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.

Releases & Single-File Installer

- Run `pwsh ./tools/New-PyshimInstaller.ps1` to regenerate `dist/Install-Pyshim.ps1` with the current shims embedded.
- Attach `dist/Install-Pyshim.ps1` to the GitHub release. End users can install by executing:

```
powershell.exe -ExecutionPolicy Bypass -File .\Install-Pyshim.ps1 -WritePath
```

- The installer mirrors `Make-Pyshim.ps1`: it copies the shims into `C:\bin\shims` and (optionally) appends that directory to the user PATH.
- The `Build Installer` GitHub workflow (`.github/workflows/build-installer.yml`) can be triggered manually or by publishing a release. It runs the generator script and, when invoked from a release, uploads the installer as a release asset automatically.

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

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\workflows\build-installer.yml

```yaml
name: Build Installer

on:
  workflow_dispatch:
  release:
    types: [published]

jobs:
  build:
    runs-on: windows-latest
    defaults:
```

```
      run:
        shell: pwsh
    steps:
      - name: Check out repository
        uses: actions/checkout@v4

      - name: Generate installer
        run: ./tools/New-PyshimInstaller.ps1 -Force -OutputPath ./dist/Install-
Pyshim.ps1

      - name: Upload installer artifact
        if: github.event_name == 'workflow_dispatch'
        uses: actions/upload-artifact@v4
        with:
          name: Install-Pyshim.ps1
          path: dist/Install-Pyshim.ps1

      - name: Attach installer to release
        if: github.event_name == 'release'
        uses: softprops/action-gh-release@v1
        with:
          files: dist/Install-Pyshim.ps1
```

.\.github\copilot-instructions.md

Project Snapshot

- pyshim routes Windows python/pip/pythonw calls via batch shims in `bin/shims` and a PowerShell module to pick the right interpreter.
- Core pieces: `python.bat` resolver, tiny wrappers (`pip.bat`, `pythonw.bat`), and `pyshim.psm1` cmdlets managing config files.

Resolution Chain (bin/shims/python.bat)

- Priority is fixed: one-shot `--interpreter` → session `PYSHIM_INTERPRETER` → app `python@%PYSHIM_TARGET%.env` → project `.python-version` up the tree → global `python.env` → fallback.
- Fallback executes `py -3.12`, then `py -3`, then `conda run -n base python`, then the first real `python.exe` outside the shim dir; never add bare `python` or we loop forever.
- `:RESOLVE_SPEC` parses specs (`py:3.12`, `conda:env`, absolute paths); `:FIND_DOTFILE` walks parents using delayed expansion.

Config Surfaces

- All `.env` and `.python-version` files are single-line ASCII with no trailing newline; editing scripts rely on `for /f "usebackq"`.
- `python.nopersist` toggles global persistence; guard `PYSHIM_FROM_PY` stops recursive launches when `py.exe` hands control back.

PowerShell Module (bin/shims/pyshim.psm1)

- Cmdlets (`Use-Python`, `Run-WithPython`, etc.) use `[CmdletBinding()]`, explicit `Param()` blocks, PascalCase variables, and write files with `Set-Content -NoNewline -Encoding ASCII -LiteralPath`.
- Always compute `$Sep = [IO.Path]::DirectorySeparatorChar` before building paths and prefer `Join-Path`; thanks again, Microsoft.
- Maintain self-awareness variables (`$thisFunctionReference` et al.) for logging when new helpers are introduced.

Testing & Verification

- Smoke test lives in `tests/smoke.ps1`; run `.\tests\smoke.ps1` from repo root when you touch the resolver or module.
- Manual sanity: `Use-Python -Spec 'py:3.12' -Persist` then `python -V`; use `Run-WithPython` for one-shot checks during debugging.

Development Habits

- Batch files assume delayed expansion and preserve `%ERRORLEVEL%`; never early-exit without `exit /b %ERRORLEVEL%`.
- Stick to four-space indents, no tabs, no stray whitespace, and keep helper functions alphabetized when you add new ones.
- New scripts inherit the comment-based help structure shown in the module; keep examples real by using existing commands.

Voice & Tone

- Write like a human who has seen things: short sentences, natural contractions, no boilerplate transitions.
- Sprinkle dry sarcasm at Microsoft tooling when it fits; avoid generic pep-talks or AI-scented phrasing.

Quick References

- Shim dir is hard-coded `C:\bin\shims`; keep paths literal and quote anything user-controlled.
- Keep the repo ASCII unless the pre-existing file already goes Unicode.
- Default Python on the box is 3.12.10; confirm interpreter specs align with that reality.
- If you spot untracked changes you didn't make, stop and ask before touching them.

When In Doubt

- Prefer tool-specific helpers (PowerShell cmdlets, batch subroutines) over inventing new workflows, and document any new CLI entrypoint.
- Ask for clarification if interpreter resolution, persistence flags, or path rules seem underspecified.

