

PROJECT COURSE

Database Regeneration - Practical Scripting

6 Chapters

Philip Osztromok

Generated with Claude

Contents

Chapter 1 Paths & File System Navigation 2

Chapter 2 Configuration & Environment Variables 11

Chapter 3 Database Connections 19

Chapter 4 Building db_utils.py 26

Chapter 5 Building rebuild_typescript_course.py 32

Chapter 6 Building Your Own Utilities 37

🗂️ Paths & File System Navigation

> *When you write a Python script that works with files, the biggest challenge isn't the logic—it's figuring out where the files actually are. This chapter teaches the most important concept in practical Python scripting: how to reliably locate and work with files on any computer, in any directory, on any operating system.*

⚠️ The Problem We're Solving

Imagine you write a script that reads HTML files. On your Windows computer, you test it:

```
with open("C:\\Users\\philip\\files\\chapters\\ts1-1.html") as f:
    content = f.read()
```

It works fine. You copy the script to your Linux web server and run it:

```
with open("C:\\Users\\philip\\files\\chapters\\ts1-1.html") as f:
    content = f.read()
```

It crashes. Why? Windows and Linux have different path formats, and the directories are different.

> ❌ **COMMON MISTAKE:** Hard-coding file paths. This makes your script work on one computer but nowhere else.

The solution: **make your script find files relative to where it's located**. This is what this chapter teaches.

✨ `__file__`: The Magic Variable

> `__file__` *is a special Python variable that automatically contains the path to the current script. It's like Python saying "here's where I am right now."*

What `__file__` Contains

Create a test file called `test.py` with this code:

```
print(__file__)
```

On Windows: Running from `C:\Users\philip\scripts\`

```
C:\Users\philip\scripts\test.py
```

On Linux: Running from `/home/philip/scripts/`

```
/home/philip/scripts/test.py
```

✅ **WHAT IT DOES:** `__file__` is a string containing the absolute path to the Python script currently running. It automatically adapts to Windows or Linux path format.

🤔 **WHY IT MATTERS:** Your script now knows where it is. This is the foundation for finding other files relative to its location.

📦 The pathlib Module and Path Objects

Python's `pathlib` module provides a better way to work with file paths. Instead of string manipulation (which is error-prone), you use `Path` objects that handle Windows/Linux differences automatically.

Creating a Path Object

```python
from pathlib import Path

# Convert __file__ to a Path object
script_path = Path(__file__)
print(script_path)
```

✅ **WHAT IT DOES:** `Path(__file__)` converts the string path into a `Path` object. Now you can call methods on it.

🤔 **WHY USE Path INSTEAD OF STRINGS:**

- Path handles Windows backslashes vs Linux forward slashes automatically
- Path has useful methods like `.parent`, `.exists()`, `.open()`
- Path prevents errors from mixing path separators

Alternative: os.path (Old Way)

```python
import os

script_path = os.path.abspath(__file__)
script_dir = os.path.dirname(script_path)
```

Old Way (os.path)

```python
import os
import os.path

chapters_dir = os.path.join(
    os.path.dirname(__file__),
    "courses",
```

New Way (pathlib)

```python
from pathlib import Path

chapters_dir = Path(__file__).parent / "courses" / "html"
```

```
    "html"
)
```

💡 **Modern Recommendation**

Always use `pathlib.Path` for new code. It's cleaner, safer, and more readable. The old `os.path` module still works, but `pathlib` is the modern standard.

👆 The .parent Property: Going Up Directories

> *The `.parent` property gets the directory that contains a file or directory. You can chain multiple `.parent` calls to go up multiple levels.*

Understanding Parent Step-by-Step

Let's say your script is at:

```
/var/www/myapp/rebuild_typescript_course.py
```

Here's what `.parent` does at each step:

Path Traversal with .parent

Stage 1:

```
Path(__file__)
```

Result:

```
/var/www/myapp/rebuild_typescript_course.py
```

This is the script file itself.

Stage 2:

```
Path(__file__).parent
```

Result:

```
/var/www/myapp
```

Go up one level to the directory containing the script.

Stage 3:

```
Path(__file__).parent.parent
```

Result:

```
/var/www
```

Go up two levels.

Stage 4:

```
Path(__file__).parent.parent.parent
```

Result:

```
/var
```

Go up three levels.

✅ **WHAT IT DOES:** Each `.parent` goes up one directory level. You can chain them: `.parent.parent.parent` goes up three levels.

🤔 **WHY THIS MATTERS:** Your script doesn't know where it will be placed. By using `.parent`, your script can find files relative to its own location, regardless of where it's copied to.

🔨 Building Paths with the / Operator

Once you have a directory, you add subdirectories and filenames using the `/` operator (the slash character). This is much cleaner than string concatenation.

The / Operator for Paths

```python
from pathlib import Path

# Start from the script's directory
script_dir = Path(__file__).parent

# Build a path step by step
chapters_dir = script_dir / "courses" / "Programming" / "TypeScript" / "html"

# Get a specific file
chapter_file = chapters_dir / "ts1-1.html"

print(chapter_file)
# Output: /var/www/myapp/courses/Programming/TypeScript/html/ts1-1.html
```

✅ **WHAT IT DOES:** The `/` operator appends a subdirectory or filename to a path. It works on Windows and Linux automatically.

🤔 **WHY USE / INSTEAD OF STRING CONCATENATION:**

- ✅ `script_dir / "chapters"` — Clean and readable
- ❌ `script_dir + "\chapters"` — Breaks on Linux (needs forward slash)
- ❌ `script_dir + "/" + "chapters"` — Verbose and fragile

Real Example: rebuild_typescript_course.py

Here's how the actual rebuild script finds chapters:

```
from pathlib import Path

# The script is at: /var/www/myapp/rebuild_typescript_course.py
# We want chapters at: /var/www/myapp/courses/Programming/TypeScript/html/

CHAPTERS_DIR = Path(__file__).parent / "courses" / "Programming" / "TypeScript" / "html"

# Now we can use it:
html_files = list(CHAPTERS_DIR.glob("ts1-*.html"))
for html_file in html_files:
    print(f"Found: {html_file}")
```

✅ Checking If Files Exist

The .exists() Method

```
from pathlib import Path

file_path = Path("/var/www/myapp/courses/TypeScript/html/ts1-1.html")

if file_path.exists():
    print("✅ File found!")
else:
    print("❌ File not found!")
```

✅ **WHAT IT DOES:** `.exists()` returns `True` if the file or directory exists, `False` otherwise.

