Prompt Code Reference

.\README.md

A deterministic, context-aware **Python shim** for Windows that lets you control *which* Python interpreter is used by apps, projects, and background tools — without breaking the global environment.

Overview

pyshim is a lightweight command router that sits in front of `python.exe`.
It intercepts all calls to `python` and dynamically decides which interpreter to run based on context and configuration.

This is especially useful when:

- Multiple Python versions (e.g., 3.8, 3.11, 3.12) are installed.
- You want per-project or per-app version pinning.
- You need background tools and scripts to consistently use the same interpreter as your shell session.
- You don't want to fight Windows' confusing PATH order or `py launcher` behavior.

How It Works

When you call `python`, pyshim resolves the appropriate interpreter using this priority chain:

1. **One-shot flag** If called with `python --interpreter "SPEC" -- [args]`, that spec is used for this invocation only.
2. **Session override** If the environment variable `PYSHIM_INTERPRETER` is set (via `Use-Python` without `-Persist`), that spec is used.
3. **App-target override** If the environment variable `PYSHIM_TARGET` is set (e.g., `MyApp`), pyshim checks for `C:\bin\shims\python@MyApp.env`.
4. **Project-level pin** If a `.python-version` file exists in the current directory or any parent, that spec is used.
5. **Global persistence file** If `C:\bin\shims\python.env` exists (and persistence isn't disabled), pyshim uses that.
6. **Fallback chain** If no specific interpreter is found, pyshim falls back to:

   ```
   py -3.12 → py -3 → conda run -n base python → real python.exe → (error if none found)
   ```

   Note: The system requires the Windows Python Launcher (`py.exe`) or Conda to be installed. The fallback intentionally uses `py.exe` and searches for real `python.exe` outside the shim directory to

avoid infinite recursion (since `python.bat` IS the `python` command in your PATH).

Installation

1. **Install the Windows Python Launcher** (if not already installed):
   - Download and install any Python version from python.org
   - Check **"Install launcher for all users (recommended)"** during installation
   - This installs `py.exe` to `C:\Windows\` (required for fallback chain)
2. Create a directory for your shims (recommended: `C:\bin\shims`).
3. Copy these files from the repository:
   - `python.bat`
   - `pip.bat`
   - `pythonw.bat`
   - `pyshim.psm1`
4. Add `C:\bin\shims` to your **PATH** and move it to the top of the PATH order.
5. Import the PowerShell module from your profile:

```
Import-Module 'C:\bin\shims\pyshim.psm1'
```

6. Restart PowerShell.

Usage

Global Interpreter

Set the global default interpreter for all sessions:

```
Use-Python -Spec 'py:3.12' -Persist
```

This writes to `C:\bin\shims\python.env`:

```
py:3.12
```

Now, any call to `python`—including from apps and services—will use that version.

Session-Only Interpreter

```
Use-Python -Spec 'conda:tools'
```

This sets the interpreter for the **current shell session** only. Background apps will still use the global default.

Disable Global Persistence

To temporarily ignore the persisted version:

```
Disable-PythonPersistence
```

This creates `C:\bin\shims\python.nopersist`, which causes the shim to skip `python.env`.

To re-enable:

```
Enable-PythonPersistence
```

Per-App Overrides

You can pin specific apps to specific interpreters:

```
Set-AppPython -App 'MyService' -Spec 'conda:svc'
```

This creates `C:\bin\shims\python@MyService.env` containing:

```
conda:svc
```

When that app launches with `PYSHIM_TARGET=MyService`, it uses the pinned interpreter.

Example:

```
@echo off
set PYSHIM_TARGET=MyService
python -V
```

Per-Project Versions

Drop a `.python-version` file in your project root:

```
py:3.11
```

or

```
conda:myenv
```

When you run `python` inside that folder, pyshim automatically respects the project's version.

One-Shot Command Execution

You can also run a single command with a specific interpreter, without persistence:

```
Run-WithPython -Spec 'py:3.11' -- -m pip --version
```

Package Strategy

To keep your system clean and predictable:

- **Global CLI tools** → Install with `pipx`. Each tool gets its own isolated virtual environment.
- **Per-project dependencies** → Use a `.venv` created by the interpreter chosen by pyshim.
- **Cache for speed** → Set `PIP_CACHE_DIR=%LOCALAPPDATA%\pip\cache`.
- **Conda users** → Continue managing environments normally (`conda:envname` specs work seamlessly).

This hybrid model ensures:

- Background apps remain stable.
- Projects stay isolated.
- Installations reuse cached wheels for efficiency.

Quick Test

Once installed, open PowerShell and run:

```
Use-Python -Spec 'py:3.12' -Persist
python -V
pip --version
Run-WithPython -Spec 'py:3.11' -- -c "print('hello from 3.11')"
```

Example Directory Layout

```
C:\
└── bin\
    └── shims\
        ├── python.bat
        ├── pip.bat
        ├── pythonw.bat
        ├── pyshim.psm1
        ├── python.env
        ├── python@MyService.env
        └── python.nopersist
```

Naming Conventions

- `python.env` — global persistent interpreter spec.
- `.python-version` — project-local interpreter spec.
- `python@AppName.env` — per-application interpreter spec.
- `python.nopersist` — disables persistence globally.

Supported Spec Formats

Format	Description
`py:3.12`	Use Python 3.12 via Windows launcher.
`conda:myenv`	Use Conda environment `myenv`.
`C:\Path\to\python.exe`	Use this exact interpreter binary.

Example Workflows

Developer Switching Between Projects

```
cd ~/dev/project-a
python -V  # => Python 3.12 (from .python-version)

cd ~/dev/project-b
python -V  # => Python 3.10 (different .python-version)
```

Background Service Isolation

```
Set-AppPython -App 'DataIndexer' -Spec 'conda:data'
set PYSHIM_TARGET=DataIndexer
python -m indexer.main
```

Temporary Testing

```
Run-WithPython -Spec 'py:3.9' -- -c "import sys; print(sys.version)"
```

License

MIT License Copyright (c) 2025 ShruggieTech

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction...

(full license text included in LICENSE)

Links

- ShruggieTech
- Latest Releases
- dev-handbook Integration Docs

Credits

Designed and maintained by h8rt3rmin8r for **ShruggieTech LLC**. Originally conceived as part of the internal “dev-handbook” initiative for consistent Python environments across projects.

```
¯\_(ツ)_/¯
```

.\.gitignore

```
# See https://help.github.com/articles/ignoring-files/ for more about ignoring
files.

# dependencies
/node_modules
/.pnp
.pnp.js
.vscode
# testing
/coverage

# next.js
/.next/
/out/

# production
```

```
/build

# misc
.DS_Store
*.pem

# debug
npm-debug.log*
yarn-debug.log*
yarn-error.log*
.pnpm-debug.log*
```

.\examples\.python-version

```
C:\Users\you\miniconda3\envs\myproj\python.exe
```

.\.github\copilot-instructions.md

This document provides instructions for AI coding agents to effectively assist in developing the pyshim project.

Project Overview: pyshim

pyshim is a deterministic, context-aware Python shim system for Windows that intercepts `python`, `pip`, and `pythonw` calls to route them to the correct interpreter based on context. The system consists of three core components:

1. **Batch Shims** (`python.bat`, `pip.bat`, `pythonw.bat`) — Entry points that resolve interpreter specs through a priority chain and delegate to the actual interpreter
2. **PowerShell Module** (`pyshim.psm1`) — User-facing cmdlets for managing interpreter selection and persistence
3. **Config Files** (`python.env`, `python@AppName.env`, `.python-version`) — Text files storing interpreter specifications

Architecture & Resolution Priority

When `python.bat` is invoked, it resolves the interpreter using this exact priority chain:

1. **One-shot flag**: `--interpreter "SPEC" --` (used by `Run-WithPython` and direct invocation)
2. **Session variable**: `$env:PYSHIM_INTERPRETER` (set by `Use-Python` without `-Persist`)
3. **App-target override**: `python@%PYSHIM_TARGET%.env` (set by `Set-AppPython`)
4. **Project pin**: `.python-version` file in current directory or parent chain (walks up directory tree)
5. **Global persistence**: `python.env` (unless `python.nopersist` marker exists)
6. **Fallback chain**: `py -3.12` → `py -3` → `conda run -n base python` → real `python.exe` (not in shim dir) → (error if none found)

Important: The fallback chain avoids infinite recursion by:

- Using `PYSHIM_FROM_PY` guard variable when called from `py.bat`
- Skipping `python.exe` results that point to the shim directory itself
- Using explicit commands (`py.exe`, `conda`, absolute paths) rather than bare `python`

Interpreter Spec Formats

The system supports three spec formats (stored in `.env` and `.python-version` files):

- `py:3.12` — Uses Windows `py` launcher with specific version
- `conda:envname` — Uses Conda environment
- `C:\Path\to\python.exe` — Absolute path to interpreter binary

These specs are parsed by the `:RESOLVE_SPEC` subroutine in `python.bat` (lines 91-115).