`.\dist\Install-Pyshim.ps1`

```
<#
.SYNOPSIS
    Single-file installer that provisions pyshim shims to the local machine.
.DESCRIPTION
```

```
    Unpacks an embedded archive containing the pyshim batch shims and PowerShell
    module, then mirrors the behaviour of Make-Pyshim.ps1: copies the payload to
    C:\bin\shims (creating the directory when needed) and optionally appends
    that directory to the user PATH.

    The embedded archive is generated from the repository's bin/shims directory
    using tools/New-PyshimInstaller.ps1. Re-run that tool whenever the shims
    change to refresh this installer before publishing a release asset.
.PARAMETER WritePath
    Automatically append C:\bin\shims to the user PATH when it is missing. If
    omitted the script prompts the user.
.PARAMETER Help
    Display the full help text for this script.
.EXAMPLE
    powershell.exe -ExecutionPolicy Bypass -File .\Install-Pyshim.ps1 -WritePath

    Installs pyshim and ensures the user PATH contains C:\bin\shims.
#>
[CmdletBinding(SupportsShouldProcess=$true,ConfirmImpact='None',DefaultParameterSe
tName='Default')]
Param(
    [Parameter(Mandatory=$false,ParameterSetName='Default')]
    [Alias('Path','P')]
    [Switch]$WritePath,

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

if ($Help -or ($PSCmdlet.ParameterSetName -eq 'HelpText')) {
    Get-Help -Name $MyInvocation.MyCommand.Path -Full
    exit 0
}

$EmbeddedArchive = @'
UEsDBBQAAAAIALlibFtfyrxgXgAAAGcAAAAHAAAAcGlwLmJhdHNITc7IV8hPS+PlKk4tyclPTszh
5Qpy9VVIzSsuLUpVKMgsUMhNLEnOSC1WKM9ILEktSy1SKKgsycjP00tKLFEoSi3OzylLTeHlUlKt
SykwQEgpKejmgrWravFyAQBQSwMEFAAAAAgAuWJsW8bajPXzBAAAQA8AAAsAAABweXNoaW0ucHNt
Mb1XbW/bNhD+HiD/4ZAYkw2EHrp9S+einqO0HmJHsLxmRRMMjHS2iFAkQVL2jC7/faBeLMkvWVoM
y4dYtnkvfO65587nENIVgk2YgQXjCAlqvITR5f0jE/cmYam5Vxv32lcmfXN6cg7jVEltGyYLLVPY
yExDJ5jdXo9vfGACArlGbRLk/PTk9GSRicgyKeB3gyTY2EQK+Hp6AgDwy3nx2g8/T2+DcBwWb93f
KJHSIFAoLZiwqJVGixoWUhc5GDTGeaYi/lFqUKgNMxaYhSWXj5TzTb8McOWHo9k4mI9vp3WMMPBH
QKMIlTVAH43kmUVQ1CYX4KnN5c/9Nz955aN3AVKDF0kR00t/+smrPAfD2XDiz/0ZhAqj2vm4kbBR
GO2fD4p0a5M7zSwWWVm5WwiHQh/FCrqZ4GgMCFndN6X6CTXgX8xY09sPNJV7oa6Qo8XjMSpknxCV
gQ6K1WXwOfw4nvw5ns79WTDLHUvBN9t4/h/DSXDj1zEa9SYOmxpTIK2EXmNaAG+l5MYDgPOq9sTl
UBievytev4zSmKP9lYmYiWW391B8HFBN024d40v+gatPd0JFTK3Um0FnQbnByiQ/Fm6MxbQfWs3E
8qHj0rlwvK4PrJmNkodOeafDX+7UoFcd6oQJS6+YhgF4zWp45dcfciL7YgUD+E0yQQJqk9rKq4tW
WWwjTQpavGi3JZFXJcQW0K199KpWrb6ao7Glr21qPfgKM0zlCsnYYgrkhlnUlO8cA3ItdYTwXHs8
RqwBdETm5KPVGuSjNBbOCodVt6OIEGJm6CPHOJeGRWYzjRBRzg10c53SeXZxr3+WJ4FLLTMRjySX
Gj4j53Jdh9JoMy2K988tVFzpW4C8kH5bDJrph2VrNRWNvCssztpgV5RqBT1QiZ2S7x2vU7ijWjCx