Other Useful Methods

```
from pathlib import Path

file_path = Path("/var/www/myapp/courses/ts1-1.html")

# Is it a file (not a directory)?
if file_path.is_file():
    print("It's a file")
```

```
# Is it a directory (not a file)?
if file_path.is_dir():
    print("It's a directory")

# Get just the filename without the directory
print(file_path.name)  # Output: ts1-1.html

# Get the file extension
print(file_path.suffix)  # Output: .html

# Get the filename without extension
print(file_path.stem)  # Output: ts1-1
```

🖥️ Windows vs Linux: Path Formats

The Path Format Problem

Windows uses backslashes:

```
C:\Users\philip\files\chapters\ts1-1.html
```

Linux uses forward slashes:

```
/home/philip/files/chapters/ts1-1.html
```

If you hard-code Windows paths, your script breaks on Linux. If you hard-code Linux paths, your script breaks on Windows.

> ✅ **THE SOLUTION:** Use `pathlib.Path`. It automatically uses the correct separator for your operating system.

```
from pathlib import Path

# This works the same way on Windows and Linux!
chapters_dir = Path(__file__).parent / "courses" / "html"

# On Windows, it becomes: C:\path\to\courses\html
# On Linux, it becomes: /path/to/courses/html
print(chapters_dir)
```

⚠️ Common Mistakes & How to Fix Them

> ❌ **MISTAKE 1: Forgetting to import Path**

```
# This will crash!
chapters_dir = Path(__file__).parent / "courses"
# NameError: name 'Path' is not defined
```

Fix:

```
from pathlib import Path  # ← Add this!

chapters_dir = Path(__file__).parent / "courses"
```

❌ MISTAKE 2: Using + instead of / to combine paths

```
# This doesn't work as expected!
chapters_dir = Path(__file__).parent + "/courses"
# TypeError: unsupported operand type(s) for +: 'PosixPath' and 'str'
```

Fix:

```
chapters_dir = Path(__file__).parent / "courses"  # ← Use /
```

❌ MISTAKE 3: Assuming `__file__` is always absolute

In some cases (like running with `python -m`), `__file__` might be relative.

Fix:

```
from pathlib import Path

# Make sure it's absolute
script_dir = Path(__file__).resolve().parent
```

🎯 Edge Cases & Special Situations

Edge Case 1: File Doesn't Exist Yet

If you're creating a new file, the path won't exist, but that's OK:

```
from pathlib import Path

file_path = Path(__file__).parent / "new_file.txt"

if not file_path.exists():
    with open(file_path, "w") as f:
        f.write("Hello")
```

Edge Case 2: Symbolic Links (Shortcuts)

`.resolve()` follows symbolic links to the actual file:

```
from pathlib import Path
```

```
# If __file__ is a symlink, follow it to the real file
real_script = Path(__file__).resolve()
```

Edge Case 3: Relative Paths from Current Directory

Sometimes you want files relative to where the user ran the command, not where the script is:

```
from pathlib import Path

# User's current working directory
current_dir = Path(".").resolve()

# Script's directory
script_dir = Path(__file__).resolve().parent
```

🌎 Real-World Example: rebuild_typescript_course.py

Here's how the concepts in this chapter are used in actual production code:

```
from pathlib import Path

# This script is at: /var/www/myapp/rebuild_typescript_course.py

# Start from the script's location
SCRIPT_DIR = Path(__file__).resolve().parent

# Build the path to chapters folder relative to the script
CHAPTERS_DIR = SCRIPT_DIR / "courses" / "Programming" / "TypeScript" / "html"

# Check if it exists
if not CHAPTERS_DIR.exists():
    print(f"❌ Chapters directory not found: {CHAPTERS_DIR}")
    exit(1)

# Find all ts1-*.html files
html_files = sorted(CHAPTERS_DIR.glob("ts1-*.html"))

for html_file in html_files:
    print(f"📄 Processing: {html_file.name}")
    with open(html file, "r") as f:
```

💻 Coding Challenge

Challenge: Create a Path Finder Script

Create a Python script that:

1. Uses `__file__` to find the script's directory
2. Creates a path to a subdirectory called `data` using `/`
3. Checks if the `data` directory exists

4. If it exists, lists all files in it
5. If it doesn't exist, creates it

Goal: Practice using `Path`, `.parent`, `.exists()`, and the `/` operator.
→ Solution

🎯 What's Next

Now that you can locate files reliably, the next chapter covers **Configuration & Environment Variables** — how to load settings like database credentials from `.env` files so your scripts can adapt to different environments.

⚙️ Configuration & Environment Variables

> *Every practical Python script needs to work in different environments. Your database might be on localhost during development, but in production it's at a different address. Your API keys are different for testing vs production. This chapter teaches how to manage configuration securely using environment variables and `.env` files.*

⚠️ The Problem: Hard-Coded Secrets

Many scripts hard-code sensitive information:

```
import mysql.connector

# ❌ NEVER DO THIS!
db = mysql.connector.connect(
    host="localhost",
    user="philip",
    password="MySecretPassword123",  # ← BAD!
    database="learning_blog"
)
```

> **❌ CRITICAL SECURITY ISSUE:**
>
> - Password is visible in the source code
> - If you commit to Git, it's in version control forever
> - Anyone who reads the code gets your database password
> - If the repo goes public, attackers have access

The solution: Move secrets to a `.env` file that's never committed to Git.

📝 What is a .env File?