Key Files & Their Roles

- **`python.bat`** (~137 lines): Core resolver logic with batch subroutines `:RESOLVE_SPEC` and `:FIND_DOTFILE` (walks directory tree for `.python-version`)
- **`pip.bat`** (4 lines): Trivial wrapper that calls `python.bat -m pip`
- **`pythonw.bat`** (4 lines): Best-effort headless wrapper (delegates to `python.bat`)
- **`pyshim.psm1`** (124 lines): PowerShell module with cmdlets `Use-Python`, `Disable-PythonPersistence`, `Enable-PythonPersistence`, `Set-AppPython`, `Run-WithPython`
- **`tests/smoke.ps1`**: Basic smoke test verifying `python -V`, `pip --version`, and `Run-WithPython`

Note: `py.bat` is NOT included — the shim relies on the native Windows Python Launcher (`py.exe`) being installed globally.

Critical Implementation Details

Batch File Constraints:

- Uses `ENABLEDELAYEDEXPANSION` for variable expansion in loops
- Subroutines use `call :LABEL` pattern with output variables passed by name (e.g., `call :RESOLVE_SPEC "%SPEC%" RESOLVED_CMD`)
- `:FIND_DOTFILE` implements parent directory walking using `%%~dpD` to extract parent paths
- Exit codes must be preserved with `exit /b %ERRORLEVEL%`
- **Critical**: Final fallback uses `py` (not `python`) to prevent infinite recursion since `python.bat` IS the `python` command in PATH

PowerShell Module Design:

- All cmdlets use `[CmdletBinding()]` with no parameters (lightweight functions)
- File operations use `-LiteralPath` for whitespace-safe handling
- Files written with `-NoNewline -Encoding ASCII` to avoid trailing newlines and BOM issues
- Hardcoded shim directory: `C:\bin\shims` (not parameterized — design choice for simplicity)

File Format Discipline:

- All `.env` files contain a single line with no trailing newline (enforced by `Set-Content -NoNewline`)
- Parsed with `for /f "usebackq delims="` in batch to preserve exact content

- `.python-version` files follow same single-line format

Development Workflows

Testing Changes:

```
## Run smoke test to verify basic functionality
.\tests\smoke.ps1

## Manual verification of resolution priority
Use-Python -Spec 'py:3.12' -Persist
python -V
```

Adding New Cmdlets:

- Follow existing pattern in `pyshim.psm1`
- Include full comment-based help with `.SYNOPSIS`, `.DESCRIPTION`, `.PARAMETER`, `.EXAMPLE`
- Use `$ShimDir = 'C:\bin\shims'` for path construction
- File writes should use `-NoNewline -Encoding ASCII`

Modifying Resolution Logic:

- Changes to priority chain happen in `python.bat` (lines 10-88)
- Spec parsing logic lives in `:RESOLVE_SPEC` subroutine (lines 91-115)
- Always preserve `%ERRORLEVEL%` when delegating to resolved interpreter

Common Pitfalls

- **Trailing Newlines**: Config files MUST NOT have trailing newlines (breaks batch parsing)
- **Path Separators**: Use `Join-Path` in PowerShell; avoid hardcoded `\` for potential Linux compatibility
- **Delayed Expansion**: Required in batch for variable mutation in loops/conditionals
- **Case Sensitivity**: Batch is case-insensitive, but `.python-version` filename is lowercase by convention (matching `pyenv`)
- **Infinite Recursion**: Never use bare `python` in fallback chain — `python.bat` IS the `python` command, so it would call itself. Always use `py.exe` or absolute paths in fallbacks

Design Philosophy

pyshim prioritizes **determinism** and **zero-configuration** over flexibility:

- No installer required — just copy files and update PATH
- Hardcoded paths (`C:\bin\shims`) for predictability
- Minimal dependencies (PowerShell 5.1+, Windows batch)
- Spec files use simple text format (no JSON/YAML parsing overhead)
- Global state is explicit (`.env` files in shim directory, not registry or AppData)

Standards for Writing Style and Tone

- Always write in a way that reads as genuinely human and free from any linguistic patterns that commonly expose AI-generated text. Avoid all "AI tells," including but not limited to: excessive politeness, generic transitions (e.g., "Furthermore," "In conclusion," "Overall"), filler phrases ("It's important to note that"), parallel-sounding constructions ("not only...but also"), and overly balanced or neatly summarized conclusions.
- Favor a natural flow that mirrors how an experienced writer or professional would actually communicate:
    - Use sentence length variation (occasional fragments are fine).
    - Use contractions naturally ("I'm," "don't," "that's").
    - Avoid corporate buzzwords and inflated adjectives unless context demands them.
    - Use clarity, subtle rhythm, and precision over "niceness."
    - Never overexplain or restate points unless it aids comprehension.
    - Eliminate template phrasing like "Here's a breakdown," "Let's explore," or "This means that."
- The tone should sound authentic, thoughtful, and intentional, not algorithmically tidy. Write as if the content were edited by a sharp human who values brevity, rhythm, and nuance more than structure or formality.
- Use a little bit of sarcasm or dry humor where appropriate and have fun talking trash about tools and frameworks developed by Microsoft (they deserve it).

System Environment Handling

Sensitivity to File and Directory Names

While this project is likely to be developed in a Windows 11 environment, it should be assumed that some (or all) of the code may be run in an Ubuntu Linux production environment. Windows file paths are case-insensitive, while Linux file paths use a much more robust case-sensitive approach. As such, when writing scripts that interact with the filesystem, care must be taken to ensure that file and directory names are treated in a universally compatible manner at all times. In addition to the matter of case sensitivity, Windows file systems use (for some unknowable reason) a `\` (backslash "escape") separator character, while Linux file systems use a `/` (forward slash) separator. So while it would be convenient to treat all paths like Internet URLs like Linux, when writing scripts that interact with the filesystem, care must be taken to ensure that file and directory paths are constructed and parsed correctly for the target operating system (thanks to Microsoft).

In Powershell, use a standard discovery method to determine the appropriate path separator for the current operating system and store that separator in a variable: `$Sep` (adapt this method for other scripting languages as needed):

```
$Sep = [IO.Path]::DirectorySeparatorChar
```

File and directory names should avoid spaces where possible. However, scripts must always account for cases where whitespace exists. When handling paths or user inputs in PowerShell, use the `-LiteralPath` parameter where supported (instead of the intuitive but ill-advised `-Path` parameter). Always verify the compatibility of the `-LiteralPath` parameter with each cmdlet to prevent errors when processing path references, as some cmdlets do NOT support `-LiteralPath`.

On a side note, one of the original creators of PowerShell publicly complained about the broken state of the `-Path` parameter and its inconsistent handling of special characters. So (as is the case with all dumpster fire code written by Microsoft) this is a known fundamental bad-practice end-user pain point that Microsoft just simply ignores (okay I'm good now - moving on).

