Use scripts in skills.

Skills can instruct agents to execute shell commands and package reusable scripts in the scripts/ directory.

How to run commands and package executable scripts within your skill.

Skills can instruct agents to run shell commands and package reusable scripts in the **scripts/** directory . This instruction includes one-time commands, independent scripts with their own dependencies, and how to design script interfaces for agent use.

One-time order

When an existing package already does what you need, you can directly reference it in your SKILL.md documentation without needing a scripts/ directory. Many ecosystems provide tools to automatically resolve these dependencies at runtime.

uvx runs Python packages in an isolated environment with robust caching capabilities. It's built into uv.

```
uvx ruff@0.8.0 check . uvx black@24.10.0 .
```

1. Not built into Python - requires separate installation.
2. Fast. The powerful cache means that repeated runs are almost instantaneous.

Tips for one-time commands in skills:

1. Pin the version (e.g., npx eslint@9.0.0) so that the command works consistently over time.
2. Clearly state the prerequisites in your SKILL.md (e.g., 'Node.js 18+ required') instead of assuming the agent environment already has them. For runtime-level requirements, use a compatibility field at the beginning of the information.
3. Move complex commands into scripts. Disposable commands work well when you call a tool with a few flags. When a command becomes complex enough to be difficult to execute correctly the first time, a tested script in scripts/ will be more reliable.

Referencing scripts from SKILL.md

Use relative paths from the skill's root directory to reference the packaged files. The agent will automatically resolve these paths—absolute paths are not necessary.

List the available scripts in your SKILL.md file so the agent knows they exist:

```
## Available scripts - **`scripts/validate.sh`** — Validates configuration files
```

Next, instruct the agent to run them:

```
## Workflow 1. Run the validation script: ```bash bash scripts/validate.sh "$INPU
```

The same relative path convention also works in supporting files like references/*.md - the script execution path (in code blocks) is relative to the skill's root directory, as the agent runs commands from there.

Standalone scripts

When you need reusable logic, package a script in scripts/ and declare its dependencies directly. The agent can run the script with a single command—no separate manifest file or installation step required.

Some languages ??support declaring dependencies directly:

PEP 723 defines a standard format for direct script metadata. Declaring dependencies in a TOML block inside # ///:

```
# /// script # dependencies = [ # "beautifulsoup4", # ] # /// from bs4 import Bea
```

Welcome

This is a test.

```
' print(BeautifulSoup(html, "html.parser").select_one("p.info").get_text())
```

Run with UV (recommended):

```
uv run scripts/extract.py
```

The `uv run` command creates an isolated environment, installs the declared dependencies, and runs the script. `pipx` (`pipx run scripts/extract.py`) also supports PEP 723.

1. Attach the version to the PEP 508 specifier: "beautifulsoup4>=4.12,5".
2. Use `requires-python` to restrict the Python version.
3. Use `uv lock --script` to create a lock file to ensure full reproducibility.

Design a script to use with the agent.

When an agent runs your script, it reads `stdout` and `stderr` to decide on the next action. Several design choices make it significantly easier for agents to use scripts.

Avoid interactive prompts.

This is a mandatory requirement of the agent execution environment. Agents operate in non-interactive shells—they cannot respond to TTY prompts, password dialogs, or confirmation menus. A script blocked by interactive input will hang indefinitely.

Accept all input via command-line flags, environment variables, or `stdin`:

```
# Bad: hangs waiting for input $ python scripts/deploy.py Target environment: _ #
```

Instructions for use using the --help command.

--help is the primary way for the agent to understand the script's interface. It includes a brief description, available flags, and usage examples:

```
Usage: scripts/process.py [OPTIONS] INPUT_FILE Process input data and produce a s
```

Keep it concise – the output will be displayed in the agent's context window along with everything else it's processing.

Write helpful error messages.

When the agent encounters an error, the message directly shapes its next attempt. A confusing message like 'Error: invalid input' wastes a processing turn. Instead, specify what went wrong, what is expected, and what needs to be tried:

```
Error: --format must be one of: json, csv, table. Received: "xml"
```

Use structured output format

Prioritize structured formats—JSON, CSV, TSV—over arbitrary text. Structured formats can be used by both agents and standard tools (jq, cut, awk), allowing your script to be integrated into data processing workflows.

```
# Whitespace-aligned — hard to parse programmatically NAME STATUS CREATED my-serv
```

Separating data from diagnostic information : Send structured data to stdout and progress messages, alerts, and other diagnostic information to stderr. This allows the agent to collect clean, analyzable output while still having access to diagnostic information when needed.

Other considerations

1. **Immutability** . The agent can retry commands. "Create if it doesn't exist" is safer than "create and report a duplicate".
2. **Input constraints** . Reject ambiguous input with clear errors instead of guesswork. Use enumerations and closed sets whenever possible.
3. **Support for testing** . For destructive or stateful operations, the --dry-run flag allows the agent to preview what will happen.
4. **Exit codes have meaning** . Use separate exit codes for different types of errors (not found, invalid argument, validation error) and include them in your --help output so the agent knows what each code means.
5. **Safe default values** . Consider whether destructive operations should require explicit confirmation flags (--confirm, --force) or other safeguards appropriate to the level of risk.
6. **Output size is predictable** . Many monitoring tools automatically truncate the tool's output when it exceeds a certain threshold (e.g., 10,000 - 30,000 characters), which can lead to the loss of important information. If your script can produce large output, default to a summary or a reasonable limit, and support flags like --offset so the agent can request more information when needed. Or, if the output is large and cannot be paginated, instruct agents to pass the --output flag to specify the output file or -- to explicitly choose to use stdout.

You finished reading the article "**Use scripts in skills.**" edited by the **TipsMake** team. We hope this article has provided you with many useful tech tips and tricks. You can search for similar articles on tips and guides. Thank you for reading and for following us regularly.

© 2019 TipsMake.com