> *A `.env` file is a simple text file that stores configuration variables. It's kept on the server (not in Git), and your script reads it at startup.*

A Real .env File Example

.env

```
# Database Configuration DB_HOST=localhost DB_NAME=learning_blog DB_USER=philip DB_PASS=your-db-password-here DB_PORT=3306 DB_CHARSET=utf8mb4 # API Keys API_KEY=sk-proj-abc123def456 API_URL=https://api.example.com # Application Settings DEBUG=true LOG_LEVEL=info
```

✅ **WHAT IT DOES:** A `.env` file is a configuration file with `KEY=VALUE` pairs. Each line is one setting. Comments start with `#`.

Format Rules

- **One setting per line:** `KEY=VALUE`
- **No quotes needed:** `DB_HOST=localhost` (not `"localhost"`)
- **Comments with #:** `# This is a comment`
- **Empty lines ignored:** Blank lines are skipped
- **Case sensitive:** `DB_HOST` and `db_host` are different
- **No spaces around =:** `KEY=value` (not `KEY = value`)

📦 Installing python-dotenv

To load `.env` files in Python, you need the `python-dotenv` package:

```
pip install python-dotenv
```

Add it to your `requirements.txt`:

```
mysql-connector-python==8.2.0
python-dotenv==1.0.0
```

🔓 Loading Environment Variables

The load_dotenv() Function

```
from dotenv import load_dotenv
import os

# Load .env file from current directory (or parents)
load_dotenv()

# Now read variables using os.environ
host = os.environ['DB_HOST']
user = os.environ['DB_USER']
password = os.environ['DB_PASS']

print(f"Connecting to {host} as {user}")
```

✅ **WHAT IT DOES:** `load_dotenv()` reads the `.env` file and loads all variables into `os.environ` (a dictionary of environment variables).

With Explicit Path

If `.env` is in a specific location:

```python
from dotenv import load_dotenv
from pathlib import Path
import os

# Specify exact path to .env
env_path = Path("/var/www/.env")
load_dotenv(env_path)

host = os.environ['DB_HOST']
```

🤔 **WHY SPECIFY A PATH:** If `.env` is outside the web root (for security), you need to tell `load_dotenv()` where to find it.

🔍 Accessing Variables

Method 1: Direct Access (Crashes if Missing)

```python
import os

# This will crash if DB_HOST is not defined
host = os.environ['DB_HOST']
# KeyError: 'DB_HOST' if variable doesn't exist
```

Method 2: Safe Access with .get()

```python
import os

# Returns the value, or None if not defined
host = os.environ.get('DB_HOST')

# With a default value if not defined
host = os.environ.get('DB_HOST', 'localhost')

# Check if it exists
if host is None:
    print("DB_HOST not defined!")
```

Direct Access

```python
host = os.environ['DB_HOST']
# Crashes if missing
# Good for required settings
```

Safe Access

```python
host = os.environ.get('DB_HOST', 'localhost')
# Returns default if missing
# Good for optional settings
```

🔄 Type Conversion

Environment variables are always strings. If you need a number or boolean, you must convert them.

Converting to Different Types

```
import os
from dotenv import load_dotenv

load_dotenv()

# String (default, no conversion needed)
host = os.environ.get('DB_HOST')  # "localhost"

# Integer (port number)
port = int(os.environ.get('DB_PORT', '3306'))
# String "3306" → Integer 3306

# Boolean (debug mode)
debug = os.environ.get('DEBUG', 'false').lower() == 'true'
# String "true" → Boolean True

# Float (timeout in seconds)
timeout = float(os.environ.get('TIMEOUT', '30.5'))

print(f"Host: {host} (type: {type(host).__name__})")
print(f"Port: {port} (type: {type(port).  name  })")
```

Helper Function for Booleans

```
def get_bool_env(key: str, default: bool = False) -> bool:
    """Convert environment variable to boolean."""
    value = os.environ.get(key, str(default))
    return value.lower() in ('true', '1', 'yes', 'on')

# Usage:
debug = get_bool_env('DEBUG', False)
```

✅ Validating Configuration

Checking Required Variables

```
from dotenv import load_dotenv
import os
import sys

load_dotenv()

# List of required variables
REQUIRED_VARS = ['DB_HOST', 'DB_USER', 'DB_PASS', 'DB_NAME']

# Check each one
missing = []
for var in REQUIRED_VARS:
    if var not in os.environ:
        missing.append(var)

if missing:
    print("❌ Missing required variables:")
    for var in missing:
```

```
        print(f"   - {var}")
    sys.exit(1)
```

🔒 Security Best Practices

Never Commit .env to Git

Add `.env` to your `.gitignore` file:

```
.gitignore
# Environment variables with secrets .env # But keep an example file (without secrets) #
.env.example
```

> 🔒 **CRITICAL:** If you accidentally commit `.env` with passwords, your database is compromised. You must:
>
> 1. Change all passwords immediately
> 2. Remove the commit from Git history (or assume secrets are leaked)
> 3. Check if anyone accessed your database

File Permissions

On Linux/Mac, restrict `.env` to be readable only by the owner:

```
chmod 600 .env
# Now only the owner can read it

# Verify:
ls -la .env
# Should show: -rw------- (owner read/write only)
```

.env.example Template

Create a template showing what variables are needed (without real values):

```
.env.example
# Copy this file to .env and fill in real values # DO NOT commit .env to git!
DB_HOST=localhost DB_NAME=learning_blog DB_USER=your_username DB_PASS=your_password_here
DB_PORT=3306 DEBUG=false LOG_LEVEL=info
```

🌎 Real-World Example: db_utils.py

Here's how the actual database utility script uses environment variables:

```
from dotenv import load_dotenv
import os
import mysql.connector
```

```
# Load .env file
load_dotenv()

# Get database credentials from environment
DB_CONFIG = {
    'host': os.environ.get('DB_HOST'),
    'user': os.environ.get('DB_USER'),
    'password': os.environ.get('DB_PASS'),
    'database': os.environ.get('DB_NAME'),
    'port': int(os.environ.get('DB_PORT', '3306')),
    'charset': os.environ.get('DB_CHARSET', 'utf8mb4')
}

class DatabaseConnection:
    def __init__(self):
        self.connection = None
```

