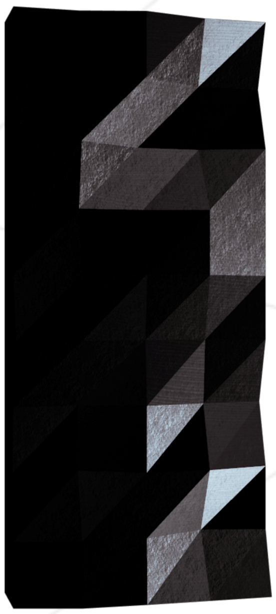

The Matrix

Welcome to the Real World of Data Engineering

Summary: You've taken the red pill. Now it's time to learn how to architect data systems in the real world. Master virtual environments, package management, and environment configuration to build your first data pipeline.

Version: 2.0

Contents

Chapter I

Foreword

```
"This is your last chance.  After this, there is no going back.  You
take the blue pill; the story ends, you wake up in your bed and
believe whatever you want to believe.  You take the red pill; you
stay in Wonderland, and I show you how deep the rabbit hole goes." -
Morpheus
```

Welcome to the real world. You've discovered that what you thought was reality was just a simulation. Now, as a **Data Architect** in Zion, you must learn to build robust data systems that can withstand the machines' attacks.

In this activity, you'll discover the fundamental tools that every data engineer uses to create **isolated environments**, manage **program dependencies**, and configure **system variables**. These aren't just technical skills—they're survival tools in the war against the machines.

```
Remember:  There is no spoon.  But there are virtual environments,
and they're very real.
```

Chapter II

AI Instructions

● Context

During your learning journey, AI can assist with many different tasks. Take the time to explore the various capabilities of AI tools and how they can support your work. However, always approach them with caution and critically assess the results. Whether it's code, documentation, ideas, or technical explanations, you can never be completely sure that your question was well-formed or that the generated content is accurate. Your peers are a valuable resource to help you avoid mistakes and blind spots.

● Main message

- ☛ Use AI to reduce repetitive or tedious tasks.
- ☛ Develop prompting skills — both coding and non-coding — that will benefit your future career.
- ☛ Learn how AI systems work to better anticipate and avoid common risks, biases, and ethical issues.
- ☛ Continue building both technical and power skills by working with your peers.
- ☛ Only use AI-generated content that you fully understand and can take responsibility for.

● Learner rules:

- You should take the time to explore AI tools and understand how they work, so you can use them ethically and reduce potential biases.
- You should reflect on your problem before prompting — this helps you write clearer, more detailed, and more relevant prompts using accurate vocabulary.
- You should develop the habit of systematically checking, reviewing, questioning, and testing anything generated by AI.
- You should always seek peer review — don't rely solely on your own validation.

● Phase outcomes:

- Develop both general-purpose and domain-specific prompting skills.
- Boost your productivity with effective use of AI tools.
- Continue strengthening computational thinking, problem-solving, adaptability, and collaboration.

● Comments and examples:

- You'll regularly encounter situations — exams, evaluations, and more — where you must demonstrate real understanding. Be prepared, keep building both your technical and interpersonal skills.
- Explaining your reasoning and debating with peers often reveals gaps in your understanding. Make peer learning a priority.
- AI tools often lack your specific context and tend to provide generic responses. Your peers, who share your environment, can offer more relevant and accurate insights.
- Where AI tends to generate the most likely answer, your peers can provide alternative perspectives and valuable nuance. Rely on them as a quality checkpoint.

✓ Good practice:

I ask AI: "How do I test a sorting function?" It gives me a few ideas. I try them out and review the results with a peer. We refine the approach together.

✗ Bad practice:

I ask AI to write a whole function, copy-paste it into my project. During peer-evaluation, I can't explain what it does or why. I lose credibility — and I fail my project.

✓ Good practice:

I use AI to help design a parser. Then I walk through the logic with a peer. We catch two bugs and rewrite it together — better, cleaner, and fully understood.

✗ Bad practice:

I let Copilot generate my code for a key part of my project. It compiles, but I can't explain how it handles pipes. During the evaluation, I fail to justify and I fail my project.