3IK4JyLMgNJoUNi3IKSFtWbWGdQ86zeSzDEC5AYPhQzRkpEUFoU9zolPlGdY3B7IVE5xzZlAIL6I
pOtfGIaj8fjYfQpIy9tjvNX9HNFtlN2M67fl4/4dXuB629lxDnx4zfVntEG9Y1QRcg0ptVHi8Ciu
mF+weyR4r3Hhw+VpRpnKYuYoLVcsxtiNUxCyClRX/i1kxmXgBAoWlPNHGj2ZJh8qOE9Pnluj/6ro
```

```
z1LTg0bn/vsmMKFPWIRkSyE1NhKCNbOJzCzEbpy51Jjtf8Mw6JWSWXJ/R4LvD2hkRQ3ieqNJkMJF
z0nhFNelDrr/841CuHYi1DoJf8NtZsk047xCrVmTkUbq+Fye7sMB0WNuB1iXqMSuDG3QffHdmM+Q
YG4NGmm80///C8D70L44ZMoA7QnTBLQwjfcE7yCyNLJshUCXlIl9XJ2sDZV69SobMLehtvQ+7zc3
qygImrqWUwq6mcEY1gkKKFt6Ppx98OeDoVLHt6x2Nu4RvMkmRL1iEXrt9cmsIu/F6oWZchu+CROZ
8TjQMkJjBh2rM/y2NappcWiLGirV3pO+y0s9ZCui5dN+AGcHaPbeBXUELvXq+HTKnXzTYGoy7U5L
i1Um9Vxvc2iWCXLHbPJqEt0KJCaRFnQmctkDmv+sYAsWNal14ZS7weWjvNnJoLWhv/GAECApKKaA
kJVzJ8V/uGXbVxT2GD/yulxrmc4wpcxtM0O9zFIUdoepuZ3JPX55eOgM9dK0yPLDQZr0H6k9A0Ka
7XpWFNGB8r5w83x68g9QSwMEFAAAAAgAuWJsWxZzY0cABgAA+A8AAAoAAABweXRob24uYmF0vVdt
b9s2EP5uwP/hIICZNcRJnG4f6kFds0RNjTq24Tjpir24jHSKudCkStJOPAT57QNJyZbitE0HbPni
mLzjPffcw+P5NSYzCTLLmo1xfAanC6pSoNeUCW1AYbJQmkkBLIPR0eQtMA0KpUpRYQpSgZ6xOSSU
cw3MaORZs8EyCMjow/nb3tm0N7gcvusNTkkQRUEnALxjBvavoNNsaDQQPDKLOkGz4ba4TCiHeHD0
Sz8+iftHH+KT+NfR0eC8Nxx4pN61CwkKoygHhVryJSrIpIJ8ZWZSgBTwnolU3mporXNpa5phWMSB
wJ5y0htH5CHND4Ji8bQ//OWoP40HlxEpDIg/cw/F8pHVYDiKx+e988mWrZA5Ks20cWlZ1AchDAW2
9UwayDi97kK7zYRBlSs0qCA4H8XHAbTbRYjhID5/O5xM7XIUOHL3exCQh45ltOYbQKvZAFhbvPAW
xTLA9nnk4TAo9mYsM7BT/7RbYbMRltjb7TbMkNuU7P//5s8fNPalAgoWBrTolZZ8YRByama7kEiR
0m48uNyFfNW9jMe7kHPKRFhwMo7Ph/3L+GR6fHZiObHyg26x6jKDgFQTJQFUfRyLKWZMYFrbgGtp
JHTHF4My5U4I56jdDZBLVIqlWPNe63cSj0fjeBKPPdtPQto23gLmyvc1aOuCHIZwlOdtQ9U1mi8C
nByNT+OJx+Y49AtPCfx1idObkELwDhneMW0gIBtvspaXvXb7GQQLjVc0ufkEKXI211EAhIyACWjV
/UJIpcfitUhGhRaf5O7JMlrrZ/BVV/GLEPZ8pu2lvZxSQOuW8htY5GGppTe9wcn0ZDh50+vHEDwy
D2B0aTfqPLslz8VzmPD2n2XhWzj4FsX8EMIpl1eUQ9GYUCQIrYXgqDWkTNMrjmnoUhPSrAv+uNOR
oCqHTbcs5fAcCqpe/zMNP4bwhnJuoUEyoxbP5qm7tk+gDrveFADaMFKYoQKFlrjVHt6hDZcr1CgM
UJHCYDiBRM6ZuIZMyTlQa3dFjXsgm43bGSqEfAWvBhd9OHy103EUk3g8Ho778WXcJ1F0AAXrjy7v
m/HwbDr6ULm9tQ6Yr6D9Yq/jW/njbP/zwE9GXRM3USvfzuGKakcaXVLGrcpKbH57DQ92dqC1Hctb
qYWAtvBn+UsZwM4mej12XybU2EemKJt/ve8QzIwaO8jYmpkZ036ISZnCxEi1aja+rt6PJa/rQw//
tCl8dEJ2fFkc+oblQMUKZswAE5qlNjr6gJnkKSpr6kawgNgJZBTAPWRMpP4RL1tz4Agqu7BSUnFc
IodO7XGvMbZpqJ9thZ4m6gY9LZXp+qNBLgwkHKniq2bDgfstX1nMf8BAbtGZyYVI7ajlBkR7GYSE
heskwOlCJDNU+65+e9B5tXPYbJRT4MuDg5cWi8dGqvAJkO83hlW5eodKS/C5kA5ExTgxp6srBJzn
ZhXuAjmEyGa0pKqYH6Y6xySyI9S6z5XCd1sFYSizkqjagPKz8yp7n/Mo2547njwcRuXy5nK409bM
r0ccz+9eYe9Lr42ySe8n3cDbldUvm/JW/X1aKJZF4K73izyANaraJSLWnpTX6HNA/QDmp+4vIs1X
z4W5RFXCzFdbGG1TIdbm8+S5YRCMvEEBLaOQGqAarBQFnWPYta9PguDvugS6lCwFhW20s7Lt0O7S
2+68maeLekVRUPaVilQ2Yi8n/3qNa3qpDQ8VbWaMo8X3pCIzUejRf7XNJSJJag8vSGOaGUynKVPa