⚠️ Common Mistakes

❌ MISTAKE 1: Forgetting load_dotenv()

```
import os

host = os.environ['DB_HOST']
# KeyError: 'DB_HOST' not found!
# (You never called load_dotenv() to load the .env file)
```

Fix:

```
from dotenv import load_dotenv

load_dotenv()  # ← Add this!
host = os.environ['DB_HOST']
```

❌ MISTAKE 2: Expecting Variables to be Numbers

```
port = os.environ['DB_PORT']  # This is a STRING "3306"

if port > 3000:  # ❌ TypeError: '>' not supported between str and int
    pass
```

Fix:

```
port = int(os.environ['DB_PORT'])  # ← Convert to integer

if port > 3000:  # ✅ Now it works
    pass
```

❌ MISTAKE 3: Committing .env to Git

```
git add .
git commit -m "Add configuration"
# Oops! You just committed DB_PASS with your real password!
```

Fix:

```
# Add to .gitignore BEFORE committing anything:
echo ".env" >> .gitignore
git add .gitignore
git commit -m "Add .gitignore"
```

💻 Coding Challenges

Challenge 1: Load and Display Configuration

Create a script that:

1. Creates a `.env` file with database settings
2. Loads it with `load_dotenv()`
3. Displays all settings (except the password)

Goal: Practice creating and loading `.env` files.
→ Solution

Challenge 2: Type Conversion and Validation

Create a script that:

1. Loads a `.env` file
2. Converts port from string to integer
3. Converts debug flag from string to boolean
4. Validates that required variables exist

Goal: Practice type conversion and validation.
→ Solution

Challenge 3: Secure Configuration Management

Create a script that:

1. Loads configuration with fallback defaults
2. Validates all required variables
3. Reports configuration status (what's set, what uses defaults)
4. Shows security warnings if DEBUG is true in production

Goal: Practice building robust configuration systems.
→ Solution

🎯 What's Next

Now that you can load configuration securely, the next chapter covers **Database Connections** — how to use this configuration to connect to MySQL/MariaDB and execute queries.

🗄️ Database Connections

> *Now that you can load configuration securely, you're ready to connect to a database. This chapter teaches how to use the mysql-connector-python library to connect to MySQL/MariaDB, execute queries, handle errors, and clean up resources properly. You'll learn the patterns used in production code.*

💡 Why Database Connections Matter

A database connection is like a phone call between your Python script and the database server. You need to:

1. **Establish the connection** — Dial the number and wait for pickup
2. **Send queries** — Ask questions and get answers
3. **Handle errors** — What if the database is unavailable?
4. **Close the connection** — Hang up when done

Many bugs happen because developers forget step 4 — connections that never close consume server resources and eventually crash the server.

📦 Installing mysql-connector-python

```
pip install mysql-connector-python==8.2.0
```

Add to `requirements.txt`:

```
mysql-connector-python==8.2.0
python-dotenv==1.0.0
```

> ✅ **WHAT IT IS:** mysql-connector-python is a pure Python implementation of the MySQL protocol. It doesn't require any C libraries, so it works on any system.

🔌 Connection Basics

The Simplest Connection

```
import mysql.connector

# Create a connection
connection = mysql.connector.connect(
    host="localhost",
    user="philip",
    password="your-db-password-here",
```

```
    database="learning_blog"
)

print("✅ Connected to database!")

# When done, close it
connection.close()
print("✅ Connection closed")
```

✅ **WHAT IT DOES:** `mysql.connector.connect()` creates a connection to the database server. You provide credentials and database name.

Connection Parameters

Parameter	Required	Example	Description
`host`	Yes	`"localhost"`	Database server address
`user`	Yes	`"philip"`	Database username
`password`	Yes	`"secret123"`	Database password
`database`	Yes	`"learning_blog"`	Database name
`port`	No	`3306`	Port (default: 3306)
`charset`	No	`"utf8mb4"`	Character set (default: utf8mb4)
`autocommit`	No	`True`	Auto-commit changes

🔄 Cursors: Executing Queries

> *A **cursor** is how you send SQL queries to the database. You create a cursor from a connection, then use it to execute queries and fetch results.*

Creating and Using a Cursor

```
import mysql.connector

connection = mysql.connector.connect(
    host="localhost",
    user="philip",
    password="your-db-password-here",
    database="learning_blog"
)

# Create a cursor from the connection
cursor = connection.cursor()

# Execute a query
cursor.execute("SELECT COUNT(*) FROM pages")

# Fetch the result
```

```
result = cursor.fetchone()
count = result[0]

print(f"Total pages: {count}")
```

✅ **WHAT IT DOES:**

- `cursor()` — Create a cursor from a connection
- `execute(sql)` — Send a SQL query
- `fetchone()` — Get one row of results
- `fetchall()` — Get all rows of results

Fetch Methods

```
# SELECT COUNT(*) FROM pages  → 1 row with 1 column
result = cursor.fetchone()
# Returns: (42,)  ← tuple with one element

# SELECT id, title FROM pages  → many rows
rows = cursor.fetchall()
# Returns: [(1, 'Chapter 1'), (2, 'Chapter 2'), ...]

# Iterate over results
for row in rows:
    id, title = row
    print(f"{id}: {title}")

# Get column count
num_columns = cursor.rowcount
print(f"Affected {num_columns} rows")
```

⚠️ Error Handling

> *Database operations fail for many reasons: server offline, wrong credentials, permission denied, network timeout. Always use try/except to handle these gracefully.*

Catching Connection Errors

```
import mysql.connector
from mysql.connector import Error

try:
    connection = mysql.connector.connect(
        host="localhost",
        user="philip",
        password="wrong_password",
        database="learning_blog"
    )

except Error as e:
    print(f"❌ Connection failed: {e}")
    print(f"   Error code: {e.errno}")
```

```
    print(f"   Error message: {e.msg}")

else:
    print("✅ Connected successfully!")
    connection.close()
```

