Programming

- Python
  - Working with json
  - uv
- Git
  - Activating an SSH Key
  - Order Of Operations
  - General commands
- Docker
  - General Commands
  - Building a first setup
- SQL
  - General commands

Python

Python

Working with json

Writing json to file

```
import json
with open('data.json', 'w', encoding='utf-8') as f:
    json.dump(data, f, ensure_ascii=False, indent=4)
```

Python

uv

uv: Python Environment & Package Manager

A single Rust binary that replaces `pip`, `venv`, `pyenv`, `pipx`, and `poetry`. It guarantees the right Python version and packages are present before running your code, so you stop managing virtual environments by hand.

Use it when: you want edit-and-run Python scripts (scrapers, automation, data jobs) without venv setup, manual activation, or a compile step.

Mental Model

uv sits between you and Python. Three modes, each storing its environment differently:

Mode	When to use	Where the env lives
Inline script (PEP 723)	One `.py` file, ad-hoc scripts	Ephemeral, in the global cache
Project	Multi-file app, reproducible builds	Visible `.venv/` in the project folder
Tool install	Installing a CLI globally (e.g. `ruff`)	Isolated, like `pipx`

For a scraper / automation script, use **inline script** mode.

Install

```
# Linux / macOS (recommended, no existing Python needed)
curl -LsSf https://astral.sh/uv/install.sh | sh
```

```
# Windows (PowerShell)
powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/install.ps1 | iex"
```

Installs the `uv` binary to:

- Linux / macOS: `~/.local/bin`
- Windows: `%USERPROFILE%\.local\bin`

> Remember this path. cron and other minimal-environment runners do not include it in `PATH`.

Alternatives: `pipx install uv`, `pip install uv`, `brew install uv`. Update later with `uv self update`.

Inline Scripts (PEP 723)

Declare dependencies in a comment header at the top of the file. No `requirements.txt`, no project, no manual venv.

```
# scraper.py
# /// script
# requires-python = ">=3.12"
# dependencies = ["httpx", "polars", "PyMySQL"]
# ///

import httpx, polars as pl, pymysql
# your code here
```

Run it:

```
uv run scraper.py
```

First run resolves, downloads, and builds a cached environment, then executes. Subsequent runs reuse the cache and start almost instantly. uv rebuilds the env only when the dependency list or Python version changes.

Add a dependency without hand-editing the header:

```
uv add --script scraper.py requests
```

Make the file directly executable:

```
#!/usr/bin/env -S uv run --script
```

```
chmod +x scraper.py
./scraper.py
```

Where the Cached venv Lives

Inline-script environments live in uv's **global cache**, not next to the file.

OS	Default cache path
Linux	`$XDG_CACHE_HOME/uv` or `~/.cache/uv`
macOS	`~/Library/Caches/uv`
Windows	`%LOCALAPPDATA%\uv\cache`

Find it on any machine:

```
uv cache dir
```

- Folder names are hashes of the Python version, dependency versions, and script name. Not human-readable.
- The cache is **deduplicated**: each package version is downloaded and built once, then shared across all scripts and projects on the machine.
- Override the location with the `UV_CACHE_DIR` environment variable.

Housekeeping:

```
uv cache clean    # wipe entire cache
uv cache prune    # remove stale entries only
du -sh "$(uv cache dir)"    # check size (Linux/macOS)
```

For **project** mode, the env is instead a normal `.venv/` in the project root, which you can inspect or delete directly.

Reproducibility (Pin Versions)

Unpinned dependencies will silently upgrade when a new release drops. For anything unattended, lock the versions:

```
uv lock --script scraper.py   # creates scraper.py.lock
```

Once locked, `uv run`, `uv add --script`, and `uv tree --script` reuse the pinned versions.

For a collection of related scripts you want to rebuild on another machine, use a **project** instead: a folder with `pyproject.toml` + `uv.lock`, then `uv sync` on the new machine reproduces the exact Python version and dependencies.

Running in Cron

It works. Two things bite people, both about the environment, not uv.

1. PATH

cron uses a stripped-down `PATH` (typically `/usr/bin:/bin`) that excludes `~/.local/bin`, so a bare `uv` fails with "command not found". Use the absolute path (find it with `which uv`).

2. HOME / cache

uv needs `HOME` set to locate `~/.cache/uv`. User crontabs usually set this correctly. System crontabs, containers, and service accounts may not, so set it explicitly.

```
# crontab -e
HOME=/home/youruser
UV_CACHE_DIR=/home/youruser/.cache/uv

*/15 * * * * /home/youruser/.local/bin/uv run /home/youruser/scrapers/scraper.py >>
/home/youruser/logs/scraper.log 2>&1
```

Always redirect stdout and stderr to a log file, or failures vanish silently.