Python Versioning and Management

- The system-wide Python installation is exactly version `3.12.10`. This should be taken into account when writing or updating scripts that may interact with the Python installation or its packages.
- Whenever possible, try to use virtual environments for Python projects to avoid dependency conflicts and ensure consistent behavior across different development and production environments.
- Consider using python-poetry.org for managing Python project dependencies and virtual environments.
- If Poetry is not installed, it can be installed using one of the following commands:

  Install Poetry on Windows 11 (Powershell):

  ```
  (Invoke-WebRequest -Uri https://install.python-poetry.org -
  UseBasicParsing).Content | py -
  ```

  Install Poetry on Ubuntu Linux (Bash):

  ```
  curl -sSL https://install.python-poetry.org | python3 -
  ```

  Note: You may need to also set up your system PATH to include Poetry's bin directory. Refer to the official Poetry documentation for guidance.
- Python is another dumpster fire situation. Backwards compatibility is a nightmare, and the ecosystem is riddled with poorly maintained packages and security vulnerabilities, and almost every project prides itself on reinventing the wheel. Always expect the worst and check the maintenance status of any third-party libraries before including them in a project, and only include well-established, actively maintained packages AT ALL TIMES.

General Code Formatting

- **Nested Helper Functions:** For complex functions, break down logic into smaller, single-purpose nested helper functions. These helpers should follow a `ParentFunctionName-HelperAction` naming convention (e.g., `ApkExtract-ResolvePath`).
- **Variable Naming Convention:** All variables should use **PascalCase** (e.g., `$NumbersCount`, `$InputFile`, `$ExitCode`). Do not use snake_case or camelCase unless absolutely necessary (like conformance with existing third-party code).
- **Clean Whitespace:**

    - Never include a line that contains only whitespace characters. If a blank line is needed for readability, it must be completely empty.
    - Never leave trailing whitespace at the end of any line.
    - Always leave a single blank line between major logical sections of code (e.g., between function declarations)
    - Never indent code using tab characters. Always use exactly **four spaces** for each level of indentation.

- **Brace Style (One True Brace Style):** When declaring functions, `if` statements, loops, or any other code block, the opening curly brace `{` **must** be on the same line as the declaration. The closing curly brace `}` **must** be on its own line, aligned with the start of the declaration.

```
# Correct
function Get-Something {
    if ($Condition) {
        # Do work
    }
}

# Incorrect
function Get-Something
{
    # ...
}
```

Scripting Standards (Powershell Focused)

All non-Powershell scripts should include an appropriate shebang line at the top of the file:

- Bash: `#!/usr/bin/env bash`
- Python: `#!/usr/bin/env python3`
- Node.js: `#!/usr/bin/env node`
- etc.

Important Note: Never include a shebang line in Powershell scripts. Doing so will prevent the script from behaving correctly in Windows environments.

All scripts and functions should closely adhere to the following general structure:

1. Introduction
    - **Comprehensive ReadMe:** Comprehensive comment-based Help Text documentation
    - `[CmdletBinding()]` **Declaration:** (Powershell only)
    - `Param()` **Block:** Define all input parameters (Powershell only)
2. Declarations
    - **Function Declarations:** Function and sub-function declarations in alphabetical order
    - **Variable & Array Declarations:** Variable and array declarations, including self-awareness variables (be sure to carefully sequence these correctly to avoid dependency issues when initializing variables that depend on other locally declared variables)

3. Core Logic
    - **Catch Help Text Requests:** Display the help text and gracefully exit if the `-Help` parameter is specified
    - **Validate Inputs:** Validate user inputs or required environment variables if necessary
    - **Main Process Logic:** The core logic of the script or function, broken down into logical sections with clear comments explaining each part
4. Conclusion
    - **Return Output:** Return outputs to the stdout stream and/or write require output files as needed
    - **Garbage Collection:** Clean up any temporary files or resources used during execution
    - **Exit Gracefully:** Exit with an appropriate exit code indicating success or failure (depending on the language, this may be implicit)

Comprehensive ReadMe

Example comment-based help block for Powershell:

```
<#
.SYNOPSIS
    A brief summary of the function's purpose.
.DESCRIPTION
    A more detailed description of what the function does, how it behaves, and the
kinds of inputs and outputs it handles.
.PARAMETER ParameterName
    A clear explanation of what this parameter is for along with any constraints
or special behaviors.
.EXAMPLE
    A practical example of how to use the function.
.LINK
    [example.com](https://example.com/)
#>
```

- When writing any Powershell script (or function or sub-function), always include a full comment-based help block at the very beginning of the file and before the internal logic of each function.
- At a minimum, in Powershell, this block must include `.SYNOPSIS`, `.DESCRIPTION`, `.PARAMETER` for each parameter, and at least one `.EXAMPLE`. Include a `.LINK` for external references where applicable.
- Relevant help text should be included in non-Powershell scripts as well (using appropriate interactive features and appropriate comment syntax).

CmdletBinding (Powershell)

Example `[CmdletBinding()]` declaration:

```
[CmdletBinding(SupportsShouldProcess=$true,ConfirmImpact='None',DefaultParameterSe
tName='Default')]
```

When writing PowerShell functions and scripts, always include a proper `[CmdletBinding()]` declaration directly following the comment-based help text. This ensures that the function behaves like a standard cmdlet, supporting common features like `-Verbose`, `-Debug`, and `-ErrorAction`.

- Avoid using empty whitespace inside the parentheses of `CmdletBinding()`. This line is already quite long, so keep it tight.
- If the function performs actions that change system state (e.g., file operations), consider including `SupportsShouldProcess=$true` in the declaration.
- Include `ConfirmImpact='None'` unless the function performs high-impact actions (who knows whatever that means).
- Always specify a `DefaultParameterSetName='Default'` to ensure predictable behavior regardless of how many parameter sets are defined.
- Additional attributes can be included if doing so would facilitate the function's specific needs and assumed use-cases (such as handling of pipeline inputs, etc). Make sure to understand the implications of each attribute before including it as these can completely break an entire script or function if misused.

Param Block (Powershell)

Define all parameters within a `Param()` block immediately following the `[CmdletBinding()]` declaration. This ensures standard cmdlet behavior. Be sure to leave a blank line between each clump of attributes related to a single parameter for readability.

Example `Param()` block:

```
Param(
    [Parameter(Mandatory=$true,ParameterSetName='Default')]
    [Alias("f")]
    [System.String]$File,

    [Parameter(Mandatory=$false,ParameterSetName='Default')]
    [Alias("o","outfile")]
    [System.String]$Output = "ProjectOutput.txt",

    [Parameter(Mandatory=$true,ParameterSetName='HelpText')]
    [Alias("h")]
    [Switch]$Help
)
```

- **Named Parameter Groups:** Organize parameters into logical groups using the `ParameterSetName` attribute. This helps clarify which parameters can be used together and improves usability and discoverability from within the standard Help Text generator.
- **Parameter Attributes:** Use attributes like `[Parameter(Mandatory=$true)]` to enforce required parameters within the context of named parameter groups. The Help text parameter (`-Help`) should always inhabit a `HelpText` parameter group and be mandatory within that group.
- **Parameter Aliases:** Provide common, best-practice aliases for parameters to improve usability (e.g., `[Alias("f","file","inputfile")]`) but carefully avoid overly generic aliases that could conflict

with other parameters. Conflicts in parameter names and aliases should be strongly avoided. Always remember that parameters are case-insensitive in PowerShell
    - Example: Including both `-File` and `-file` aliases would break the script or function.
- **Type Constraints:** Strongly type all parameters (e.g., `[System.String]`, `[System.Boolean]`, `[Switch]`).
- **Default Values:** If an input has a highly-likely default value, assign default values to optional parameters directly in the `Param()` block (e.g., `$Verbosity = $true`) and avoid assigning defaults to mandatory parameters (as this angers the syntax parsers in Visual Studio).

Self-Awareness Variables

For effective logging and verbosity, functions and scripts should all declare "self-awareness" variables at the beginning to establish a caller reference string. This is especially important for nested functions to create a logical call stack.