Common Error Codes

Error Code	Meaning	Solution
1045	Access denied (wrong password/user)	Check credentials in .env
1049	Unknown database	Check database name
2003	Can't connect to server	Check if server is running
2006	MySQL server has gone away	Connection timed out, reconnect

🎯 Context Managers: The Right Way

> *Using `try/except` with `close()` is error-prone. If an exception occurs before `close()`, the connection stays open forever. The `with` statement (context manager) handles this automatically.*

The Wrong Way (Connections Can Leak)

```
# ❌ RISKY: If an exception occurs, close() never runs
connection = mysql.connector.connect(...)
cursor = connection.cursor()
cursor.execute("SELECT * FROM pages")
connection.close()  # ← Never reached if error occurs!
```

The Right Way (With Context Manager)

```
import mysql.connector
from contextlib import closing

connection = mysql.connector.connect(
    host="localhost",
    user="philip",
    password="your-db-password-here",
    database="learning_blog"
)

# ✅ SAFE: closing() ensures close() is always called
with closing(connection.cursor()) as cursor:
    cursor.execute("SELECT COUNT(*) FROM pages")
    result = cursor.fetchone()
    print(f"Pages: {result[0]}")

# At this point, cursor is closed automatically!
connection.close()
```

🤔 **WHY CONTEXT MANAGERS:** The `with` statement guarantees cleanup code runs, even if exceptions occur. This prevents resource leaks that crash servers.

🌍 Real-World Example: DatabaseConnection Class

Here's how db_utils.py structures database operations:

```
class DatabaseConnection:
    def __init__(self, host, user, password, database, port=3306):
        self.config = {
            'host': host,
            'user': user,
            'password': password,
            'database': database,
            'port': port,
            'charset': 'utf8mb4'
        }
        self.connection = None

    def connect(self) -> bool:
        """Establish database connection."""
        try:
            self.connection = mysql.connector.connect(**self.config)
            return True
        except Error as e:
            print(f"❌ Connection failed: {e}")
            return False
```

⚠️ Common Mistakes

❌ MISTAKE 1: Forgetting to close connections

```
connection = mysql.connector.connect(...)
cursor = connection.cursor()
cursor.execute("SELECT ...")
# ❌ Never closed! Connection leaks!
```

Fix:

```
try:
    connection = mysql.connector.connect(...)
    cursor = connection.cursor()
    cursor.execute("SELECT ...")
finally:
    cursor.close()
    connection.close()
```

❌ MISTAKE 2: Not handling connection errors

```
connection = mysql.connector.connect(host="wrong_host", ...)
# ❌ If connection fails, this crashes!
```

Fix:

```
try:
    connection = mysql.connector.connect(...)
except Error as e:
    print(f"Connection failed: {e}")
    sys.exit(1)
```

❌ MISTAKE 3: Accessing columns by wrong index

```
# SELECT id, title, created_at FROM pages
row = cursor.fetchone()
print(row[5])  # ❌ IndexError! Only 3 columns (0, 1, 2)
```

Fix:

```
id, title, created = row  # ✅ Unpack correctly
print(id, title, created)
```

💻 Coding Challenges

Challenge 1: Test Database Connection

Create a script that:

1. Loads configuration from .env
2. Attempts to connect to the database
3. Shows success/failure message with error details
4. Safely closes the connection

Goal: Practice basic connection and error handling.
→ Solution

Challenge 2: Query and Display Results

Create a script that:

1. Connects to the database
2. Executes a SELECT query
3. Displays results in a formatted table
4. Handles empty results gracefully

Goal: Practice executing queries and displaying results.

→ Solution

Challenge 3: DatabaseConnection Class

Create a reusable DatabaseConnection class that:

1. Accepts configuration parameters
2. Has connect/disconnect methods
3. Has execute_query method for SELECT
4. Has execute_update method for INSERT/UPDATE/DELETE
5. Handles all errors gracefully

Goal: Build a reusable database utility class.
→ Solution

🎯 What's Next

You now have the tools to connect to databases. Chapter 4 covers **Building db_utils.py** — a complete breakdown of the actual database utility script used in production, line-by-line explanation of every function.

🏗️ Building db_utils.py

> *This chapter is a complete line-by-line breakdown of the actual `db_utils.py` module used in production to populate the learning_blog database with course chapters. You'll see how everything from chapters 1-3 comes together: paths, configuration, database connections, error handling, and real-world SQL operations.*

📋 Overview: What db_utils.py Does

The `db_utils.py` module provides database utility functions for:

1. **DatabaseConnection class** — Connects to MySQL/MariaDB
2. **get_or_create_subtopic()** — Find or create a course
3. **insert_chapter_with_content()** — Insert a chapter and its HTML content
4. **rebuild_course()** — Rebuild an entire course from chapters

It's used by rebuild_typescript_course.py to populate the database with generated course chapters.

📁 File Location

```
debserver/claude/generated/scripts/python/db_utils.py
```

📦 Imports & Setup

```python
import mysql.connector
from mysql.connector import Error
import os
from dotenv import load_dotenv
from typing import Optional, Dict, List, Tuple
from urllib.parse import quote

# Load environment variables from .env file
load_dotenv()
```

✅ **WHAT EACH IMPORT DOES:**

- `mysql.connector` — Database connection library
- `Error` — Database error exception class
- `dotenv` — Load .env configuration
- `typing` — Type hints for function parameters
- `urllib.parse.quote` — URL-encode strings for slug generation

🔌 DatabaseConnection Class

The core class that manages all database operations.

