Section 10: Working with External Libraries

This section introduces you to the world of external libraries in Python. These libraries extend Python's capabilities and allow you to perform complex tasks more easily. We'll cover virtual environments, package management, working with APIs, regular expressions, and asynchronous programming.

Virtual Environments & Package Management

As you start working on more Python projects, you'll likely use different versions of libraries. Virtual environments help isolate project dependencies, preventing conflicts between different projects.

Virtual Environments:

A virtual environment is a self-contained directory that contains its own Python interpreter and libraries. This means that libraries installed in one virtual environment won't interfere with libraries in another.

Creating a virtual environment (using `venv` - recommended):

```
python3 -m venv my_env  # Creates a virtual environment named "my
```

Activating the virtual environment:

- **Windows:** `my_env\Scripts\activate`
- **macOS/Linux:** `source my_env/bin/activate`

Once activated, you'll see the virtual environment's name in your terminal prompt (e.g., `(my_env)`).

Package Management (using `pip`):

`pip` is Python's package installer. It's used to install, upgrade, and manage external libraries.

Installing a package:

```
pip install requests  # Installs the "requests" library
pip install numpy==1.20.0 # Installs a specific version
```

Listing installed packages:

```
pip list
```

Upgrading a package:

```
pip install --upgrade requests
```

Uninstalling a package:

```
pip uninstall requests
```

Generating a requirements file:

A `requirements.txt` file lists all the packages your project depends on. This makes it easy to recreate the environment on another machine.

```
pip freeze > requirements.txt  # Creates the requirements file
pip install -r requirements.txt  # Installs packages from the fil
```

Deactivating the virtual environment:

```
deactivate
```

Requests Module - Working with APIs

The `requests` library simplifies making HTTP requests. This is essential for interacting with web APIs (Application Programming Interfaces).

```
import requests

url = "https://api.github.com/users/octocat"  # Example API endpo
```

```
response = requests.get(url)

if response.status_code == 200:
    data = response.json()  # Parse the JSON response
    print(data["name"])  # Access data from the JSON
else:
    print(f"Error: {response.status_code}")

# Making a POST request (for sending data to an API):
# data = {"key": "value"}
# response = requests.post(url, json=data)  # Sends data as JSON

# Other HTTP methods: put(), delete(), etc.
```

Regular Expressions in Python

Regular expressions (regex) are powerful tools for pattern matching in strings. Python's `re` module provides support for regex.

```
import re

text = "The quick brown fox jumps over the lazy dog."

# Search for a pattern
match = re.search("brown", text)
if match:
    print("Match found!")
    print("Start index:", match.start())
    print("End index:", match.end())

# Find all occurrences of a pattern
matches = re.findall("the", text, re.IGNORECASE)  # Case-insensit
print("Matches:", matches)

# Replace all occurrences of a pattern
new_text = re.sub("fox", "cat", text)
print("New text:", new_text)

# Compile a regex for efficiency (if used multiple times)
```

```
pattern = re.compile(r"\b\w+\b")  # Matches whole words
words = pattern.findall(text)
print("Words:", words)
```

Lets understand the regex pattern `re.compile(r"\b\w+\b")` used in the above code: | Part | Meaning | |——|————| | `\b` | **Word boundary** (ensures we match full words, not parts of words) | | `\w+` | **One or more word characters** (letters, digits, underscores) | | `\b` | **Word boundary** (ensures we match entire words) |

Multithreading

These techniques allow your programs to perform multiple tasks concurrently, improving performance.

Multithreading (using `threading` module):

Multithreading is suitable for I/O-bound tasks (e.g., waiting for network requests).

```
import threading
import time

def worker(num):
    print(f"Thread {num}: Starting")
    time.sleep(2)  # Simulate some work
    print(f"Thread {num}: Finishing")

threads = []
for i in range(3):
    thread = threading.Thread(target=worker, args=(i,))
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()  # Wait for all threads to finish

print("All threads completed.")
```