Example self-awareness variable declarations:

```
## Internal self-awareness variables for use in verbosity and logging
$thisFunctionReference = "{0}" -f $MyInvocation.MyCommand
$thisSubFunction = "{0}" -f $MyInvocation.MyCommand
$thisFunction = if ($null -eq $thisFunction) { $thisSubFunction } else { -
join("$thisFunction", ":", "$thisSubFunction") }
```

.\prompt\Make-PromptCodeReference.ps1

```
<#
.SYNOPSIS
    Generates a Markdown code reference file from project root files (intended for
use in AI prompting).
.DESCRIPTION
    Creates a comprehensive Markdown document containing all files from the
project root directory
    with appropriate syntax highlighting. Excludes specific files like LICENSE and
workspace files.
    README.md is processed first with headers downgraded one level. Each file is
formatted as a
    level-two header followed by a code block with proper language syntax
highlighting.
.PARAMETER Directory
    The directory to scan for files. Defaults to the project root (..\). Use this
to point the
    script at a different directory for processing.
    Alias: d
.PARAMETER Recurse
    When specified, recursively scans subdirectories within the target directory.
By default,
    only files in the root of the target directory are processed (non-recursive).
    Alias: r, recursive
```

```
.PARAMETER Force
    When specified, includes hidden files in the output. By default, hidden files
are excluded.
    Alias: f
.PARAMETER IncludeMarkdown
    When specified, includes the full content of all Markdown files (other than
README.md).
    By default, non-README Markdown files are listed with a placeholder message
instead of
    their full content.
    Alias: i, includemd
.PARAMETER Help
    Displays detailed help information for this script.
    Alias: h
.EXAMPLE
    .\Make-PromptCodeReference.ps1

    Generates a code reference from files in the project root directory (non-
recursive).
.EXAMPLE
    .\Make-PromptCodeReference.ps1 -Directory 'C:\MyProject'

    Generates a code reference from files in C:\MyProject (non-recursive).
.EXAMPLE
    .\Make-PromptCodeReference.ps1 -Recurse

    Generates a code reference from all files in the project root and
subdirectories (recursive).
.EXAMPLE
    .\Make-PromptCodeReference.ps1 -d 'C:\MyProject' -r

    Generates a code reference from all files in C:\MyProject and subdirectories
using parameter aliases.
.EXAMPLE
    .\Make-PromptCodeReference.ps1 -Force

    Generates a code reference from the project root including hidden files.
.EXAMPLE
    .\Make-PromptCodeReference.ps1 -IncludeMarkdown

    Generates a code reference including the full content of all Markdown files
(other than README.md).
.LINK
    https://github.com/shruggietech/pyshim
#>

[CmdletBinding(SupportsShouldProcess=$false,ConfirmImpact='None',DefaultParameterS
etName='Default')]
Param(
    [Parameter(Mandatory=$false,ParameterSetName='Default')]
    [Alias("d")]
    [System.String]$Directory,

    [Parameter(Mandatory=$false,ParameterSetName='Default')]
```

```
    [Alias("r","recursive")]
    [Switch]$Recurse,

    [Parameter(Mandatory=$false,ParameterSetName='Default')]
    [Alias("f")]
    [Switch]$Force,

    [Parameter(Mandatory=$false,ParameterSetName='Default')]
    [Alias("i","includemd")]
    [Switch]$IncludeMarkdown,

    [Parameter(Mandatory=$true,ParameterSetName='HelpText')]
    [Alias("h")]
    [Switch]$Help
)

# Internal self-awareness variables for use in verbosity and logging
$ThisScriptPath = $MyInvocation.MyCommand.Path
$ThisFunctionReference = "{0}" -f $MyInvocation.MyCommand
$ThisSubFunction = "{0}" -f $MyInvocation.MyCommand
$ThisFunction = if ($null -eq $ThisFunction) { $ThisSubFunction } else { -
join("$ThisFunction", ":", "$ThisSubFunction") }

# Catch help text requests
if (($Help) -or ($PSCmdlet.ParameterSetName -eq 'HelpText')) {
    Get-Help "$ThisScriptPath" -Detailed
    exit
}

#-------------------------------------------------------------------------------
# Configuration
#-------------------------------------------------------------------------------

$ProjectRoot = if ($Directory) { $Directory } else { Join-Path $PSScriptRoot '..'
}
$OutputFile = Join-Path $PSScriptRoot 'Prompt-Code-Reference.md'
$ExcludedFiles = @('LICENSE')

# Map file extensions to GitHub Markdown syntax highlighting tags
$SyntaxMap = @{
    '.bash'             = 'bash'
    '.bat'              = 'batch'
    '.c'                = 'c'
    '.cc'               = 'cpp'
    '.cfg'              = 'ini'
    '.clj'              = 'clojure'
    '.cmd'              = 'batch'
    '.code-workspace'   = 'json'
    '.conf'             = 'conf'
    '.cpp'              = 'cpp'
    '.cs'               = 'csharp'
    '.css'              = 'css'
    '.cxx'              = 'cpp'
    '.dockerfile'       = 'dockerfile'
```

```
    '.env'              = 'bash'
    '.gitignore'        = 'gitignore'
    '.go'               = 'go'
    '.h'                = 'c'
    '.hpp'              = 'cpp'
    '.htm'              = 'html'
    '.html'             = 'html'
    '.ini'              = 'ini'
    '.java'             = 'java'
    '.js'               = 'javascript'
    '.json'             = 'json'
    '.jsx'              = 'jsx'
    '.kt'               = 'kotlin'
    '.less'             = 'less'
    '.lua'              = 'lua'
    '.markdown'         = 'markdown'
    '.md'               = 'markdown'
    '.perl'             = 'perl'
    '.php'              = 'php'
    '.pl'               = 'perl'
    '.ps1'              = 'powershell'
    '.psd1'             = 'powershell'
    '.psm1'             = 'powershell'
    '.py'               = 'python'
    '.r'                = 'r'
    '.rb'               = 'ruby'
    '.rs'               = 'rust'
    '.sass'             = 'sass'
    '.scala'            = 'scala'
    '.scss'             = 'scss'
    '.sh'               = 'bash'
    '.sql'              = 'sql'
    '.swift'            = 'swift'
    '.tex'              = 'latex'
    '.toml'             = 'toml'
    '.ts'               = 'typescript'
    '.tsx'              = 'tsx'
    '.txt'              = 'text'
    '.vim'              = 'vim'
    '.xml'              = 'xml'
    '.yaml'             = 'yaml'
    '.yml'              = 'yaml'
    '.zsh'              = 'bash'
}

#--------------------------------------------------------------------------------
# Main Process
#--------------------------------------------------------------------------------

Write-Host "Generating Prompt Code Reference from project root files..." -
ForegroundColor Green
Write-Host "  Target directory: $(Resolve-Path -LiteralPath $ProjectRoot)" -
ForegroundColor Cyan
```

```powershell
# Delete existing output file if it exists to prevent it from being included in
processing
if (Test-Path -LiteralPath $OutputFile) {
    Remove-Item -LiteralPath $OutputFile -Force -ErrorAction SilentlyContinue
}

# Get all files from project root (recursive if -Recurse specified)
# Note: Get-ChildItem without -Force already excludes hidden files by default
$GetChildItemParams = @{
    LiteralPath = $ProjectRoot
    File = $true
}

if ($Recurse) {