The __init__ Method

```
class DatabaseConnection:
    """Manage MySQL/MariaDB database connections."""

    def __init__(self):
        """Initialize connection from environment variables."""
        self.connection = None

        # Get credentials from environment
        self.config = {
            'host': os.environ.get('DB_HOST'),
            'user': os.environ.get('DB_USER'),
            'password': os.environ.get('DB_PASS'),
            'database': os.environ.get('DB_NAME'),
            'port': int(os.environ.get('DB_PORT', '3306')),
            'charset': os.environ.get('DB_CHARSET', 'utf8mb4')
        }
```

🤔 **WHY THIS STRUCTURE:**

- Loads credentials from .env (not hard-coded)
- Sets sensible defaults (port=3306, charset=utf8mb4)
- Stores config for reuse across method calls
- Converts port string to integer

The connect() Method

```
def connect(self) -> bool:
    """Establish database connection."""
    try:
        self.connection = mysql.connector.connect(**self.config)
        print("✅ Database connection established")
        return True

    except Error as e:
        print(f"❌ Connection failed: {e}")
        return False
```

The disconnect() Method

```
def disconnect(self) -> None:
    """Close database connection."""
    if self.connection and self.connection.is_connected():
        self.connection.close()
        print("✅ Database connection closed")
```

🔧 Helper Functions

generate_slug() Function

Convert a title into a URL-safe slug:

```
def generate_slug(title: str) -> str:
    """
    Convert title to URL-safe slug.

    Example: "Why TypeScript & Setup" → "why-typescript-setup"
    """
    # Convert to lowercase
    slug = title.lower()

    # Replace spaces with hyphens
    slug = slug.replace(' ', '-')

    # Remove special characters
    slug = quote(slug, safe='-')

    # Remove multiple consecutive hyphens
    while '--' in slug:
        slug = slug.replace('--', '-')

    return slug
```

✅ **WHAT IT DOES:**

- Converts "Why TypeScript & Setup" → "why-typescript-setup"
- .lower() makes it lowercase
- .replace() converts spaces to hyphens
- quote() removes special characters
- Final loop removes double hyphens

💾 Core Database Functions

get_or_create_subtopic()

Find a course (subtopic) or create it if it doesn't exist:

```
def get_or_create_subtopic(
    db: DatabaseConnection,
    subject_id: int,
    subtopic_name: str,
    description: str = ""
) -> Optional[int]:
    """
    Get or create a subtopic (course).
    Returns the subtopic ID, or None if error.
    """
    cursor = db.connection.cursor()
```

```
    try:
        # Check if subtopic already exists
        query = "SELECT id FROM subtopics WHERE name = %s AND subject_id = %s"
        cursor.execute(query, (subtopic_name, subject_id))
        result = cursor.fetchone()

        if result:
            return result[0]  # Found existing subtopic
```

✅ **WHAT IT DOES:**

- Checks if subtopic (course) already exists
- If exists: returns its ID
- If not: creates it and returns new ID
- Uses cursor.lastrowid to get the new ID
- Commits transaction if successful
- Rolls back if error occurs

insert_chapter_with_content()

Insert a chapter and its HTML content into the database:

```
def insert_chapter_with_content(
    db: DatabaseConnection,
    subtopic_id: int,
    chapter_title: str,
    html_content: str,
    sort_order: int = 10
) -> Optional[int]:
    """
    Insert a chapter with HTML content.
    Returns page_id, or None if error.
    """
    cursor = db.connection.cursor()

    try:
        # Generate slug from title
        slug = generate_slug(chapter_title)

        # Insert into pages table
        page_query = """
            INSERT INTO pages (subtopic_id, title, slug, sort_order)
            VALUES (%s, %s, %s, %s)
```

🎯 Why This Pattern Works

💡 **Key Design Decisions**

- **Separation of Concerns:** DatabaseConnection handles connections, helper functions handle operations
- **Parameterized Queries:** All queries use %s placeholders (prevents SQL injection)
- **Error Handling:** Every database operation has try/except
- **Transactions:** Commit on success, rollback on error
- **Resource Cleanup:** finally blocks ensure cursors always close

- **Type Hints:** Return types are clear (int, bool, None, etc.)
- **Configuration:** All credentials come from .env, never hard-coded

📌 Common Patterns Used

Pattern	Purpose	Example
Parameterized Query	Prevent SQL injection	`cursor.execute("SELECT * FROM t WHERE id = %s", (id,))`
Check Before Insert	Avoid duplicates	SELECT first, then INSERT if not found
lastrowid	Get inserted row ID	`page_id = cursor.lastrowid`
Commit/Rollback	Transaction control	commit() on success, rollback() on error
try/except/finally	Error handling + cleanup	finally block always closes cursor

🌎 Real-World Usage Example

```
from db_utils import DatabaseConnection, get_or_create_subtopic, insert_chapter_with_content

# Connect to database
db = DatabaseConnection()
if not db.connect():
    exit(1)

try:
    # Get or create TypeScript course
    subtopic_id = get_or_create_subtopic(
        db,
        subject_id=3,  # Programming
        subtopic_name="TypeScript Fundamentals",
        description="Learn TypeScript from basics to generics"
    )

    # Insert a chapter
    page_id = insert_chapter_with_content(
        db,
        subtopic_id=subtopic_id,
```

💻 Coding Challenges

Challenge 1: Test db_utils.py Connection

Create a script that:

1. Imports DatabaseConnection from db_utils
2. Attempts to connect using credentials from .env
3. Shows connection status and database info

4. Safely closes the connection

Goal: Verify db_utils works with your database.
→ Solution

Challenge 2: Insert a Test Subtopic

Create a script that:

1. Connects using db_utils
2. Uses get_or_create_subtopic() to create a test course
3. Shows the returned subtopic_id
4. Calls it again to verify it returns the same ID (no duplicate)

Goal: Practice using db_utils helper functions.
→ Solution

Challenge 3: Complete Course Integration

Create a script that:

1. Creates a test subtopic
2. Inserts 3 test chapters with HTML content
3. Verifies chapters are in database by querying them back
4. Reports success with all IDs

Goal: Full database integration workflow.
→ Solution

🎯 What's Next

Chapter 5 shows how **rebuild_typescript_course.py** uses db_utils to automatically populate the database by reading generated HTML chapter files and inserting them with all the functions you've learned here.