Chapter III

General Instructions

III.1 Python Requirements

- Your project must be written in **Python 3.10 or later**.
- Your code must respect the flake8 linter standards.
- All functions and methods must include type hints.
- Your functions should handle exceptions gracefully using `try-except` blocks.
- Follow Python naming conventions (snake_case for variables and functions).
- Include clear comments explaining your logic, especially for environment detection.
- Test your programs in different environments (with/without virtual env, with/without dependencies).

III.2 Project Structure

- Each exercise must be completed in its own directory: `ex0/`, `ex1/`, `ex2/`
- Your code must demonstrate understanding of the concepts, not just copy-paste solutions
- All exercises build upon each other; complete them in order
- Document your learning process through clear code comments and meaningful variable names
- Submit your work to your assigned Git repository

Chapter IV

Exercise 0: Entering the Matrix

	Exercise0
construct	
Directory: *ex*0/	
Files to Submit: `construct.py`	
Authorized: `sys, os, site modules, print()`	

The Construct

`"What is real? How do you define 'real'?" - Morpheus`

Before anyone could learn to bend the rules, they had to understand the construct; a safe, isolated environment where they could train without affecting the real world. In data engineering, we call this a virtual environment.

Your first mission is to create a training program that demonstrates your understanding of Python virtual environments. The machines have corrupted the global Python installation, and you need to create a clean, isolated space to work.

Mission Briefing

Create a program called `construct.py` that:

- Detects whether it is running inside a virtual environment
- Displays information about the current Python environment
- Provides instructions for creating and activating a virtual environment if none is detected
- Shows the difference between global and virtual environment package locations

Your program should work both inside and outside virtual environments, providing different outputs for each scenario.

Usage Examples

Here are some examples of how to test your program:

Testing outside virtual environment

```
$> python3 construct.py
# Should detect no virtual environment and provide instructions
```

Creating and testing virtual environment

```
$> python3 -m venv matrix_env
$> source matrix_env/bin/activate
(matrix_env) $> python3 construct.py
# Should detect virtual environment and show details
```

Expected Output

When run outside a virtual environment:

Outside the Matrix

```
$> python construct.py

MATRIX STATUS: You're still plugged in

Current Python: /usr/bin/python3.11
Virtual Environment: None detected

WARNING: You're in the global environment!
The machines can see everything you install.

To enter the construct, run:
python -m venv matrix_env
source matrix_env/bin/activate # On Unix
matrix_env
Scripts
activate    # On Windows

Then run this program again.
```

When run inside a virtual environment:
Inside the Construct

```
$> python construct.py

MATRIX STATUS: Welcome to the construct

Current Python: /path/to/matrix_env/bin/python
Virtual Environment: matrix_env
Environment Path: /path/to/matrix_env

SUCCESS: You're in an isolated environment!
Safe to install packages without affecting
the global system.

Package installation path:
/path/to/matrix_env/lib/python3.11/site-packages
```

Chapter V

Exercise 01: Loading Programs

	Exercise01
loading	
Directory: *ex*01/	
Files to Submit: `loading.py`, `requirements.txt`, `pyproject.toml`	
Authorized: `pandas`, `requests`, `matplotlib`, `numpy`, `sys`, `importlib`	

Loading Programs

`"I need guns. Lots of guns." - Neo`

Just as the resistance fighters needed weapons loaded into their minds, data engineers need packages loaded into their environments. But unlike the Matrix's instant downloads, we need to manage dependencies carefully to avoid conflicts.

Your mission is to create a program that demonstrates mastery of package management using both pip and Poetry. You'll build a data analysis tool that requires external libraries.

Mission Briefing

Create a data analysis program called `loading.py` that:

- Uses pandas for data manipulation
- Uses numpy for numerical computations
- Uses matplotlib for visualization
- Demonstrates the difference between pip and Poetry dependency management
- Includes proper dependency files for both approaches

Your program should analyze "Matrix data" (you can simulate this with sample data) and generate a simple visualization.