zrzd90f9dxejuhStG/mdTDPxhCCrm9vMbv/2+2shEsOk2NerOWfiBpJVwlHXpp4aLB/RieText3P
o+C+iHtftjWnmB7sH3cDUvO2LU4suG3/FWRrzVTTf+RYxgiajRCQa6yJrepYtWy6lI/lPLf3OqFC
CmZ/cuZU2Td9oa1QCHmwWlZzytnf1LLhH4fNEFPQurdXGWGm/ozIetsebAP1svJk/LSgXEOyUO5r
ytQu3OJ3CsHqGJSUBlqpYku0v68vBsfhT6CNzL1Wma2zP8mptcyoSlpVQ2vbUqWlbv4BUEsDBBQA
AAAIALlibFv1p0aWOQAAADcAAAAKAAAAcHl0aG9uLmVudnO2igktTi0qjsmwKCoxLsrNzLMoisnN
zMtMzs9LSTSOSc0rK44pqDQ2NIopqCzJyM/TS61I5eUCAFBLAwQUAAAACAC5YmxbOFZA31UAAABT
AAAACwAAAHB5dGhvbncuYmF0c0hNzshXyE9L4+UqTi3JyU9OzOHlCnL1VchITUzJSS0uVsjMK0kt
KihKLUktUtBISi0u0U1NS8svKtHk5VJSrUspMCioLMnIz9NLSixRUlDV4uUCAFBLAQIUABQAAAAI
ALlibFtfyrxgXgAAAGcAAAAHAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
W8bajPXzBAAAQA8AAAsAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
WxZzY0cABgAA+A8AAAoAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
9adGljkAAAA3AAAACgAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
VkDfVQAAAFMAAAALAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
DAAAAAA=
'@

Add-Type -AssemblyName System.IO.Compression.FileSystem

function Add-PyshimPathEntry {
    Param(
        [Parameter(Mandatory=$true)]
        [System.String]$TargetPath,
```

```
        [Parameter(Mandatory=$true)]
        [System.String]$CurrentUserPath
    )

    $SplitPaths = @()
    if ($CurrentUserPath) {
        $SplitPaths = $CurrentUserPath -split ';'
    }

    if (-not ($SplitPaths | Where-Object { $_.TrimEnd('\\') -ieq
$TargetPath.TrimEnd('\\') })) {
        $SplitPaths = @($SplitPaths | Where-Object { $_ }) + $TargetPath
    }

    return ($SplitPaths | Where-Object { $_ }) -join ';'
}

function Get-PyshimPathScopes {
    Param()

    return [PSCustomObject]@{
        Process = $env:Path
        User    = [Environment]::GetEnvironmentVariable('Path','User')
        Machine = [Environment]::GetEnvironmentVariable('Path','Machine')
    }
}

function Test-PyshimPathPresence {
    Param(
        [Parameter(Mandatory=$true)]
        [System.String]$TargetPath,

        [Parameter(Mandatory=$true)]
        [System.String[]]$Scopes
    )

    foreach ($Scope in $Scopes) {
        if (-not $Scope) {
            continue
        }

        $Entries = $Scope -split ';'
        if ($Entries | Where-Object { $_.TrimEnd('\\') -ieq
$TargetPath.TrimEnd('\\') }) {
            return $true
        }
    }

    return $false
}

function Expand-PyshimArchive {
    Param(
        [Parameter(Mandatory=$true)]
```

```
        [System.String]$DestinationPath
    )

    $Bytes = [Convert]::FromBase64String($EmbeddedArchive)
    $ZipPath = [IO.Path]::GetTempFileName()
    try {