🔄 Building rebuild_typescript_course.py

> *This chapter brings everything together.* `rebuild_typescript_course.py` *is the complete automation script that uses all the tools from chapters 1-4: navigating the filesystem (paths), loading configuration (.env), connecting to databases, and using db_utils to populate the database with generated course chapters. This is a real-world, production-ready script.*

📋 Overview: What This Script Does

The script automates the complete workflow:

Step 1: Find Chapter Files
Use Path and glob() to find all ts1-*.html files

Step 2: Parse Chapter Data
Read HTML files and extract title from <h2> tags with regex

Step 3: Connect to Database
Load .env and use DatabaseConnection to connect

Step 4: Create Course
Use get_or_create_subtopic() to create TypeScript Fundamentals course

Step 5: Insert Chapters
For each chapter, call insert_chapter_with_content()

Step 6: Report Results
Display summary of what was created

📁 Directory Structure

The script expects this layout:

```
myapp/
├── .env                          ← Configuration (loaded by script)
├── rebuild_typescript_course.py  ← This script
└── courses/
    └── Programming/
        └── TypeScript/
            └── html/
                ├── ts1-1.html    ← Chapter files
```

```
                ├── ts1-2.html
                ├── ts1-3.html
                ...
                └── ts1-12.html
```

> 🧐 **WHY THIS STRUCTURE:**
> Using Path(__file__).parent, the script can navigate relative to itself. No matter where you run it from, it finds the chapters in the correct location.

🔧 Main Components

Imports & Constants

```python
from pathlib import Path
import sys
import re

from db_utils import DatabaseConnection, get_or_create_subtopic, insert_chapter_with_content

# Constants
SUBJECT_ID = 3  # Programming subject
SUBJECT_NAME = "Programming"
SUBTOPIC_NAME = "TypeScript Fundamentals"

# Calculate path to chapters directory
SCRIPT_DIR = Path(__file__).resolve().parent
CHAPTERS_DIR = SCRIPT_DIR / "courses" / "Programming" / "TypeScript" / "html"
```

Finding Chapter Files

```python
def find_chapters() -> list:
    """Find all chapter HTML files."""
    if not CHAPTERS_DIR.exists():
        print(f"❌ Chapters directory not found: {CHAPTERS_DIR}")
        return []

    # Find all ts1-*.html files and sort them
    html_files = sorted(CHAPTERS_DIR.glob("ts1-*.html"))

    if not html_files:
        print(f"⚠️ No chapter files found in {CHAPTERS_DIR}")
        return []

    print(f"✅ Found {len(html_files)} chapter files")
    return html_files
```

Parsing Chapter Data

```python
def parse_chapter(html_file: Path) -> dict:
    """Extract chapter data from HTML file."""
    try:
```

```
        # Read HTML content
        html_content = html_file.read_text(encoding='utf-8')

        # Extract chapter number from filename (ts1-1.html → 1)
        match = re.match(r"ts1-(\d+)", html_file.stem)
        if not match:
            return None

        chapter_num = int(match.group(1))

        # Extract title from <h2> tag
        title_match = re.search(r'<h2>(.*?)</h2>', html_content)
        chapter_title = title_match.group(1) if title_match else f"Chapter {chapter_num}"

        return {
            'number': chapter_num,
            'title': chapter_title,
```

Main Workflow

```
def main():
    """Main rebuild workflow."""
    print("=" * 60)
    print("🚀 TypeScript Course Rebuild")
    print("=" * 60 + "\n")

    # Step 1: Find chapters
    html_files = find_chapters()
    if not html_files:
        exit(1)

    # Step 2: Connect to database
    print("\n🔌 Connecting to database...")
    db = DatabaseConnection()
    if not db.connect():
        exit(1)

    try:
        # Step 3: Create or get subtopic
        print("\n📚 Creating course...")
        subtopic_id = get_or_create_subtopic(
```

📌 Key Patterns Used

Pattern	From Chapter	Purpose
Path(__file__).parent	Chapter 1	Find script's directory
glob("ts1-*.html")	Chapter 1	Find all chapter files
load_dotenv()	Chapter 2	Load database credentials
DatabaseConnection	Chapter 3	Connect to database
get_or_create_subtopic()	Chapter 4	Create course
insert_chapter_with_content()	Chapter 4	Insert chapters

🚀 Running the Script

```
# From the myapp directory:
python rebuild_typescript_course.py

# Output:
============================================================
🚀 TypeScript Course Rebuild
============================================================

✅ Found 12 chapter files

🔌 Connecting to database...
✅ Database connection established

📚 Creating course...
✅ Created subtopic 'TypeScript Fundamentals' with ID 8

📝 Inserting 12 chapters...
✅ Created chapter with ID: 101
✅ Created chapter with ID: 102
✅ Created chapter with ID: 103
...
```

💻 Coding Challenges

Challenge 1: Test File Discovery

Create a script that:

1. Uses Path(__file__) to find its directory
2. Navigates to courses/Programming/TypeScript/html/
3. Uses glob() to find all ts1-*.html files
4. Reports how many found and lists them

Goal: Practice path navigation and file discovery.
→ Solution

Challenge 2: Parse Chapter Metadata

Create a script that:

1. Finds all chapter HTML files
2. For each file, extract: chapter number, title from <h2>
3. Display a table with all chapters
4. Show chapter count

Goal: Practice HTML parsing with regex.
→ Solution

Challenge 3: Complete Rebuild Script

Create a script that:

1. Finds all chapter files
2. Parses metadata from HTML
3. Connects to database using db_utils
4. Creates subtopic and inserts all chapters
5. Reports final count

Goal: Build the complete automation workflow.
→ Solution

🎯 What's Next

Chapter 6, the final chapter, covers **Building Your Own Utilities** — how to take the patterns you've learned and create your own reusable scripts for any project. You'll also learn best practices for organizing, documenting, and testing your utilities.