Cron checklist

- [ ] Pin dependencies (`uv lock --script`) so a 3am package update cannot break the job.
- [ ] Warm the cache once by hand; the first run does the slow install work.
- [ ] Test the exact command in a minimal shell before scheduling:

```
env -i HOME=/home/youruser /home/youruser/.local/bin/uv run scraper.py
```

Command Cheat Sheet

Command	Does
`uv run script.py`	Run a script (project or inline)
`uv add --script script.py pkg`	Add a dependency to an inline script
`uv lock --script script.py`	Pin versions for an inline script
`uv init`	Create a new project
`uv add pkg`	Add a dependency to a project
`uv sync`	Rebuild a project env from the lockfile
`uv python install 3.12`	Install a specific Python version
`uv tool install ruff`	Install a CLI tool globally (pipx-style)
`uv cache dir`	Show the cache location
`uv cache clean`	Wipe the cache
`uv self update`	Update uv itself

Gotchas

- **MariaDB driver:** avoid the official `mariadb` PyPI package; it builds a C extension and needs MariaDB Connector/C installed system-wide. Use **PyMySQL** (pure Python, no compile) or **SQLAlchemy + PyMySQL** instead.
- **Unpinned scripts upgrade silently.** Lock anything that must not break unattended.
- **cron PATH/HOME** as above.
- **Cache and Python env should be on the same filesystem** for fast hard-linking; otherwise uv falls back to slow copies.

Sources

- uv official docs: https://docs.astral.sh/uv (storage, cache, running scripts, CLI reference)
- PEP 723 (inline script metadata): https://peps.python.org/pep-0723/
- `uv-cache` man page (platform cache paths)

Last reviewed: June 2026. Verify install URLs and cache behavior against the current docs if uv has had a major version bump.

Git

Git

Activating an SSH Key

```
eval "$(ssh-agent -s)"
ssh-add ~/.ssh/server
```

Git

Order Of Operations

Local > remote

1. Add the changed files
2. commit the files
3. push to repo

```
git add FILE_NAME
git commit -m "message"
git push origin main
```

Remote -> local

1. Pull changes to local folder

```
git pull origin main
```

Git

General commands

Force pull

```
git fetch origin
git reset --hard origin/main
```

Stash changes

1. Stash your changes

`git stash push -m "My local changes"`

- `push` → saves your uncommitted changes to the stash stack.
- `-m "message"` → lets you label the stash so you know what it is later.

Now your working directory is clean.

---

2. Pull the latest code (optional but recommended)

`git pull origin main`

- `origin` → the name of your remote (default is usually `origin`).
- `main` → replace with your actual branch (could be `master`, `develop`, etc.).
- This ensures you're up to date before pushing.

---

3. Push your branch

`git push origin main`

- Sends your local commits to the remote repository.

---

4. Re-apply your stashed changes (if you want them back)

`git stash pop`

- This re-applies the last stashed set of changes and removes it from the stash list.
- If you want to keep the stash around, use `git stash apply` instead of `pop`.

Docker

General Commands

Containers

List running containers

```
docker ps
```

List all containers

```
docker ps -a
```

Run a container from an image

```
docker run [OPTIONS] imagename[:tag]
```

Options:

- --rm (remove after close)
- -p 8000:8000 (bind docker port 8000 to computer port 8000)
- -d (run in background)
- --env-file .env (adds env file to container run)

Restart container

```
docker start container-name
```

Stop a container

```
docker stop container-name
```

Remove a container

```
docker rm myubuntu
```

Images

List images

```
docker images
```

Remove image

```
docker rmi IMAGE_ID
```

Build image

```
sudo docker build -t image_name .
```

Building a first setup

Bulding a container

```
sudo docker build -t CONTAINER_NAME .
```

`-t` is the container "tag" name
`.` assumes the "Dockerfile" is in the same directory as the PWD

Running and opening a container

```
docker run -it --rm CONTAINER_NAME bash
```

`-it` interactive terminal like bash
`--rm` removes the container after exiting

SQL

SQL

General commands

Logging into remote db

```
mysql -h HOST -P PORT -u USERNAME -p
```

Creating a table

Syntax

```
CREATE TABLE table_name (
    column1 datatype contraint,
    column2 datatype,
    column3 datatype,
);
```

Example

```
CREATE TABLE products (
    id INT AUTO_INCREMENT PRIMARY KEY,
    product_name VARCHAR(32),
    company_id INT,
    FOREIGN KEY (company_id) REFERENCES companies(id)
) ENGINE=InnoDB;
```

View table creation code

```
SHOW CREATE TABLE `table_name`;
```

Adding a new foreign key

```
ALTER TABLE table_name ADD CONSTRAINT fk_foreign_key_name FOREIGN KEY (column_name) REFERENCES table_name(table_column);
```