        [IO.File]::WriteAllBytes($ZipPath,$Bytes)

[IO.Compression.ZipFile]::ExtractToDirectory($ZipPath,$DestinationPath,$true)
    } finally {
        if (Test-Path -LiteralPath $ZipPath) {
            Remove-Item -LiteralPath $ZipPath -Force -ErrorAction SilentlyContinue
        }
    }
}

$ShimDir = 'C:\bin\shims'
$WorkingRoot = Join-Path -Path ([IO.Path]::GetTempPath()) -ChildPath ("pyshim_" +
[Guid]::NewGuid().ToString('N'))
$Null = New-Item -ItemType Directory -Path $WorkingRoot -Force

try {
    Expand-PyshimArchive -DestinationPath $WorkingRoot
    $PayloadSource = $WorkingRoot

    if (-not (Test-Path -LiteralPath $ShimDir)) {
        if ($PSCmdlet.ShouldProcess($ShimDir,'Create shim directory')) {
            New-Item -ItemType Directory -Path $ShimDir -Force | Out-Null
        }
    }

    if ($PSCmdlet.ShouldProcess($ShimDir,'Copy embedded shims')) {
        Copy-Item -Path (Join-Path -Path $PayloadSource -ChildPath '*') -
Destination $ShimDir -Recurse -Force
    }
} finally {
    if (Test-Path -LiteralPath $WorkingRoot) {
        Remove-Item -LiteralPath $WorkingRoot -Recurse -Force -ErrorAction
SilentlyContinue
    }
}

$PathScopes = Get-PyshimPathScopes
$AllScopes = @($PathScopes.Process,$PathScopes.User,$PathScopes.Machine)
$PathPresent = Test-PyshimPathPresence -TargetPath $ShimDir -Scopes $AllScopes

if ($PathPresent) {
    Write-Host "C:\bin\shims already present in PATH." -ForegroundColor Green
    exit 0
}

$ShouldWritePath = $false
if ($WritePath) {
    $ShouldWritePath = $true
```

```
} else {
    $Response = Read-Host "Add 'C:\bin\shims' to your user PATH? [y/N]"
    if ($Response -and ($Response.Trim() -match '^(y|yes)$')) {
        $ShouldWritePath = $true
    }
}

if ($ShouldWritePath) {
    if ($PSCmdlet.ShouldProcess('User PATH','Append shim directory')) {
        $NewUserPath = Add-PyshimPathEntry -TargetPath $ShimDir -CurrentUserPath
$PathScopes.User
        [Environment]::SetEnvironmentVariable('Path',$NewUserPath,'User')
        $EnvEntries = $env:Path -split ';'
        if (-not ($EnvEntries | Where-Object { $_.TrimEnd('\\') -ieq
$ShimDir.TrimEnd('\\') })) {
            $env:Path = ($EnvEntries + $ShimDir | Where-Object { $_ }) -join ';'
        }
        Write-Host "Added 'C:\bin\shims' to the user PATH. Restart existing
shells." -ForegroundColor Green
    }
    exit 0
} else {
    Write-Host "Skipped PATH update. To add it later run:" -ForegroundColor Yellow
    Write-Host "    [Environment]::SetEnvironmentVariable('Path',( '{0};' +
[Environment]::GetEnvironmentVariable('Path','User')).Trim(';'),'User')" -f
$ShimDir
    exit 0
}
```

