3. How do you start the Python REPL? The Node.js REPL?
4. What does a `SyntaxError` tell you?
5. Name one advantage TypeScript has over JavaScript for catching bugs.

Answers:

1. `npx tsx test_login.ts` (on Node.js 24+, `node test_login.ts` also works via native type-stripping)
2. `print()` is Python's output function. `console.log()` is used by both JavaScript and TypeScript. TypeScript's runtime output is identical to JavaScript – the types are erased at runtime.
3. Python: type `python` or `python3`. Node: type `node`.
4. That the code structure is invalid – something is misspelled, unmatched, or missing (like a closing quote or parenthesis).
5. TypeScript catches type errors at compile time, before the code runs. JavaScript only catches them at runtime.

Career Translation

Resume phrasing

- Established a multi-language development environment (Python 3.10+, Node.js 24+, TypeScript 5+) and built foundational test scripts, demonstrating cross-language fluency from day one
- Wrote and executed first automated test scripts across three languages, validating environment setup and establishing the development workflow used throughout the automation project
- Set up REPL-based debugging workflows for rapid prototyping and troubleshooting of test logic in Python and Node.js, reducing iteration time during test development

Cover letter framing

Every automation project starts with a working environment and a first script that proves it runs. I have set up development environments from scratch – Python, Node.js, TypeScript, VS Code, linters, formatters – and I understand the full chain from writing a script to executing it and reading the output. This foundational competence means I can onboard quickly in any codebase and debug toolchain issues that block less technical team members.

Interview framing

"I approach new projects by first verifying the development environment end to end – can I write a script, run it, and see correct output in all required languages? I optimize for reproducibility: if I set up the environment, I document every step so the next person can do it in 15 minutes instead of a day. Trade-offs include the time spent on environment setup versus jumping straight into test writing, but a broken environment wastes far more time than a careful setup."

What not to say

- "I have never set up a development environment" – this signals inability to work independently
- "I only know one language" – multi-language fluency is expected for modern QA roles
- "Someone else set up my environment" – you should be able to install Python, Node, and TypeScript yourself
- "I do not use a REPL" – the REPL is essential for rapid prototyping and debugging

Interview Depth Check

Question 1

Prompt: You join a new team and need to set up your local environment for a test automation project that uses Python and TypeScript. Walk me through your process.

What a strong answer should cover:

- Install Python, Node.js, and TypeScript at the correct versions
- Clone the repo and follow setup documentation (or create it if it does not exist)
- Run existing tests to verify the environment works end to end
- Set up VS Code with appropriate extensions and linting

Example answer:

- I start by checking the project's README or setup guide for required versions of Python, Node.js, and TypeScript
- I install the correct versions (using pyenv for Python, nvm for Node) and verify with `python --version`, `node --version`, `tsc --version`
- I install project dependencies: `pip install -r requirements.txt` and `npm install`
- I run the existing test suite to confirm everything passes on my machine before making any changes
- If setup documentation is missing or outdated, I create it as my first contribution – this helps the next person and proves I understand the setup

Question 2

Prompt: What is the difference between running a TypeScript file with `npx tsx` versus compiling it with `tsc` first? When does it matter?

What a strong answer should cover:

- `npx tsx` runs TypeScript directly without a separate compile step (JIT)
- `tsc` compiles to JavaScript first, which can be run with `node`
- For development and testing, `tsx` is faster and more convenient
- For production and CI, `tsc` catches all type errors upfront and produces distributable JavaScript
- Node.js 24+ (active LTS) also runs `.ts` files directly via native type-stripping – same caveat: it does not type-check

Example answer:

- `npx tsx hello.ts` compiles and runs in one step, which is ideal during development for fast iteration
- Since Node.js 24 (the active LTS as of July 2026), `node hello.ts` also runs directly thanks to stable native type-stripping – but like `tsx`, it does not type-check
- `tsc hello.ts` produces a JavaScript file that is then run with `node hello.js` – two steps, but the compile step catches all type errors across the entire project
- In my workflow, I use `tsx` for running individual tests and quick experiments, and `tsc --noEmit` in CI to verify the entire codebase compiles without errors
- The distinction matters because `tsx` only checks the file being run, while `tsc` checks the entire project, so CI should always run the full compile check

Question 3

Prompt: A colleague reports that their Python test script works but the same logic fails in TypeScript. How do you help them debug this?