The requests library is authorized for optional use if you want to fetch real data from an API, but it's not required. You can work with simulated/sample data instead.

Requirements

- Create both `requirements.txt` (for pip) and `pyproject.toml` (for Poetry)
- Your program must handle missing dependencies gracefully
- Include a comparison function that shows installed package versions
- Show the differences between pip and Poetry through your program's output

Your program should detect which packages are available and provide helpful error messages if dependencies are missing.

Usage Examples

Testing without dependencies

```
$> python3 loading.py
# Should show missing dependencies and installation instructions
```

Installing with pip

```
$> pip install -r requirements.txt
$ python3 loading.py
# Should run analysis and create visualization
```

Installing with Poetry

```
$> poetry install
$ poetry run python loading.py
# Should run analysis with Poetry environment
```

Expected Behavior

Loading Programs

```
$> python loading.py

LOADING STATUS: Loading programs...

Checking dependencies:
[OK] pandas (2.1.0) - Data manipulation ready
[OK] requests (2.31.0) - Network access ready
[OK] matplotlib (3.7.2) - Visualization ready

Analyzing Matrix data...
Processing 1000 data points...
Generating visualization...

Analysis complete!
Results saved to: matrix\_analysis.png}
```

Chapter VI

Exercise 02: Accessing the Mainframe

	Exercise02
oracle	
Directory: *ex*02/	
Files to Submit: `oracle.py, .env.example, .gitignore`	
Authorized: `os, sys, python-dotenv modules, file operations`	

The Oracle

`"Know thyself." - The Oracle`

The Oracle knows everything because she has access to the mainframe's configuration. In the real world, applications need to access sensitive information like database credentials, API keys, and system settings without hardcoding them.

Your final mission is to create a secure configuration system using environment variables and `.env` files. You'll build a data pipeline that connects to external systems safely.

You should use the python-dotenv library to load environment variables from .env files. The goal is to learn how to use .env files for configuration management, not to implement a custom parser.

Mission Briefing

Create a program called `oracle.py` that:

- Loads configuration from environment variables
- Uses a `.env` file for development settings
- Demonstrates different configuration for development/production
- Includes proper error handling for missing configuration
- Shows how to keep secrets secure

Configuration Requirements

Your program should handle these configuration variables:

- `MATRIX_MODE` - "development" or "production"
- `DATABASE_URL` - Connection string for data storage
- `API_KEY` - Secret key for external services
- `LOG_LEVEL` - Logging verbosity
- `ZION_ENDPOINT` - URL for the resistance network

Never commit real secrets to version control! Your .env file should be in .gitignore.

Usage Examples

Testing without configuration

```
$> python3 oracle.py
# Should show default/missing configuration warnings
```

Using .env file

```
$> cp .env.example .env
$> # Edit .env with your values
$> python3 oracle.py
# Should load configuration from .env file
```

Environment variable override

```
$> MATRIX_MODE=production API_KEY=secret123 python3 oracle.py
# Should use environment variables over .env file
```

Expected Output

Accessing the Mainframe

```
$> python oracle.py

ORACLE STATUS: Reading the Matrix...

Configuration loaded:
Mode: development
Database: Connected to local instance
API Access: Authenticated
Log Level: DEBUG
Zion Network: Online

Environment security check:
[OK] No hardcoded secrets detected
[OK] .env file properly configured
[OK] Production overrides available

The Oracle sees all configurations.
```

Chapter VII

Submission and Peer-Review

 "There is a difference between knowing the path and walking the path." - Morpheus

Submit your assignment in your Git repository as usual. Only the work inside your repository will be reviewed during the peer-review. Don't hesitate to double check the names of your files to ensure they are correct.

During peer-review, you'll demonstrate:

- Your understanding of virtual environments and why they're important
- The differences between pip and Poetry for dependency management
- How environment variables keep applications secure and configurable
- Your ability to explain these concepts to other learners

Remember: The goal isn't just to make the code work; it's to understand the **why** behind these tools. In the real world of data engineering, these skills will keep your systems secure, maintainable, and scalable.

"Welcome to the real world." - Morpheus