UV

uv is a tool to install Python packages and manage projects. It replaces `pip` with a much faster tool.

Installation

On MacOS and Linux, you can install `uv` with the following command:

```
$ curl -LsSf https://astral.sh/uv/install.sh | sh
```

On Windows, you can install `uv` with the following command (might need to run in Admin mode - right click on powershell, run in admin):

```
> powershell -c "irm https://astral.sh/uv/install.ps1 | iex"
```

Alternatively, you can use `pip` to install `uv`:

```
$ pip install uv
```

Updating

When installed with the standalone installer (`curl`), you can ask `uv` to update itself:

```
$ uv self update
```

When installed with `pip`, you can update `uv` with the following command:

```
$ pip install --upgrade uv
```

Shell Completion

You can get auto completion for `uv` by running the following command:

```
# Determine your shell (e.g., with `echo $SHELL`), then run one of:
echo 'eval "$(uv generate-shell-completion bash)"' >> ~/.bashrc
echo 'eval "$(uv generate-shell-completion zsh)"' >> ~/.zshrc
echo 'uv generate-shell-completion fish | source' >> ~/.config/fish/config.fish
echo 'eval (uv generate-shell-completion elvish | slurp)' >> ~/.elvish/rc.elv
```

For Windows, you can use the following command:

```
Add-Content -Path $PROFILE -Value '(& uv generate-shell-completion powershell) | Out-String | Invoke-Expression'
```

Uninstallation

You can uninstall `uv` with the following command:

```
$ rm ~/.cargo/bin/uv ~/.cargo/bin/uvx
```

Getting Help

```
% uv help
An extremely fast Python package manager.

Usage: uv [OPTIONS] <COMMAND>

Commands:
  run                      Run a command or script
  init                     Create a new project
  add                      Add dependencies to the project
  remove                   Remove dependencies from the project
  sync                     Update the project's environment
  lock                     Update the project's lockfile
  export                   Export the project's lockfile to an alternate format
  tree                     Display the project's dependency tree
  tool                     Run and install commands provided by Python packages
  python                   Manage Python versions and installations
  pip                      Manage Python packages with a pip-compatible interface
  venv                     Create a virtual environment
  cache                    Manage uv's cache
  version                  Display uv's version
  generate-shell-completion  Generate shell completion
  help                     Display documentation for a command
```

UV Concepts

- Project - A directory containing a `pyproject.toml` file.
  - Application - A project with a `hello.py` script. Think Flask or Django or command-line tools. You can *package* an application into a distributable application.
  - Library - A project that has a `src` directory with Python files. Think requests or pandas.
- Environment - A virtual environment, typically located in the `.venv` directory.
- Workspace - A directory containing multiple projects.
- Tools - Python packages that provide commands. For example, `black` provides the `black` command. It is possible to install as an executable and not as a library.

UV prefers working at the project level. The environment level is more like a 'low level' detail.

Python Installation

`uv` is meant to be able to bootstrap itself. It does not depend on any Python installation. However, it can manage Python installations for you. This also makes it easy to try out different Python versions.

```
% uv python list
cpython-3.12.5-macos-aarch64-none    /opt/homebrew/opt/python@3.12/bin/python3.12 -> ../Frameworks/Python.framework/Versions/3.12/bin/
python3.12
cpython-3.12.5-macos-aarch64-none    <download available>
cpython-3.12.1-macos-aarch64-none    /opt/miniconda3/bin/python3.12
cpython-3.12.1-macos-aarch64-none    /opt/miniconda3/bin/python3 -> python3.12
cpython-3.12.1-macos-aarch64-none    /opt/miniconda3/bin/python -> python3.12
cpython-3.11.9-macos-aarch64-none    /opt/homebrew/opt/python@3.11/bin/python3.11 -> ../Frameworks/Python.framework/Versions/3.11/bin/
python3.11
cpython-3.11.9-macos-aarch64-none    <download available>
cpython-3.10.14-macos-aarch64-none   /opt/homebrew/opt/python@3.10/bin/python3.10 -> ../Frameworks/Python.framework/Versions/3.10/bin/
```

```
python3.10
cpython-3.10.14-macos-aarch64-none    <download available>
cpython-3.9.19-macos-aarch64-none     <download available>
cpython-3.9.6-macos-aarch64-none      /Library/Developer/CommandLineTools/usr/bin/python3 -> ../../Library/Frameworks/Python3.framework/
Versions/3.9/bin/python3
cpython-3.8.19-macos-aarch64-none     <download available>
```

Installing Python 3.8.19:

```
% uv python install cpython-3.8
Searching for Python versions matching: cpython-3.8.19-macos-aarch64-none
Installed Python 3.8.19 in 1.32s
 + cpython-3.8.19-macos-aarch64-none
```

Running Python 3.8.19:

```
% uv run --python 3.8 python
Python 3.8.19 (default, Aug 14 2024, 04:42:21)
[Clang 18.1.8 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Virtual Environments and Pip

This is the *low-level* interface to manage Python packages and virtual environments. Generally, you should create *projects* instead.

```
# reqsmall.txt
pandas
notebook
xgboost
matplotlib
```

Create a virtual environment with a specific Python version:

```
% uv venv --python cpython-3.8 ~/.envs/uvenv38
% time uv pip install -r /tmp/reqsmall.txt --python ~/.envs/uvenv38/bin/python
3.86s user 2.41s system 132% cpu 4.725 total
```

vs (slow `pip`)

```
% time pip install -r /tmp/reqsmall.txt
15.73s user 4.10s system 76% cpu 26.061 total
```

From the `uv` documentation (https://docs.astral.sh/uv/pip/environments/#discovery-of-python-environments)

> When running a command that mutates an environment such as `uv pip sync` or `uv pip install`, `uv` will search for a virtual environment in the following order:

- An activated virtual environment based on the `VIRTUAL_ENV` environment variable.
- An activated Conda environment based on the `CONDA_PREFIX` environment variable.
- A virtual environment at `.venv` in the current directory, or in the nearest parent directory.
- If no virtual environment is found, `uv` will prompt the user to create one in the current directory via uv venv.

Editable Packages

Run:

```
% uv pip install -e .
```

Probably want to run:

```
% uv pip install -e .[dev]
```

Projects

Create a library project. A libray project has a `src` directory with Python files.