What a strong answer should cover:

- Compare syntax differences between the two languages (print vs console.log, indentation vs braces)
- Check for type-specific issues (TypeScript's strict null checks, type narrowing)
- Use the REPL in both languages to isolate the failing expression
- Verify both environments are using the expected language versions

Example answer:

- First, I check the error message: a TypeScript compile error (type mismatch) is very different from a runtime error (wrong logic)
- I compare the two implementations side by side looking for language-specific differences: Python's truthiness rules differ from JavaScript's (empty list is falsy in Python but truthy in JS)

- I use the Node REPL and Python REPL to test the specific expression that produces different results
- I verify the TypeScript version and tsconfig settings – strict mode may catch issues that non-strict mode would ignore

About This Sample

You have just read **Chapter 1 of Programming for QA: Python, JavaScript, and TypeScript from Zero to Fluent** — the free sample. The complete book continues with every remaining chapter, the interview Q&A, and all of the code.

Get the full book

Sample readers get a discount at
https://leanpub.com/modernqa-12-programming-for-qa/c/SAMPLE10

Read online first

Chapter one of **every** book in the series is free to read at
https://modernqacourse.com/library

About the Author

Yuri Syuganov is a software quality engineer with more than twenty years of hands-on testing experience, spanning the arc from manual test design through Selenium-era automation to today's AI-augmented, agent-driven frameworks. He is the author of The Modern QA Engineer Skillset, a 26-book series built on a blunt analysis of what companies actually expect from QA engineers, and the creator of modernQAcourse.com, where the course behind the series is free and readers can bring real testing problems to experienced practitioners. He revises the series as the industry changes — because it does.

Also in This Series

The Modern QA Engineer Skillset — a 26-book course in modern software quality assurance.

AI-Augmented QA (Books 1–10)

- **Book 1:** Agent Skills for Browser Automation: The Complete Guide
- **Book 2:** AI-Augmented Test Design: From Prompts to Test Suites
- **Book 3:** Agentic Testing Architectures: Building Autonomous QA Systems
- **Book 4:** API and Contract Testing with AI: Schema to Shield
- **Book 5:** Performance and Chaos Engineering: Breaking Things on Purpose
- **Book 6:** Observability-Driven Testing: Testing in Production
- **Book 7:** Security Testing for AI Applications: The New Attack Surface
- **Book 8:** Infrastructure as Code Testing: Terraform to Kubernetes
- **Book 9:** Mobile and Cross-Platform Testing: 24,000 Devices, One Strategy
- **Book 10:** Visual and Accessibility Testing: Pixels and People

Technical Foundations (Books 11–19)

- **Book 11:** Manual Testing Fundamentals: The Bedrock
- **Book 12:** Programming for QA: Python, JavaScript, and TypeScript from Zero to Fluent
- **Book 13:** Browser Test Automation: Playwright, Patterns, and the Death of Fragile Tests

- **Book 14:** API Testing Fundamentals: Postman to Pytest
- **Book 15:** SQL and Database Testing: Query Your Way to Quality
- **Book 16:** CI/CD Pipelines: From Commit to Confidence
- **Book 17:** Git and Version Control: Beyond Push and Pull
- **Book 18:** Test Management Tools: Making Quality Visible
- **Book 19:** Linux and Command Line: Your Production Survival Kit

Professional Skills (Books 20–26)

- **Book 20:** Agile and Scrum for QA: Quality in Every Sprint
- **Book 21:** Communication and Stakeholder Management for QA: Influence Without Authority
- **Book 22:** Test Strategy and Quality Metrics: Measuring What Matters
- **Book 23:** QA Leadership and Mentoring: Multiplying Your Impact
- **Book 24:** Technical Writing for QA: Documents That Drive Action
- **Book 25:** Interview Preparation and Career Strategy: Land the Job
- **Book 26:** Testing Like a Senior: Patterns & Anti-Patterns Across Software QA Career Levels

Visit https://modernQAcourse.com for the full course catalog, updates, and companion resources.

Stuck on a Real Problem?

The full course behind this series is free to read at https://modernQAcourse.com — and it stays free.

When a chapter meets the real world and something breaks, post it on the **Help Board** at https://modernQAcourse.com. Asking is free, and experienced practitioners answer right in the thread. Need hands-on help? Book a 1:1 screen-share session with a QA expert.