.\tools\Install-Pyshim.template.ps1

```
<#
.SYNOPSIS
    Single-file installer that provisions pyshim shims to the local machine.
.DESCRIPTION
    Unpacks an embedded archive containing the pyshim batch shims and PowerShell
    module, then mirrors the behaviour of Make-Pyshim.ps1: copies the payload to
    C:\bin\shims (creating the directory when needed) and optionally appends
    that directory to the user PATH.

    The embedded archive is generated from the repository's bin/shims directory
    using tools/New-PyshimInstaller.ps1. Re-run that tool whenever the shims
    change to refresh this installer before publishing a release asset.
.PARAMETER WritePath
    Automatically append C:\bin\shims to the user PATH when it is missing. If
    omitted the script prompts the user.
.PARAMETER Help
    Display the full help text for this script.
.EXAMPLE
    powershell.exe -ExecutionPolicy Bypass -File .\Install-Pyshim.ps1 -WritePath
```

```
    Installs pyshim and ensures the user PATH contains C:\bin\shims.
#>
[CmdletBinding(SupportsShouldProcess=$true,ConfirmImpact='None',DefaultParameterSe
tName='Default')]
Param(
    [Parameter(Mandatory=$false,ParameterSetName='Default')]
    [Alias('Path','P')]
    [Switch]$WritePath,

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

if ($Help -or ($PSCmdlet.ParameterSetName -eq 'HelpText')) {
    Get-Help -Name $MyInvocation.MyCommand.Path -Full
    exit 0
}

$EmbeddedArchive = @'
__PYSHIM_EMBEDDED_ARCHIVE__
'@

Add-Type -AssemblyName System.IO.Compression.FileSystem

function Add-PyshimPathEntry {
    Param(
        [Parameter(Mandatory=$true)]
        [System.String]$TargetPath,

        [Parameter(Mandatory=$true)]
        [System.String]$CurrentUserPath
    )

    $SplitPaths = @()
    if ($CurrentUserPath) {
        $SplitPaths = $CurrentUserPath -split ';'
    }

    if (-not ($SplitPaths | Where-Object { $_.TrimEnd('\\') -ieq
$TargetPath.TrimEnd('\\') })) {
        $SplitPaths = @($SplitPaths | Where-Object { $_ }) + $TargetPath
    }

    return ($SplitPaths | Where-Object { $_ }) -join ';'
}

function Get-PyshimPathScopes {
    Param()

    return [PSCustomObject]@{
        Process = $env:Path
        User    = [Environment]::GetEnvironmentVariable('Path','User')
        Machine = [Environment]::GetEnvironmentVariable('Path','Machine')
```

```
    }
}

function Test-PyshimPathPresence {
    Param(
        [Parameter(Mandatory=$true)]
        [System.String]$TargetPath,

        [Parameter(Mandatory=$true)]
        [System.String[]]$Scopes
    )

    foreach ($Scope in $Scopes) {
        if (-not $Scope) {
            continue
        }

        $Entries = $Scope -split ';'
        if ($Entries | Where-Object { $_.TrimEnd('\\') -ieq
$TargetPath.TrimEnd('\\') }) {
            return $true
        }
    }

    return $false
}

function Expand-PyshimArchive {
    Param(
        [Parameter(Mandatory=$true)]
        [System.String]$DestinationPath
    )

    $Bytes = [Convert]::FromBase64String($EmbeddedArchive)
    $ZipPath = [IO.Path]::GetTempFileName()
    try {
        [IO.File]::WriteAllBytes($ZipPath,$Bytes)

[IO.Compression.ZipFile]::ExtractToDirectory($ZipPath,$DestinationPath,$true)
    } finally {
        if (Test-Path -LiteralPath $ZipPath) {
            Remove-Item -LiteralPath $ZipPath -Force -ErrorAction SilentlyContinue
        }
    }
}

$ShimDir = 'C:\bin\shims'
$WorkingRoot = Join-Path -Path ([IO.Path]::GetTempPath()) -ChildPath ("pyshim_" +
[Guid]::NewGuid().ToString('N'))
$Null = New-Item -ItemType Directory -Path $WorkingRoot -Force

try {
    Expand-PyshimArchive -DestinationPath $WorkingRoot
    $PayloadSource = $WorkingRoot
```

```
    if (-not (Test-Path -LiteralPath $ShimDir)) {
        if ($PSCmdlet.ShouldProcess($ShimDir,'Create shim directory')) {
            New-Item -ItemType Directory -Path $ShimDir -Force | Out-Null
        }
    }

    if ($PSCmdlet.ShouldProcess($ShimDir,'Copy embedded shims')) {
        Copy-Item -Path (Join-Path -Path $PayloadSource -ChildPath '*') -
Destination $ShimDir -Recurse -Force
    }
} finally {
    if (Test-Path -LiteralPath $WorkingRoot) {
        Remove-Item -LiteralPath $WorkingRoot -Recurse -Force -ErrorAction
SilentlyContinue
    }
}

$PathScopes = Get-PyshimPathScopes
$AllScopes = @($PathScopes.Process,$PathScopes.User,$PathScopes.Machine)
$PathPresent = Test-PyshimPathPresence -TargetPath $ShimDir -Scopes $AllScopes

if ($PathPresent) {
    Write-Host "C:\bin\shims already present in PATH." -ForegroundColor Green
    exit 0
}

$ShouldWritePath = $false
if ($WritePath) {
    $ShouldWritePath = $true
} else {
    $Response = Read-Host "Add 'C:\bin\shims' to your user PATH? [y/N]"
    if ($Response -and ($Response.Trim() -match '^(y|yes)$')) {
        $ShouldWritePath = $true
    }
}

if ($ShouldWritePath) {
    if ($PSCmdlet.ShouldProcess('User PATH','Append shim directory')) {
        $NewUserPath = Add-PyshimPathEntry -TargetPath $ShimDir -CurrentUserPath
$PathScopes.User
        [Environment]::SetEnvironmentVariable('Path',$NewUserPath,'User')
        $EnvEntries = $env:Path -split ';'
        if (-not ($EnvEntries | Where-Object { $_.TrimEnd('\\') -ieq
$ShimDir.TrimEnd('\\') })) {
            $env:Path = ($EnvEntries + $ShimDir | Where-Object { $_ }) -join ';'
        }
        Write-Host "Added 'C:\bin\shims' to the user PATH. Restart existing
shells." -ForegroundColor Green
    }
    exit 0
} else {
    Write-Host "Skipped PATH update. To add it later run:" -ForegroundColor Yellow
    Write-Host "    [Environment]::SetEnvironmentVariable('Path',( '{0};' +
```

```
[Environment]::GetEnvironmentVariable('Path','User')).Trim(';'),'User')" -f
$ShimDir
    exit 0
}
```