```
% uv init --lib mylib
Initialized project `mylib` at `/private/tmp/mylib`
% tree mylib
mylib
├── README.md
├── pyproject.toml
└── src
    └── mylib
        └── __init__.py
```

Create an application project. An application project has a `hello.py` script.

```
% uv init myapp
Initialized project `myapp` at `/private/tmp/myapp`
% tree myapp
myapp
├── README.md
├── hello.py
└── pyproject.toml
```

Create a *packaged* application.

```
% uv init --package mypkgapp
```

A packaged application is configured to have a command line function in `pyproject.toml`

```
[project-script]
hello = "mypkgapp:hello"
```

Add Dependencies

To add dependencies to a project, use the `uv add` command.

```
% uv add pandas
```

This will update both the `pyproject.toml` file and the `uv.lock` file.

- `pyproject.toml` - The project's configuration file. Specifies broad dependencies.
- `uv.lock` - The project's lockfile. Specifies exact dependencies. Don't edit this file. Should work on other machines.

When you run `uv run` or `uv sync`, `uv` will check to make sure the dependencies are installed.

Options for `add`:

- `--dev` - Add a development dependency.
- `-r <file>` - Add dependencies from a file.
- `--optional` - Add an optional dependency.
- `--editable` - Add an editable dependency.
- `--script` - Add dependencies to a script.

Generally, `uv` will create a virtual environment in the `.venv` directory.

Running Commands

- You can activate the virtual environment with `source .venv/bin/activate`. Then run `python script.py`.
- You can run a command with `uv run script.py`.

Running Jupyter with Pandas 2.2

create `req-pd2.2.txt`:

```
pandas==2.2.0
```

Run:

```
% uv run --with notebook --with-requirements req-pd2.2.txt jupyter lab
```

To run with a specific Python version:

```
% uv run --python 3.8 --with notebook --with-requirements req-pd2.2.txt jupyter lab
  × No solution found when resolving `--with` dependencies:
  ╰─▸ Because the current Python version (3.8.19) does not satisfy Python>=3.9 and pandas==2.2.0 depends on Python>=3.9, we can conclude
that pandas==2.2.0
      cannot be used.
      And because you require pandas==2.2, we can conclude that your requirements are unsatisfiable.
```

Inside of Jupyter, I can install new packages with `!uv install <package>`.

To run the old version of Jupyter:

```
% uv run --python 3.9 --with notebook --with-requirements ~/Dropbox/computer/pyreq/req-pd2.2.txt --with "notebook<7" jupyter notebook
```

`pyproject.toml`

`pyproject.toml` is a standardized configuration file for Python projects that consolidates build requirements, package metadata, and tool configurations into a TOML-formatted file. Introduced by PEP 518 and expanded upon by subsequent PEPs, it allows developers to specify build dependencies and settings for various tools like linters, formatters, and testing frameworks. This centralization simplifies project setup and maintenance by providing a unified place for configurations, enabling tools like `pip` (or `uv`) to create isolated build environments and streamlining the packaging and distribution process.

See https://packaging.python.org/en/latest/guides/writing-pyproject-toml/ for more information.

The default `uv` configuration for a library project is:

```
[project]
name = "mylib"
version = "0.1.0"
description = "Add your description here"
readme = "README.md"
requires-python = ">=3.8"
dependencies = []

[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"
```

After running `uv add pandas`, the `pyproject.toml` file will be updated to:

```
[project]
name = "mylib"
version = "0.1.0"
description = "Add your description here"
readme = "README.md"
requires-python = ">=3.8"
dependencies = [
    "pandas>=2.0.3",
]

[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"
```

The `uv.lock` file will also be updated.

You can install optional dependencies with:

```
%  uv add catboost --optional boost
```

This will add catboost as an optional dependency in `pyproject.toml`:

```
[project]
name = "mylib"
version = "0.1.0"
description = "Add your description here"
readme = "README.md"
requires-python = ">=3.8"
dependencies = [
    "pandas>=2.0.3",
]

[project.optional-dependencies]
boost = [
    "catboost>=1.2.5",
]

[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"
```

To add a development dependency, use the `--dev` flag:

```
% uv add pytest --dev
```

```
[project]
name = "mylib"
version = "0.1.0"
description = "Add your description here"
readme = "README.md"
requires-python = ">=3.8"
dependencies = [
    "pandas>=2.0.3",
]

[project.optional-dependencies]
boost = [
    "catboost>=1.2.5",
]

[build-system]
requires = ["hatchling"]
```

```
build-backend = "hatchling.build"

[tool.uv]
dev-dependencies = [
    "pytest>=8.3.2",
]
```

Tools

`uv` considers a *tool* to be a Python package that provides a command. For example, `black` provides the `black` command.

You can run a tool with `uv tool`:

```
uv tool run black
```

`uv` also provides a convenient way to run tools with `uvx`:

```
uvx black
```

To install a tool, use the `uv tool install` command:

```
uv tool install black
```

This will install the tool in a different location (`~/.local/share/uv/tools` or run `uv tool dir`) than the project's virtual environment. The documentation recommends using `uvx` to run tools rather than installing them.