levi nine

Modules, Packages, Virtual Environments

Summary

- File system operations with os
- Command-line arguments with sys
- Measuring and manipulating time with time
- Generating random values with random
- Mathematical computations with math
- Data serialization with JSON
- Introduction to regular expressions with re
- Modules
- Packages
- Virtual Environments

levi nine

File system operations with os

File System Operations with os

The os module allows your Python script to interact directly with the underlying operating system, providing functions to manage files and directories.

Examples:

- Automatically organizing downloaded files by creating specific folders and moving files into them.
- Checking if a file exists before attempting to read or write to it, preventing errors.

```
import os
# Create a new directory
os.mkdir("my_new_folder")
# Check folder exists
print(os.path.isdir("my_new_folder"))       # True
```

```
import os
# List contents of the current directory
print(f"Files in current directory: {os.listdir('.')}")   # Files in current
                    directory: ['.config', 'my_new_folder', 'sample_data']
```

levi nine

Command-line arguments with sys

Command-Line Arguments with sys

The sys module provides access to system-specific parameters and functions, including the ability for your script to read arguments passed to it from the command line.

Examples:

- Script that takes the input file path directly from the command line
- Script that accepts flags or options, like --verbose or --output_format, to modify its behavior.
- Displaying a "usage" message if the user doesn't provide the expected number of arguments when running your script.

```
import sys
# Access all command-line arguments
sys.argv = ["my_script.py", "argument1"]

if len(sys.argv) > 1:
    print(f"First argument (excluding script name): {sys.argv[1]}")
# Exit the script with an error code
elif len(sys.argv) < 2:
    print("Usage: python my_script.py {sys.argv[0]}")
    sys.exit(1) # Exit with a non-zero status code indicating an error
```

Response:

```
# First argument (excluding script name): argument1
```

Command-Line Arguments with sys

```
import sys
# Access all command-line arguments
sys.argv = ["my_script.py"]
if len(sys.argv) > 1:
    print(f"First argument (excluding script name): {sys.argv[1]}")
# Exit the script with an error code
elif len(sys.argv) < 2:
    print(f"Usage: python {sys.argv[0]}")
    sys.exit(1) # Exit with a non-zero status code indicating an error
```

```
# Output: Usage: python my_script.py
```

Measuring and manipulating time with time

Measuring and manipulating time with time

The time module offers various functions for working with time, including pausing script execution, getting the current time, and measuring performance.

Examples:

- Adding a delay in a game or simulation to control the flow of events (e.g., time.sleep(3)).
- Measuring the execution time of a specific block of code or a function to identify performance bottlenecks.
- Logging events with precise timestamps to create an audit trail or debug sequence.

```
import time
# Measure execution time of a task
start_time = time.time()
print(f"Task started at: {start_time}, or maybe I can say
{time.ctime(start_time)}, or {datetime.fromtimestamp(start_time)}")
for _ in range(1000000):  pass
end_time = time.time()
print(f"Task took {end_time - start_time:.4f} seconds.")
```

```
# Task started at: 1753167059.0344357, or maybe I can say
Tue Jul 22 06:50:59 2025, or 2025-07-22 06:50:59.034436
# Task took 0.0429 seconds.
```

Measuring and manipulating time with time

```
import time
current_timestamp = time.time()
dt_object = datetime.fromtimestamp(current_timestamp)
formatted_full = dt_object.strftime("%A, %B %d, %Y %I:%M:%S
%p")
print(f"Full formatted: {formatted_full}")

formatted_date = dt_object.strftime("%Y-%m-%d")
print(f"Date only: {formatted_date}")

formatted_time = dt_object.strftime("%H:%M:%S")
print(f"Time only (24-hour): {formatted_time}")

formatted_custom = dt_object.strftime("%b %d, %y - %I:%M %p")
print(f"Custom format: {formatted_custom}")
```

```
# Full formatted: Tuesday, July 22, 2025
06:51:02 AM
# Date only: 2025-07-22
# Time only (24-hour): 06:51:02
# Custom format: Jul 22, 25 - 06:51 AM
```

levi nine

Generating random values with random

Generating random values with random

The random module provides functions for generating pseudo-random numbers, which are essential for simulations, games, and security applications.

Examples:

- Simulating a dice roll or a coin flip in a game.
- Shuffling a list of items, such as drawing cards from a deck or randomizing quiz questions.
- Generating random passwords or temporary IDs for security purposes.

```python
import random
# Generate a random integer between 1 and 10 (inclusive)
random_number = random.randint(1, 10)
print(f"Random number: {random_number}")  # Output: Random number: 9
```

```python
import random
# Shuffle a list in place
my_list = ['apple', 'banana', 'cherry']
random.shuffle(my_list)
print(f"Shuffled list: {my_list}") # Output: Shuffled list: ['banana', 'cherry', 'apple']
```

levi nine

Mathematical computations with math

Mathematical computations with math

The math module provides access to common mathematical functions and constants, enabling complex numerical operations.