    $GetChildItemParams['Recurse'] = $true
}

$RootFiles = Get-ChildItem @GetChildItemParams | Where-Object {
    # Exclude files in exclusion list
    if ($_.Name -in $ExcludedFiles) {
        return $false
    }
    # Unless -Force is specified, exclude hidden files and dotfiles
    if (-not $Force) {
        # Exclude files with Hidden attribute
        if ($_.Attributes -band [System.IO.FileAttributes]::Hidden) {
            return $false
        }
        # Exclude dotfiles (files starting with .)
        if ($_.Name.StartsWith('.')) {
            return $false
        }
    }
    return $true
} | Sort-Object Name

if ($RootFiles.Count -eq 0) {
    Write-Warning "No files found in project root to process."
    exit 1
}

Write-Host "  Found $($RootFiles.Count) files to process" -ForegroundColor Cyan

# Initialize output content
$MarkdownContent = @()
$MarkdownContent += "# Prompt Code Reference"
$MarkdownContent += ""

# Check for README file variants and process first
# Match: README.md, README (no extension), README.markdown, etc.
# Exclude: README.txt (treat as plain text)
$ReadmeFile = $RootFiles | Where-Object {
    $_.BaseName -eq 'README' -and $_.Extension -ne '.txt'
} | Select-Object -First 1
```

```powershell
if ($ReadmeFile) {
    Write-Host "  Processing: $($ReadmeFile.FullName) (with header
adjustments)..." -ForegroundColor Yellow

    try {
        $ReadmeContent = Get-Content -LiteralPath $ReadmeFile.FullName -Raw -
ErrorAction Stop

        # Downgrade all headers (deepest first to avoid collisions)
        # Level 6 -> 7 (but markdown only supports up to 6, so these become
invalid)
        # Level 5 -> 6
        # Level 4 -> 5
        # Level 3 -> 4
        # Level 2 -> 3
        # Level 1 -> 2
        $ReadmeContent = $ReadmeContent -replace '(?m)^######\s+', '####### '
        $ReadmeContent = $ReadmeContent -replace '(?m)^#####\s+', '###### '
        $ReadmeContent = $ReadmeContent -replace '(?m)^####\s+', '##### '
        $ReadmeContent = $ReadmeContent -replace '(?m)^###\s+', '#### '
        $ReadmeContent = $ReadmeContent -replace '(?m)^##\s+', '### '
        $ReadmeContent = $ReadmeContent -replace '(?m)^#\s+', '## '

        # Remove the first level-2 header line only (which was originally level-1)
        $ReadmeContent = $ReadmeContent -replace '^##\s+[^\r\n]+(\r?\n)?', ''

        # Fix list formatting: Insert blank line before list items that don't
follow other list items
        # Match lines starting with '-' where the previous line doesn't start with
'-' or whitespace+'-'
        $ReadmeContent = $ReadmeContent -replace '(?m)(?<=^(?!.*\s*-)[^\r\n]+\r?
\n)(?=\s*-)', "`n"

        # Get relative path for README
        $RelativePath = ".\$($ReadmeFile.Name)"

        # Always add README.md as level-2 header at the top with relative path
        $ReadmeContent = "## ``$RelativePath```n`n" + $ReadmeContent.TrimStart()

        # Add the processed README content
        $MarkdownContent += $ReadmeContent.TrimEnd()
        $MarkdownContent += ""
        $MarkdownContent += ""
    }
    catch {
        Write-Warning "Failed to read README.md: $_"
    }
}

# Process all other files (excluding README variants)
$OtherFiles = $RootFiles | Where-Object {
    -not ($_.BaseName -eq 'README' -and $_.Extension -ne '.txt')
}
```

```
foreach ($File in $OtherFiles) {
    Write-Host "  Processing: $($File.FullName)" -ForegroundColor Yellow

    # Calculate relative path from project root
    $FullProjectRoot = (Resolve-Path -LiteralPath $ProjectRoot).Path
    $FullFilePath = $File.FullName

    if ($FullFilePath.StartsWith($FullProjectRoot)) {
        $RelativePath =
$FullFilePath.Substring($FullProjectRoot.Length).TrimStart('\', '/')
        $RelativePath = ".\$RelativePath"
    } else {
        $RelativePath = ".\$($File.Name)"
    }

    # Check if this is a Markdown file (excluding README.md which is already
processed)
    $IsMarkdown = $File.Extension -eq '.md'

    # If it's a Markdown file and -IncludeMarkdown is NOT specified, add
placeholder
    if ($IsMarkdown -and -not $IncludeMarkdown) {
        $MarkdownContent += "## ``$RelativePath``"
        $MarkdownContent += ""
        $MarkdownContent += "(All non-readme markdown content is excluded by
default from this project summary document.)"
        $MarkdownContent += ""
        continue
    }

    # Determine syntax highlighting language
    $Extension = $File.Extension.ToLower()
    $Language = if ($SyntaxMap.ContainsKey($Extension)) {
        $SyntaxMap[$Extension]
    } else {
        'text'
    }

    # Read file content
    try {
        $FileContent = Get-Content -LiteralPath $File.FullName -Raw -ErrorAction
Stop

        # If this is a Markdown file (and -IncludeMarkdown was specified), process
headers
        if ($IsMarkdown) {
            # Downgrade all headers (deepest first to avoid collisions)
            $FileContent = $FileContent -replace '(?m)^######\s+', '####### '
            $FileContent = $FileContent -replace '(?m)^#####\s+', '###### '
            $FileContent = $FileContent -replace '(?m)^####\s+', '##### '
            $FileContent = $FileContent -replace '(?m)^###\s+', '#### '
            $FileContent = $FileContent -replace '(?m)^##\s+', '### '
            $FileContent = $FileContent -replace '(?m)^#\s+', '## '
```

```
            # Remove the first level-2 header line only (which was originally
level-1)
            $FileContent = $FileContent -replace '^##\s+[^\r\n]+(\r?\n)?', ''

            # Fix list formatting: Insert blank line before list items that don't
follow other list items
            # Match lines starting with '-' where the previous line doesn't start
with '-' or whitespace+'-'
            $FileContent = $FileContent -replace '(?m)(?<=^(?!.*\s*-)[^\r\n]+\r?
\n)(?=\s*-)', "`n"

            # Add header with relative path and processed content
            $MarkdownContent += "## ``$RelativePath``"
            $MarkdownContent += ""
            $MarkdownContent += $FileContent.TrimStart().TrimEnd()
            $MarkdownContent += ""
        } else {
            # Add header and code block with relative path in backticks
            $MarkdownContent += "## ``$RelativePath``"
            $MarkdownContent += ""
            $MarkdownContent += "``````$Language"
            $MarkdownContent += $FileContent.TrimEnd()
            $MarkdownContent += "``````"
            $MarkdownContent += ""
        }
    }
    catch {
        Write-Warning "Failed to read file: $($File.Name) - $_"
        continue
    }
}

# Write output file (force clobber if exists)
try {
    $FinalContent = ($MarkdownContent -join "`n")

    # Replace three sequential line breaks with two line breaks
    $FinalContent = $FinalContent -replace '(\r?\n){3}', "`n`n"

    $FinalContent | Set-Content -LiteralPath $OutputFile -NoNewline -Encoding UTF8
-Force -ErrorAction Stop
    Write-Host ""
    Write-Host "Successfully generated: $OutputFile" -ForegroundColor Green
    Write-Host "  Total files processed: $($RootFiles.Count)" -ForegroundColor
Cyan
}
catch {
    Write-Error "Failed to write output file: $_"
    exit 1
}
```