🛠️ Building Your Own Utilities

> *You've learned the core patterns: paths, configuration, databases, and automation. Now it's time to take these patterns and apply them to build reusable utilities for any project. This final chapter teaches best practices for structure, documentation, testing, and sharing your utilities.*

📋 The Utility Template

Every utility you build should follow this structure:

```
my-utility/
├── .env.example              ← Configuration template
├── requirements.txt          ← Python dependencies
├── README.md                 ← Usage instructions
├── utils.py                  ← Core functionality
├── main.py                   ← Entry point
└── tests/
    └── test_utils.py         ← Unit tests
```

✅ Best Practices

1. Separate Concerns

```
# ❌ BAD: Everything in one file
main.py:
  - Load config
  - Connect to database
  - Process files
  - Insert into database
  - Display results

# ✅ GOOD: Separate modules
config.py        → Configuration loading
db_utils.py      → Database operations
file_utils.py    → File operations
main.py          → Orchestration
```

2. Configuration Management

```
# ✅ GOOD: All config in one place
config.py:
  - Load from .env
  - Set defaults
  - Validate
  - Return structured config
```

```
# ✅ GOOD: .env.example for documentation
.env.example:
  DB_HOST=localhost
  DB_USER=your_username
  DB_PASS=your_password
```

3. Error Handling

```
# ✅ GOOD: Graceful failure
try:
    # Main operation
except SpecificError as e:
    print(f"❌ {e}")
    return False
finally:
    # Cleanup
```

4. Type Hints

```
def process_file(filepath: str, config: dict) -> bool:
    """Process a file and return success status."""
    pass

def find_files(directory: str) -> list[str]:
    """Find all files in directory."""
    pass
```

5. Documentation

```
# README.md should explain:

1. What it does (1 paragraph)
2. Installation (pip install -r requirements.txt)
3. Setup (.env configuration)
4. Usage (how to run it)
5. Examples (real-world usage)
6. Troubleshooting (common issues)
```

🔄 Adapting Patterns to New Projects

From TypeScript Course to Any Course

Component	TypeScript Example	Generic Pattern
Find Files	glob("ts1-*.html")	glob("{prefix}-*.{ext}")
Parse Metadata	Extract from <h2>	Extract key data from file
Database Entity	TypeScript Fundamentals	Any course name

Insert Operation	insert_chapter_with_content()	insert_<item>()

Real-World Examples

Example 1: Import Blog Posts

- Find: blog-*.md files
- Parse: Extract title, date, content
- Database: Insert into posts table

Example 2: Sync User Data

- Find: users.csv file
- Parse: Read CSV rows
- Database: Update users table

Example 3: Process Images

- Find: *.jpg files in directory
- Parse: Extract metadata (size, dimensions)
- Database: Insert into images table

🧪 Testing Your Utility

Unit Tests

```
import pytest
from utils import parse_metadata, find_files

def test_parse_metadata():
    """Test metadata parsing."""
    result = parse_metadata("chapter-1.html")
    assert result['number'] == 1
    assert result['title'] is not None

def test_find_files(tmp_path):
    """Test file discovery."""
    # Create test files
    (tmp_path / "chapter-1.html").touch()
    (tmp_path / "chapter-2.html").touch()

    # Test finding them
    files = find_files(tmp_path)
    assert len(files) == 2
```

Integration Tests

```
def test_full_workflow(tmp_path, test_db):
    """Test complete workflow."""
    # Create test files
    # Run main workflow
    # Verify database
    pass
```

🎯 Common Utility Patterns

Pattern 1: File Discovery + Database Insert
Find files → Parse → Insert into database

Pattern 2: Data Transformation
Read source → Transform → Write result

Pattern 3: API Sync
Fetch from API → Parse → Store in database

Pattern 4: CSV/Excel Import
Read file → Parse rows → Insert records

Pattern 5: Batch Processing
Find items → Process each → Report results

📝 Documentation Checklist

- ✅ README.md with overview and usage
- ✅ .env.example showing all config options
- ✅ requirements.txt with all dependencies
- ✅ Type hints on all functions
- ✅ Docstrings on modules and functions
- ✅ Error messages that explain what went wrong
- ✅ Examples of how to use the utility
- ✅ Troubleshooting section for common issues

💻 Final Challenges

Challenge 1: Create a Blog Post Importer

Build a utility that:

1. Finds all blog-*.md files
2. Parses title and date from front matter
3. Inserts into database
4. Reports count

Goal: Apply patterns to a new domain (blog posts vs. courses).
→ Solution

Challenge 2: Data Transformation Utility

Build a utility that:

1. Reads a CSV file
2. Transforms data (cleanup, validation)
3. Writes to database
4. Generates report

Goal: Practice the data transformation pattern.
→ Solution

Challenge 3: Reusable Utility Package

Build a complete utility package with:

1. Main script with proper structure
2. Configuration management
3. Error handling and logging
4. Unit tests
5. Complete documentation

Goal: Build a production-ready utility you could share.
→ Solution

🎓 Course Conclusion

🏆 You've Mastered

- ✅ File system navigation with Path and glob
- ✅ Configuration management with .env
- ✅ Database connections and operations
- ✅ Building reusable utility modules
- ✅ Complete automation workflows
- ✅ Best practices for production code

What You Can Build Now

With these patterns, you can build utilities for:

- 📚 Course content management systems
- 📝 Blog post importers
- 📊 Data migration tools
- 🔄 API sync utilities
- 📁 Batch file processors
- 🗄️ Database administration tools
- 🚀 CI/CD automation scripts

Next Steps

1. **Pick a project:** What do you want to automate?
2. **Follow the template:** Use the structure from this course
3. **Test thoroughly:** Write unit and integration tests
4. **Document well:** Help future users understand it
5. **Share it:** Put it on GitHub so others can use it

> 💡 **Remember**
>
> The patterns you've learned work for any project. The details change (different file types, different databases, different APIs), but the structure stays the same. Master the patterns, adapt them to your needs, and you can build anything!