Azhar Language: A Complete A–Z Guide

Contents

1 Introduction

This guide walks you step-by-step through everything you built: from creating project folders/-files, explaining why each module exists, to a full usage tutorial covering every language feature with plenty of code examples.

2 Project Structure (A–Z)

Your Azhar project follows a classic Python package layout:

```
azhar-latest/
     README.md
     LICENSE
     CHANGELOG.md
     pyproject.toml
     cli_entry.py         # frozen entry-point for PyInstaller
    azhar    /              # language implementation as a Python package
      __init__.py
      tokens.py
      errors.py
      ast.py
      lexer.py
      parser.py
      typechecker.py
      builtins.py
      interp.py
      repl.py
      cli.py
      modules/   # for any extension modules (optional)
    tests    /
      test_lexer.py
      test_parser.py
      test_typechecker.py
      test_interp.py
      test_integration.py
    dist    /                  # PyInstaller output: azhar.exe, hello.azhar
    build    /                 # build artifacts (safe to remove)

azhar_installer_script.iss # Inno Setup script for installer
```

Listing 1: Directory Structure

2.1 Why this structure?

- Keeps code modular (easy to maintain, scale, or freeze).
- Follows Python best-practices for packaging and distribution.
- Makes building a stand-alone `.exe` and an installer straightforward.

3 What Does Each File/Folder Do?

azhar/__init__.py Marks `azhar` as a package. Can be empty or define `__all__`.

tokens.py Defines all the token types and (usually) a `Token` class for lexing.

errors.py Contains custom error classes (for lexer errors, parse errors, type errors, runtime errors), providing user-friendly messages.

ast.py Defines the Abstract Syntax Tree: classes for each language construct (expressions, statements, etc.).

lexer.py Converts raw source code into a sequence of tokens using regular expressions or character scanning.

parser.py Turns a token stream into an AST (your program's structure), reporting syntax errors.

typechecker.py (was `types.py`) Checks the AST to catch type errors before running code (ensures int-to-int assignments, valid function argument types, etc.).

builtins.py Implements built-in functions (e.g., `print`, `output`, `read_int`, `read_string`) and registers them for the interpreter.

interp.py The interpreter: walks the AST, executing code.

repl.py Provides the interactive shell (Read-Eval-Print Loop), lets users type & run code directly (great for experimentation).

cli.py The main entry point for scripts/command-line. Handles file loading, error display, passing control to REPL or script runner.

modules/ For any extra language extensions or standard libraries you might want to implement later.

tests/ Each test file exercises a part of the compiler/interpreter (tokenization, parsing, type checking, etc.). For reliability and future development.

README.md Documentation, usage guide, language feature summary.

pyproject.toml (If you want PyPI installs) Defines build system, dependencies.

azhar_installer_script.iss Inno Setup script: builds a Windows installer that adds Azhar to PATH, associates `.azhar` files, includes icons, etc.

4 Building, Freezing, and Packaging

4.1 Step-by-Step (from your root folder):

4.1.1 Set up a virtual environment and install dependencies

```
python -m venv .venv
.venv\Scripts\activate
python -m pip install --upgrade pip pyinstaller
```

4.1.2 Compile to a single .exe

```
pyinstaller --onefile --name azhar --collect-all azhar cli_entry.py
# Output: dist/azhar.exe
```

4.1.3 Build an installer

1. Open `azhar_installer_script.iss` in Inno Setup.
2. Click “Build”.
3. Output: `Azhar-Language-Setup.exe` (ready for distribution)

5 Language Features: A Complete Tutorial

5.1 A. Hello World

```
print("Hello, world!")
```

Listing 2: Hello World

5.2 B. Variable Declaration and Assignment

```
let a: int = 3
let name: string = "Azhar"
let ok: bool = true

a = a + 1
ok = false
name = "Lang"
```

Listing 3: Variables

`let x: type = value` declares (and assigns) a variable.
Assignment uses `=` (must match declared type).

5.3 C. Data Types

- `int`: Integer numbers.
- `string`: Text.
- `bool`: `true` or `false`.
- `void`: Used for functions that don’t return a value.

5.4 D. Arithmetic & Comparisons

```
let x: int = 10
let y: int = 2
print(x + y)    # 12
print(x / y)    # 5
print(x == 10)  # true
print(x < y)    # false
```

Listing 4: Arithmetic

5.5 E. Input/Output

```
let age: int = read_int()
print("You are " + age + " years old!")
let text: string = read_string()
print(text)
```

Listing 5: I/O

`print(val)` outputs with newline.
`output(val)` outputs without newline.

5.6 F. Conditionals

```
let ok: bool = true
if ok do
    print(1)
else do
    print(2)
end
```

Listing 6: If-Else

5.7 G. Loops and Break

```
let i: int = 0
while i < 3 do
    print(i)
    if i == 1 do
        break
    end
    i = i + 1
end
```

Listing 7: While Loop

`while ... do` for loops.
`break` exits the loop.

5.8 H. Functions

```
function add(a:int, b:int) -> int do
    return a + b
end

print(add(4, 5))    # 9

function show() -> void do
    print("Hi!")
end
```

Listing 8: Functions

5.9 I. Types and Type Checking

Every function/variable must have a declared type.

TypeChecker enforces:

- Assignments match variable type.
- Function calls: argument count and type match.
- Return values match declared function return type.
- Using undeclared variables = error.

5.10 J. Errors and Diagnostics

All key errors (syntax, type, runtime) provide clear messages pointing to the relevant file and line number thanks to your custom error classes.

5.11 K. REPL (interactive shell) Usage

Run `azhar.exe` (no arguments) to enter an interactive shell:

Type code, press Enter, see result. Useful for learning or debugging.

5.12 L. Full Code Example: All Features

```
print("AZHAR")
function mul(a:int, b:int) -> int do
    return a * b
end
let x: int = 1
x = mul(x, 2)
print(x)
let name: string = read_string()
if name == "Azhar" do
    print("Welcome!")
else do
    print("Hello, " + name)
end
```

Listing 9: Complete Example

6 Why This Organization and Each Module?

Separation of Concerns: Each `.py` file targets one aspect (tokens, lexing, parsing, errors, AST, type checking, runtime, CLI, REPL, and built-ins), making code easy to understand, test, and extend.

Testing: Having `tests/` for each stage allows you to check for regressions when extending the language.

Packaging/Distribution: The split ensures PyInstaller can safely locate all imports, the installer is reliable, and every user—regardless of setup—can run Azhar easily.

7 Installation and Distribution

Run the generated installer (`Azhar-Language-Setup.exe`).

This places `azhar.exe` in Program Files, adds it to your PATH, and makes `.azhar` files double-clickable.

8 FAQ / Troubleshooting

- **Window closes instantly?** Run from CMD or use a `.bat` file that calls `azhar.exe file.azhar` followed by `pause`.
- **ImportError (azhar.types)?** Rename to `typechecker.py` and use `from azhar.typechecker import ....`
- **Frozen exe "No module named azhar"?** Use `-collect-all azhar` and ensure all imports are absolute.

9 Next Steps / Extending

- Add more built-in functions or modules under `azhar/modules/` and register them in `builtins.py`.
- Extend your stdlib or add more advanced language features!

This documentation gets anyone familiar with your Azhar language from setup, file-by-file understanding, to advanced program writing and installing for others to use.