.\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

# Purge any existing prompt artifacts before generating new output
$CleanupPattern = 'Prompt-Code-Reference*'
$ExistingArtifacts = Get-ChildItem -LiteralPath $PSScriptRoot -Filter
$CleanupPattern -File -ErrorAction SilentlyContinue
if ($ExistingArtifacts) {
    Write-Host "  Removing $($ExistingArtifacts.Count) existing prompt
artifact(s)..." -ForegroundColor Yellow
    foreach ($Artifact in $ExistingArtifacts) {
        Remove-Item -LiteralPath $Artifact.FullName -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

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\Make-Pyshim.ps1

```
<#
.SYNOPSIS
    Install the pyshim shims and optionally wire them into the user PATH.
.DESCRIPTION
    Copies all files from the repository shim folder (`bin/shims`) into the
    fixed installation directory (`C:\bin\shims`). Existing files are
    overwritten. If the target directory does not exist it is created.

    After copying, the script validates whether `C:\bin\shims` already lives
    in the effective PATH (process, user, or machine scopes). If the entry is
    missing you can either supply `-WritePath` up front or respond to the
```

```
    interactive prompt to append it to the user PATH. Refusing the update
    prints the manual command you need to run yourself.
.PARAMETER WritePath
    Automatically append `C:\bin\shims` to the user PATH when it is missing.
    Without this switch the script will prompt for confirmation.
.PARAMETER Help
    Display the full help text for this script.
.EXAMPLE
    .\Make-Pyshim.ps1 -WritePath

    Copies the shims into place and appends `C:\bin\shims` to the user PATH if
    it is not already present.
.#>
[CmdletBinding(SupportsShouldProcess=$true,ConfirmImpact='None',DefaultParameterSe
tName='Default')]
Param(
    [Parameter(Mandatory=$false,ParameterSetName='Default')]
    [Alias("Path","P")]
    [Switch]$WritePath,

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

# Catch Help Text Requests
if (($Help) -or ($PSCmdlet.ParameterSetName -eq 'HelpText')) {
    Get-Help -Name $MyInvocation.MyCommand.Path -Full
    Exit 0
}

# Internal self-awareness variables for verbosity/logging
$thisFunctionReference = "{0}" -f $MyInvocation.MyCommand
$thisScript = $thisFunctionReference

function Add-PyshimPathEntry {
    Param(
        [Parameter(Mandatory=$true)]
        [System.String]$TargetPath,

        [Parameter(Mandatory=$true)]
        [System.String]$CurrentUserPath
    )

    $SplitPaths = @()
    if ($CurrentUserPath) {
        $SplitPaths = $CurrentUserPath -split ';'
    }

    if (-not ($SplitPaths | Where-Object { $_.TrimEnd('\') -ieq
$TargetPath.TrimEnd('\') })) {
        $SplitPaths = @($SplitPaths | Where-Object { $_ }) + $TargetPath
    }
```

```
    return ($SplitPaths | Where-Object { $_ }) -join ';'
}

function Get-PyshimPathScopes {
    Param()

    return [PSCustomObject]@{
        Process = $env:Path
        User    = [Environment]::GetEnvironmentVariable('Path','User')
        Machine = [Environment]::GetEnvironmentVariable('Path','Machine')
    }
}

function Test-PyshimPathPresence {
    Param(
        [Parameter(Mandatory=$true)]
        [System.String]$TargetPath,

        [Parameter(Mandatory=$true)]
        [System.String[]]$Scopes
    )

    foreach ($Scope in $Scopes) {
        if (-not $Scope) {
            continue
        }

        $Entries = $Scope -split ';'
        if ($Entries | Where-Object { $_.TrimEnd('\') -ieq
$TargetPath.TrimEnd('\') }) {
            return $true
        }
    }

    return $false
}

$ShimDir = 'C:\bin\shims'
$SourceDir = Join-Path -Path $PSScriptRoot -ChildPath 'shims'

if (-not (Test-Path -LiteralPath $SourceDir)) {
    throw "Shim source directory '$SourceDir' was not found."
}

if (-not (Test-Path -LiteralPath $ShimDir)) {
    if ($PSCmdlet.ShouldProcess($ShimDir,'Create shim directory')) {
        New-Item -ItemType Directory -Path $ShimDir -Force | Out-Null
    }
}

$CopySource = Join-Path -Path $SourceDir -ChildPath '*'
if ($PSCmdlet.ShouldProcess($ShimDir,'Copy shim files')) {
    Copy-Item -Path $CopySource -Destination $ShimDir -Recurse -Force
    Write-Verbose "[$thisScript] Copied shims from '$SourceDir' to '$ShimDir'."
```

```
}

$PathScopes = Get-PyshimPathScopes
$AllScopes = @($PathScopes.Process,$PathScopes.User,$PathScopes.Machine)
$PathPresent = Test-PyshimPathPresence -TargetPath $ShimDir -Scopes $AllScopes

if ($PathPresent) {
    Write-Verbose "[$thisScript] Shim directory already present in PATH."
    return
}

$ShouldWritePath = $false
if ($WritePath) {
    $ShouldWritePath = $true
} else {
    $Response = Read-Host "Add '$ShimDir' to your user PATH? [y/N]"
    if ($Response -and ($Response.Trim() -match '^(y|yes)$')) {
        $ShouldWritePath = $true
    }
}

if ($ShouldWritePath) {
    if ($PSCmdlet.ShouldProcess('User PATH','Append shim directory')) {
        $NewUserPath = Add-PyshimPathEntry -TargetPath $ShimDir -CurrentUserPath
$PathScopes.User
        [Environment]::SetEnvironmentVariable('Path',$NewUserPath,'User')
        $EnvEntries = $env:Path -split ';'
        if (-not ($EnvEntries | Where-Object { $_.TrimEnd('\') -ieq
$ShimDir.TrimEnd('\') })) {
            $env:Path = ($EnvEntries + $ShimDir | Where-Object { $_ }) -join ';'
        }
        Write-Host "Added '$ShimDir' to the user PATH. Restart shells that were
already open."
    }
} else {
    Write-Host "Skipping PATH update. To add it later run:`n
[Environment]::SetEnvironmentVariable('Path',( '{0};' +
[Environment]::GetEnvironmentVariable('Path','User')).Trim(';'),'User')" -f
$ShimDir
}
```