.\bin\shims\pip.bat

```
@echo off
setlocal
REM ensure pip matches whatever python.bat resolved
"%~dp0python.bat" -m pip %*
```

.\examples\profile.ps1

```
# Quiet verbose unless you explicitly opt in later
$VerbosePreference = 'SilentlyContinue'

# ===== ShruggieTech Python Shim Profile =====
# Ensures the shim is first on PATH and loads helper functions

# --- 1) PATH: ensure C:\bin\shims is first and unique
$ShimDir = 'C:\bin\shims'

# Bridge for when Windows Launcher is missing
if (-not (Get-Command py -ErrorAction SilentlyContinue)) {
    Set-Alias py "$ShimDir\python.bat" -Scope Global
}

# Split PATH into parts, remove empties, sort and unique
$pathParts = ($env:PATH -split ';') | Where-Object { $_ -and $_.Trim() -ne '' } |
Sort-Object -Unique

# Remove all occurrences of ShimDir
$pathParts = $pathParts | Where-Object { $_ -ne $ShimDir }

# Prepend ShimDir
$env:PATH = ($ShimDir + ';' + ($pathParts -join ';'))

# Optional: make sure pipx shims are reachable
$PipxBin = Join-Path $env:USERPROFILE '.local\bin'
if (-not ($env:PATH -split ';' | Where-Object { $_ -eq $PipxBin })) {
    $env:PATH = "$PipxBin;$env:PATH"
}

# --- 2) Import pyshim module (provides Use-Python, Set-AppPython, etc.)
$PyShimModule = Join-Path $ShimDir 'pyshim.psm1'
if (Test-Path -LiteralPath $PyShimModule) {
    Import-Module $PyShimModule -Force -DisableNameChecking
} else {
    Write-Warning "pyshim module not found at $PyShimModule. Install shims and
module from repo."
}

# --- 3) DO NOT override the Windows 'py' launcher
# If you previously had: New-Alias py python  => REMOVE IT.
```

```
# The py launcher is used by specs like 'py:3.12' and must remain available.

# --- 4) Friendly helpers -------------------------------------------------

function Show-PyShim {
    Write-Host "PATH[0] = $ShimDir"
    Write-Host "pyshim module loaded: " -NoNewline
    if (Get-Module -Name (Split-Path $PyShimModule -Leaf) -ErrorAction
SilentlyContinue) {
        Write-Host "yes"
    } else {
        Write-Host "no"
    }

    $globalEnv = Join-Path $ShimDir 'python.env'
    $nopersist = Join-Path $ShimDir 'python.nopersist'
    if (Test-Path $globalEnv) {
        $spec = Get-Content -LiteralPath $globalEnv -Raw
        Write-Host "Global spec (python.env): $spec"
    } else {
        Write-Host "Global spec (python.env): <none>"
    }
    Write-Host "Global persistence disabled? " -NoNewline
    Write-Host ($(Test-Path $nopersist))
    if ($env:PYSHIM_INTERPRETER) { Write-Host "Session spec:
$($env:PYSHIM_INTERPRETER)" }
    if ($env:PYSHIM_TARGET)      { Write-Host "App target : $($env:PYSHIM_TARGET)"
}
}

# A quick bootstrap: if you want a default interpreter for new shells but not when
nopersist is set
$DefaultSpec = $null   # e.g. 'py:3.12' or 'conda:base'  (leave $null to do
nothing)
$NoPersistMarker = Join-Path $ShimDir 'python.nopersist'
$GlobalEnvFile  = Join-Path $ShimDir 'python.env'

if ($DefaultSpec -and -not (Test-Path $NoPersistMarker) -and -not (Test-Path
$GlobalEnvFile)) {
    try {
        Use-Python -Spec $DefaultSpec -Persist
        Write-Host "Initialized global Python spec -> $DefaultSpec"
    } catch {
        Write-Warning "Failed to initialize global Python spec:
$($_.Exception.Message)"
    }
}

# --- 5) Conda wrapper that cooperates with pyshim ----------------------------
# When you run: conda activate <env>, set only the *session* spec to that env.
# This keeps the shell consistent without flipping global persistence.
function conda {
    # Call the real conda
    $RealConda = Join-Path $env:USERPROFILE 'miniconda3\Scripts\conda.exe'
```

```
    if (-not (Test-Path -LiteralPath $RealConda)) {
        Write-Error "Conda not found at $RealConda"; return
    }
    & $RealConda @Args

    # Detect "conda activate <env>"
    if ($Args.Count -ge 2 -and $Args[0] -eq 'activate') {
        $envName = $Args[1]
        # Session-only switch to that env
        try {
            Use-Python -Spec "conda:$envName"
            Write-Host "Session Python -> conda:$envName (not persisted)"
        } catch {
            Write-Warning ("Failed to set session interpreter to
conda:$($envName): $($_.Exception.Message)")
        }
    }
}

# --- 6) Convenience: quick commands ----------------------------------------
Set-Alias which Get-Command -Scope Global -ErrorAction SilentlyContinue

function pyver { & "$ShimDir\python.bat" -V }
function pipver { & "$ShimDir\python.bat" -m pip --version }

# Uncomment if you want to see current shim state on every new shell
# Show-PyShim
```

.\pyshim.code-workspace

```
{
    "folders": [
        {
            "name": "pyshim",
            "path": "."
        }
    ],
    "settings": {
        "powershell.cwd": "pyshim",
        "powershell.buttons.showPanelMovementButtons": true,
        "powershell.enableReferencesCodeLens": true,
        "powershell.codeFormatting.autoCorrectAliases": true,
        "powershell.codeFormatting.avoidSemicolonsAsLineTerminators": true,
        "powershell.codeFormatting.ignoreOneLineBlock": true,
        "powershell.codeFormatting.openBraceOnSameLine": true,
        "powershell.codeFormatting.whitespaceAfterSeparator": false,
        "powershell.codeFormatting.whitespaceAroundOperator": false,
        "powershell.codeFormatting.whitespaceBeforeOpenParen": true,
        "powershell.analyzeOpenDocumentsOnly": true,
        "editor.foldingStrategy": "indentation",
        /*"editor.defaultFormatter": "ms-vscode.powershell",*/
```

```
        "editor.formatOnSave": false,
        "editor.tabSize": 4,
        "editor.insertSpaces": true,
        "editor.detectIndentation": true,
        "editor.wordWrap": "on",
        "editor.minimap.enabled": true,
        "editor.renderWhitespace": "all",
        "editor.renderControlCharacters": true,
        "editor.renderLineHighlight": "all",
        "editor.renderFinalNewline": "on",
        "editor.rulers": [
            80,
            84,
            88,
            120,
            124,
            128,
            160
        ],
        "editor.codeLens": false,
        "editor.fontSize": 12,
        "editor.fontLigatures": true,
        "editor.lineHeight": 20,
        "editor.letterSpacing": 0.6,
        "editor.cursorBlinking": "smooth",
        "editor.cursorSmoothCaretAnimation": "on",
        "editor.cursorStyle": "line",
        "editor.cursorWidth": 2,
        "editor.cursorSurroundingLines": 3,
        "editor.cursorSurroundingLinesStyle": "default",
        "editor.hover.delay": 3000,
        "workbench.hover.delay": 3000,
        "workbench.sash.hoverDelay": 2000,
        "inlineChat.lineEmptyHint": false,
        "json.format.enable": true,
        "json.validate.enable": true,
        "json.schemas": [
            {
                "fileMatch": [
                    "manifest.json",
                    "manifest.webmanifest",
                    "app.webmanifest"
                ],
                "url": "https://json.schemastore.org/web-manifest.json"
            },{
                "fileMatch": [
                    "*_meta.json",
                    "*_directorymeta.json",
                    "*-feeds-export_index.json"
                ],
                "url": "https://cdn.h8rt3rmin8r.com/schemas/MakeIndex.meta.json"
            }
        ],
        "[powershell]": {
```

```
            "editor.defaultFormatter": "ms-vscode.powershell"
        },
        "[log]": {
            "editor.wordWrap": "off"
        }
    }
}
```