.\tools\New-PyshimInstaller.ps1

```
<#
.SYNOPSIS
    Generates the single-file Install-Pyshim.ps1 installer for release builds.
.DESCRIPTION
    Zips the repository's bin/shims directory, converts it to Base64, and injects
    the payload into tools/Install-Pyshim.template.ps1. The rendered installer is
    written to dist/Install-Pyshim.ps1 (or a custom destination when specified).
.PARAMETER OutputPath
```

```
    Optional destination for the generated installer. Defaults to dist/Install-
Pyshim.ps1
    relative to the repository root.
.PARAMETER Force
    Overwrite the output file if it already exists.
.EXAMPLE
    pwsh ./tools/New-PyshimInstaller.ps1

    Writes dist/Install-Pyshim.ps1 with the current shims payload embedded.
.EXAMPLE
    pwsh ./tools/New-PyshimInstaller.ps1 -OutputPath ./Install-Pyshim.ps1 -Force

    Generates the installer at the repository root, overwriting any existing file.
#>
[CmdletBinding()]
Param(
    [Parameter(Mandatory=$false)]
    [System.String]$OutputPath,

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

$RepoRoot = Resolve-Path -Path (Join-Path -Path $PSScriptRoot -ChildPath '..')
$SourceDir = Join-Path -Path $RepoRoot -ChildPath 'bin/shims'
$TemplatePath = Join-Path -Path $PSScriptRoot -ChildPath 'Install-
Pyshim.template.ps1'

if (-not $OutputPath) {
    $OutputDir = Join-Path -Path $RepoRoot -ChildPath 'dist'
    if (-not (Test-Path -LiteralPath $OutputDir)) {
        New-Item -ItemType Directory -Path $OutputDir -Force | Out-Null
    }
    $OutputPath = Join-Path -Path $OutputDir -ChildPath 'Install-Pyshim.ps1'
} else {
    $OutputPath = Resolve-Path -Path $OutputPath -ErrorAction SilentlyContinue -
OutVariable resolved | Out-Null
    if ($resolved) {
        $OutputPath = $resolved.ProviderPath
    } else {
        $OutputPath = [IO.Path]::GetFullPath((Join-Path -Path (Get-Location) -
ChildPath $OutputPath))
    }
}

if (-not (Test-Path -LiteralPath $SourceDir)) {
    throw "Shim source directory '$SourceDir' was not found."
}

if (-not (Test-Path -LiteralPath $TemplatePath)) {
    throw "Template file '$TemplatePath' was not found."
}

if ((Test-Path -LiteralPath $OutputPath) -and (-not $Force)) {
```

```
    throw "Output file '$OutputPath' already exists. Use -Force to overwrite."
}

$TempRoot = New-Item -ItemType Directory -Path (Join-Path
([IO.Path]::GetTempPath()) ("pyshim_dist_" + [Guid]::NewGuid().ToString('N'))) -
Force
$TempZip = Join-Path -Path $TempRoot -ChildPath 'payload.zip'

try {
    Compress-Archive -Path (Join-Path -Path $SourceDir -ChildPath '*') -
DestinationPath $TempZip -Force
    $Bytes = [IO.File]::ReadAllBytes($TempZip)
    $Base64 = [Convert]::ToBase64String($Bytes)
    $Builder = New-Object System.Text.StringBuilder
    for ($Offset = 0; $Offset -lt $Base64.Length; $Offset += 76) {
        $ChunkLength = [Math]::Min(76, $Base64.Length - $Offset)
        [void]$Builder.AppendLine($Base64.Substring($Offset,$ChunkLength))
    }
    $WrappedBase64 = $Builder.ToString().TrimEnd()
    Write-Verbose ("Payload size: {0} bytes ({1} base64 characters)" -f
$Bytes.Length,$Base64.Length)

    $TemplateContent = Get-Content -LiteralPath $TemplatePath -Raw
    $Rendered = $TemplateContent -replace
'__PYSHIM_EMBEDDED_ARCHIVE__',$WrappedBase64
    if ($Rendered -notlike '*__PYSHIM_EMBEDDED_ARCHIVE__*') {
        Write-Verbose "Embedded archive inserted."
    } else {
        throw 'Failed to embed archive payload.'
    }

    Set-Content -LiteralPath $OutputPath -Value $Rendered -Encoding ASCII -Force
    Write-Host "Installer written to $OutputPath" -ForegroundColor Green
} finally {
    if (Test-Path -LiteralPath $TempRoot) {
        Remove-Item -LiteralPath $TempRoot -Recurse -Force -ErrorAction
SilentlyContinue
    }
}
```

.\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
```

.\examples\python.env

```
py:3.12
```

.\bin\shims\python.env

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

.\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
```

```powershell
$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 "    Result: OK (duration: $Duration seconds)" -ForegroundColor
Green

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