.\bin\shims\pyshim.psm1

```
# Save this file here: C:\bin\shims\pyshim.psm1
# Import this file from your $PROFILE in Powershell

function Use-Python {
    <#
    .SYNOPSIS
        Choose a Python interpreter for this session and/or persist it globally.
    .DESCRIPTION
        SPEC accepts absolute path, 'py:3.12', 'py:3', or 'conda:ENV'.
    .PARAMETER Spec
        Interpreter spec.
    .PARAMETER Persist
        Write SPEC to C:\bin\shims\python.env (unless nopersist marker exists).
    .PARAMETER NoPersist
        Delete C:\bin\shims\python.env (session keeps $env:PYSHIM_INTERPRETER
only).
    .EXAMPLE
        Use-Python -Spec 'py:3.12' -Persist
    .EXAMPLE
        Use-Python -Spec 'conda:tools'   # session-only
    #>
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$false)]
        [System.String]$Spec,

        [Switch]$Persist,

        [Switch]$NoPersist
    )

    $ShimDir = 'C:\bin\shims'
    $GlobalEnv = Join-Path $ShimDir 'python.env'
    $NoPersistMarker = Join-Path $ShimDir 'python.nopersist'

    if ($NoPersist) {
        if (Test-Path $GlobalEnv) { Remove-Item -LiteralPath $GlobalEnv -Force }
        $env:PYSHIM_INTERPRETER = $null
        Write-Host "Global persistence disabled for future calls (file removed)."
-ForegroundColor Yellow
        return
```

```
    }

    if ($Spec) {
        $env:PYSHIM_INTERPRETER = $Spec
        Write-Host "Session interpreter -> $Spec"
        if ($Persist) {
            if (Test-Path $NoPersistMarker) {
                Write-Warning "Global nopersist marker is present; not writing
python.env."
            } else {
                Set-Content -LiteralPath $GlobalEnv -Value $Spec -NoNewline -
Encoding ASCII
                Write-Host "Persisted globally -> $GlobalEnv"
            }
        }
    } else {
        if (Test-Path $GlobalEnv) {
            $env:PYSHIM_INTERPRETER = Get-Content -LiteralPath $GlobalEnv -Raw
            Write-Host "Session now matching global -> $($env:PYSHIM_INTERPRETER)"
        } else {
            Write-Host "No SPEC provided and no global python.env; using shim
fallbacks."
        }
    }
}

function Disable-PythonPersistence {
    <#
    .SYNOPSIS
        Make shim ignore python.env without deleting it.
    #>
    [CmdletBinding()]
    Param()
    $marker = 'C:\bin\shims\python.nopersist'
    if (-not (Test-Path $marker)) { New-Item -ItemType File -Path $marker | Out-
Null }
    Write-Host "Created $marker. Global persistence is now ignored."
}

function Enable-PythonPersistence {
    <#
    .SYNOPSIS
        Re-enable reading python.env.
    #>
    [CmdletBinding()]
    Param()
    $marker = 'C:\bin\shims\python.nopersist'
    if (Test-Path $marker) { Remove-Item -LiteralPath $marker -Force }
    Write-Host "Removed nopersist marker. Global persistence active again."
}

function Set-AppPython {
    <#
    .SYNOPSIS
```

```
        Pin an interpreter SPEC for a named app (used when PYSHIM_TARGET=App).
    .EXAMPLE
        Set-AppPython -App 'MyService' -Spec 'conda:svc'
    #>
    [CmdletBinding(SupportsShouldProcess=$true)]
    Param(
        [Parameter(Mandatory=$true)]
        [System.String]$App,

        [Parameter(Mandatory=$true)]
        [System.String]$Spec
    )
    $file = "C:\bin\shims\python@$App.env"
    Set-Content -LiteralPath $file -Value $Spec -NoNewline -Encoding ASCII
    Write-Host "Wrote $file => $Spec"
}

function Run-WithPython {
    <#
    .SYNOPSIS
        One-shot run with a specific interpreter, no persistence.
    .EXAMPLE
        Run-WithPython -Spec 'py:3.11' -- -m pip --version
    #>
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$true)]
        [System.String]$Spec,

        [Parameter(ValueFromRemainingArguments=$true)]
        [string[]]$Args
    )
    & "C:\bin\shims\python.bat" --interpreter "$Spec" -- @Args
}
```

.\bin\shims\python.bat

```
@echo off
REM Guard against recursion if PATH is reordered or shim calls itself
if "%PYSHIM_INVOKING%"=="1" exit /b 1
set "PYSHIM_INVOKING=1"

setlocal ENABLEDELAYEDEXPANSION
REM PYSHIM: central resolver for python on Windows (recursion-safe)

set "SHIMDIR=%~dp0"
set "GLOBAL_ENV=%SHIMDIR%python.env"
set "GLOBAL_NOPERSIST=%SHIMDIR%python.nopersist"

REM 0) One-shot flag: --interpreter "SPEC" --
set "ONESHOT_SPEC="
```

```
if /I "%~1"=="--interpreter" (
  if /I "%~3"=="--" (
    set "ONESHOT_SPEC=%~2"
    shift & shift & shift
  )
)

REM --- helpers -----------------------------------------------------
REM Resolve a SPEC (absolute path, conda:ENV, py:VER, plain)
set "RESOLVED_CMD="
call :RESOLVE_SPEC "%ONESHOT_SPEC%" RESOLVED_CMD
if defined RESOLVED_CMD goto :RUN

REM 1) Session override
if defined PYSHIM_INTERPRETER (
  call :RESOLVE_SPEC "%PYSHIM_INTERPRETER%" RESOLVED_CMD
  if defined RESOLVED_CMD goto :RUN
)

REM 2) App-target override
if defined PYSHIM_TARGET (
  set "TARGET_ENV=%SHIMDIR%python@%PYSHIM_TARGET%.env"
  if exist "%TARGET_ENV%" (
    for /f "usebackq delims=" %%P in ("%TARGET_ENV%") do set "SPEC=%%P"
    call :RESOLVE_SPEC "%SPEC%" RESOLVED_CMD
    if defined RESOLVED_CMD goto :RUN
  )
)

REM 3) .python-version (walk up)
call :FIND_DOTFILE ".python-version" PVFILE
if defined PVFILE (
  for /f "usebackq delims=" %%P in ("%PVFILE%") do set "SPEC=%%P"
  call :RESOLVE_SPEC "%SPEC%" RESOLVED_CMD
  if defined RESOLVED_CMD goto :RUN
)

REM 4) Global persistence (unless disabled)
if not exist "%GLOBAL_NOPERSIST%" if exist "%GLOBAL_ENV%" (
  for /f "usebackq delims=" %%P in ("%GLOBAL_ENV%") do set "SPEC=%%P"
  call :RESOLVE_SPEC "%SPEC%" RESOLVED_CMD
  if defined RESOLVED_CMD goto :RUN
)

REM 5) Fallback chain (recursion guards):
REM    - Prefer real py.exe if present and NOT coming from a py.bat shim
where py >NUL 2>&1
if %ERRORLEVEL%==0 if not defined PYSHIM_FROM_PY (
  set "RESOLVED_CMD=py -3.12"
  goto :RUN
)

where py >NUL 2>&1
if %ERRORLEVEL%==0 if not defined PYSHIM_FROM_PY (
```

```
  set "RESOLVED_CMD=py -3"
  goto :RUN
)

REM    - Try conda base if available
where conda >NUL 2>&1 && (set "RESOLVED_CMD=conda run -n base python" & goto :RUN)

REM    - Locate a real python.exe that is NOT this shim directory
for /f "usebackq delims=" %%P in (`where python.exe 2^>NUL`) do (
  REM skip any hit inside the shim folder
  echo "%%~dpP" | find /I "%SHIMDIR%" >NUL
  if errorlevel 1 (
    set "RESOLVED_CMD=%%P"
    goto :RUN
  )
)

REM    - Last resort: error out clearly
echo [pyshim] No real python.exe found on PATH and no usable launcher/conda. 1>&2
exit /b 9009

:RUN
%RESOLVED_CMD% %*
exit /b %ERRORLEVEL%

:RESOLVE_SPEC
REM %1 = SPEC (maybe empty), %2 = outvar
set "_spec=%~1"
if not defined _spec goto :eof

REM absolute path?
if exist "%_spec%" (
  set "%~2=%_spec%"
  goto :eof
)

REM conda:ENV
echo.%_spec%| findstr /b /c:"conda:" >NUL
if not errorlevel 1 (
  set "_env=%_spec:conda:=%"
  set "%~2=conda run -n %_env% python"
  goto :eof
)

REM py:VERSION
echo.%_spec%| findstr /b /c:"py:" >NUL
if not errorlevel 1 (
  set "_ver=%_spec:py:=%"
  set "%~2=py -%_ver%"
  goto :eof
)

REM plain token (treat as exe name): force .exe to avoid re-entering this .bat
if /I "%_spec%"=="python" set "_spec=python.exe"
```

```
  set "%~2=%_spec%"
  goto :eof

  :FIND_DOTFILE
  REM %1 = filename, %2 = outvar
  set "_fn=%~1"
  set "_here=%cd%"
  set "_visited_dirs="
  :WALKUP
  if exist "%_here%\%_fn%" (
    set "%~2=%_here%\%_fn%"
    goto :eof
  )
  REM Guard against junction/symlink cycles
  if defined _visited_dirs (
    echo.|set /p="|%_here%|" | findstr /I /C:"%_visited_dirs%" >nul && goto :eof
    set "_visited_dirs=%_visited_dirs%|%_here%"
  ) else (
    set "_visited_dirs=|%_here%"
  )
  REM Compute canonical parent using %%~f normalization
  for %%P in ("%_here%\..") do set "_parent=%%~fP"
  REM If parent equals current dir, we're at a root (drive or UNC); stop
  if /i "%_parent%"=="%_here%" goto :eof
  set "_here=%_parent%"
  goto :WALKUP
```

.\bin\shims\python.env

```
C:\Users\h8rt3rmin8r\miniconda3\envs\py312\python.exe
```

.\examples\python.env

```
py:3.12
```

.\examples\python@MyService.env

```
conda:myservice
```

.\bin\shims\pythonw.bat

```
@echo off
setlocal
```

```
REM headless interpreter (best-effort)
"%~dp0python.bat" %*
```

.\tests\smoke.ps1

```
$ErrorActionPreference = 'Stop'
$SepLine = "`n" + ("-" * 60) + "`n"
$TotalDurationMax = 12    # seconds
$SingleDurationMax = 3    # seconds
$StartTime = Get-Date
$TestsFailed = $false

#-------------------------------------------------------------------------------
# Helper function to run a command with timeout and exit code monitoring
#-------------------------------------------------------------------------------
function Invoke-TimedCommand {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$true)]
        [System.String]$Description,

        [Parameter(Mandatory=$true)]
        [ScriptBlock]$Command,

        [Parameter(Mandatory=$false)]
        [System.Int32]$TimeoutSeconds = $script:SingleDurationMax,

        [Parameter(Mandatory=$false)]
        [Switch]$IsGetCommand
    )

    Write-Host "  Running: $Description" -ForegroundColor Cyan
    $CommandStart = Get-Date
    $Job = Start-Job -ScriptBlock $Command

    $Completed = Wait-Job -Job $Job -Timeout $TimeoutSeconds
    $CommandEnd = Get-Date
    $Duration = ($CommandEnd - $CommandStart).TotalSeconds

    if ($null -eq $Completed) {
        Write-Host "    TIMEOUT after $Duration seconds (max: $TimeoutSeconds)" -
ForegroundColor Red
        Stop-Job -Job $Job -ErrorAction SilentlyContinue
        Remove-Job -Job $Job -Force -ErrorAction SilentlyContinue
        $script:TestsFailed = $true
        return $false
    }

    $JobOutput = Receive-Job -Job $Job -ErrorAction SilentlyContinue
    $JobState = $Job.State
    $JobError = $Job.ChildJobs[0].Error
```

```
    Remove-Job -Job $Job -Force -ErrorAction SilentlyContinue

    if ($JobState -eq 'Failed' -or ($JobError -and $JobError.Count -gt 0)) {
        Write-Host "    FAILED (duration: $Duration seconds)" -ForegroundColor Red
        if ($JobError) {
            $JobError | ForEach-Object { Write-Host "      ERROR: $_" -
ForegroundColor Red }
        }
        $script:TestsFailed = $true
        return $false
    }

    Write-Host "    OK (duration: $Duration seconds)" -ForegroundColor Green

    if ($JobOutput) {
        if ($IsGetCommand) {
            $JsonOutput = $JobOutput | Select-Object CommandType, Name, Version,
Source | ConvertTo-Json -Compress
            Write-Host "      $JsonOutput" -ForegroundColor Gray
        } else {
            $FlatOutput = ($JobOutput | Out-String).Trim() -replace "`r`n", ", " -
replace "`n", ", "
            Write-Host "      $FlatOutput" -ForegroundColor Gray
        }
    }
    return $true
}

#-------------------------------------------------------------------------------
Write-Host ''
Write-Host "Beginning smoke tests for pyshim." -ForegroundColor Green
#-------------------------------------------------------------------------------

$SepLine | Write-Host
Write-Host "Checking for py, python, and pip commands in PATH:" -ForegroundColor
Yellow

Invoke-TimedCommand -Description "where.exe py" -Command {
    where.exe py
    $LASTEXITCODE
} | Out-Null

Invoke-TimedCommand -Description "where.exe python" -Command {
    where.exe python
    $LASTEXITCODE
} | Out-Null

Invoke-TimedCommand -Description "where.exe pip" -Command {
    where.exe pip
    $LASTEXITCODE
} | Out-Null

#-------------------------------------------------------------------------------
```

```
$SepLine | Write-Host
Write-Host "Verifying that py, python, and pip commands are functional:" -
ForegroundColor Yellow

Invoke-TimedCommand -Description "Get-Command py" -IsGetCommand -Command {
    Get-Command py -ErrorAction Stop
} | Out-Null

Invoke-TimedCommand -Description "Get-Command python" -IsGetCommand -Command {
    Get-Command python -ErrorAction Stop
} | Out-Null

Invoke-TimedCommand -Description "Get-Command pip" -IsGetCommand -Command {
    Get-Command pip -ErrorAction Stop
} | Out-Null

#------------------------------------------------------------------------------
$SepLine | Write-Host
Write-Host "Checking versions of py, python, and pip:" -ForegroundColor Yellow

Invoke-TimedCommand -Description "py -V" -Command {
    py -V
    $LASTEXITCODE
} | Out-Null

Invoke-TimedCommand -Description "python -V" -Command {
    python -V
    $LASTEXITCODE
} | Out-Null

Invoke-TimedCommand -Description "pip --version" -Command {
    pip --version
    $LASTEXITCODE
} | Out-Null

#------------------------------------------------------------------------------
$SepLine | Write-Host
Write-Host "Running a simple Python command using the pyshim function Run-
WithPython:" -ForegroundColor Yellow

Invoke-TimedCommand -Description "Run-WithPython -Spec 'py:3' -- -c
`"print('ok')`"" -Command {
    Import-Module 'C:\bin\shims\pyshim.psm1' -Force
    Run-WithPython -Spec 'py:3' -- -c "print('ok')"
    $LASTEXITCODE
} | Out-Null

#------------------------------------------------------------------------------
$SepLine | Write-Host
Write-Host "Testing dotfile search from drive root (regression check for infinite
loop):" -ForegroundColor Yellow

Invoke-TimedCommand -Description "python -c `"print('ok from root')`" (from C:\)"
-Command {
```

```
    Push-Location C:\
    try {
        $output = & python -c "print('ok from root')" 2>&1
        $exitCode = $LASTEXITCODE
        Pop-Location
        if ($exitCode -ne 0) { throw "Exit code: $exitCode" }
        $exitCode
    } catch {
        Pop-Location
        throw
    }
} | Out-Null

#-------------------------------------------------------------------------------
$SepLine | Write-Host
Write-Host "Testing dotfile search from UNC path (if available):" -ForegroundColor
Yellow

$UncPath = "\\localhost\c$"
if (Test-Path -LiteralPath $UncPath -ErrorAction SilentlyContinue) {
    Invoke-TimedCommand -Description "python -c `"print('ok from UNC')`" (from
$UncPath)" -Command {
        Push-Location $UncPath
        try {
            $output = & python -c "print('ok from UNC')" 2>&1
            $exitCode = $LASTEXITCODE
            Pop-Location
            if ($exitCode -ne 0) { throw "Exit code: $exitCode" }
            $exitCode
        } catch {
            Pop-Location
            throw
        }
    } | Out-Null
} else {
    Write-Host "  Skipped (UNC path not accessible)" -ForegroundColor Gray
}

#-------------------------------------------------------------------------------
$SepLine | Write-Host
$EndTime = Get-Date
$TotalDuration = ($EndTime - $StartTime).TotalSeconds

if ($TestsFailed) {
    Write-Host "Total duration: $TotalDuration seconds." -ForegroundColor Red
    Write-Host "Smoke FAILED: One or more tests failed." -ForegroundColor Red
    Write-Host "Exiting script." -ForegroundColor Red
    exit 1
} elseif ($TotalDuration -gt $TotalDurationMax) {
    Write-Host "Total duration: $TotalDuration seconds." -ForegroundColor Red
    Write-Host "Smoke FAILED: Total duration exceeds maximum of $TotalDurationMax
seconds." -ForegroundColor Red
    Write-Host "Exiting script." -ForegroundColor Red
    exit 1
```

```
} else {
    Write-Host "Total duration: $TotalDuration seconds." -ForegroundColor Green
    Write-Host "Smoke OK (Tests passed successfully)." -ForegroundColor Green
    Write-Host "Exiting script." -ForegroundColor Green
    exit 0
}
```