Examples:

- Calculating the square root of a number.
- Performing trigonometric calculations (sine, cosine, tangent) for engineering or graphics applications.
- Working with logarithmic or exponential functions in scientific computations.

```
import math
# Calculate square root and floor/ceil
print(f"Square root of 25: {math.sqrt(25)}")  # Output: Square root of 25: 5.0
print(f"Ceiling of 4.2: {math.ceil(4.2)}")    # Output: Ceiling of 4.2: 5

# Use trigonometric functions
angle_radians = math.pi / 2 # 90 degrees
print(f"Sine of pi/2: {math.sin(angle_radians)}")  # Output: Sine of pi/2: 1.0
```

levi nine

Data serialization with JSON

Data serialization with JSON

The json module allows you to convert Python data structures (like dictionaries and lists) into JSON strings and vice-versa, facilitating data exchange and storage.

Examples:

- Saving game progress or user settings to a file in a human-readable and easily parsable format.
- Exchanging structured data with web services (APIs), as JSON is a widely used format for web communication.
- Storing configuration data for an application, where settings can be loaded from a JSON file.

```python
import json
# Convert Python dictionary to JSON string
data = {"name": "Bob", "age": 25, "is_student": False}
json_output = json.dumps(data, indent=4) # Pretty print
print(f"JSON string:\n{json_output}")
```

```
Output:
JSON string:
{
    "name": "Bob",
    "age": 25,
    "is_student": false
}
```

Data serialization with JSON

```
import json
# Convert JSON string back to Python dictionary
json_string = '{"city": "New York", "population": 8000000}'
python_dict = json.loads(json_string)
print(f"Python dictionary: {python_dict}")
print(f"City: {python_dict['city']}")

Output:
Python dictionary: {'city': 'New York', 'population': 8000000}
City: New York
```

levi nine

Introduction to regular expressions with re

Introduction to regular expressions using re

The re module provides powerful tools for pattern matching and manipulation of strings using regular expressions, enabling sophisticated text processing.

Examples:

- Validating input formats, such as ensuring an email address or phone number follows a specific pattern.
- Extracting specific information from unstructured text, like finding all URLs or dates in a document.
- Replacing all occurrences of a certain word or phrase in a text with another, even if the word has slight variations.

```python
import re
# Search for a pattern in a string
text = "The quick brown fox jumps over the lazy dog."
match = re.search(r"quick (.*) fox", text)
if match:
    print(f"Found: {match.group(0)}")
    print(f"Captured group: {match.group(1)}")
```

```
Output:
Found: quick brown fox
Captured group: brown
```

```python
import re
# Replace all occurrences of a pattern
text = "Red is my favorite color."
new_text = re.sub(r"Red", "Blue", text, flags=re.IGNORECASE)
print(f"Modified text: {new_text}")
```

```
Output:
Modified text: Blue is my favorite colour.
```

levi nine

Modules

Modules

A module is a single Python file (.py) containing reusable code, such as functions, classes, and variables, that can be imported and used in other Python scripts. They help organize code into logical, manageable units.

Examples:

- Creating a utils.py file to store common helper functions (e.g. format_date, validate_email) that can be imported into multiple main scripts.
- Separating the data processing logic into data_processor.py and the reporting logic into report_generator.py within a larger project.
- Distributing a set of related mathematical functions in a my_math_lib.py file for others to use.

Modules

```python
# Example: my_module.py
def greet(name):
    return f"Hello, {name} from my_module!"

# Example: main_script.py
import my_module
print(my_module.greet("Alice"))
```

```
Output:
Hello, Alice from my_module!
```

```python
# Example: calculations.py
PI = 3.14159
def area_circle(radius):
    return PI * radius**2

# Example: another_script.py
from calculations import area_circle, PI
print(f"Area of circle with radius 5:
{area_circle(5)}")
print(f"Value of PI: {PI}")
```

```
Output:
Area of circle with radius 5: 7853975
Value of PI: 3.14159
```

levi nine

Packages

Packages

A package is a way to organize related modules into a directory hierarchy, typically marked by an __init__.py file, allowing for more structured and scalable code organization.

Examples:

- Building a large web application where different functionalities (e.g., authentication, database, api) are separated into subdirectories, each as a package.
- Creating a reusable library with multiple sub-components, such as a data_analysis package containing plotting and statistics modules.
- Structuring a complex scientific simulation into packages for physics_models, data_io, and visualization.

Packages

```
Example: my_app/
├── __init__.py
├── models/
│   └── __init__.py
│   └── user.py
│       class User: pass
└── views/
    └── __init__.py
    └── auth_view.py
        def render_login(): return "Login Page"

# Example: run.py
from my_app.models.user import User
from my_app.views.auth_view import render_login
user = User()
print(render_login())
```

levi nine

Virtual Environments

Virtual Environments

A virtual environment is a self-contained directory that holds a specific Python interpreter and its own set of installed packages, isolating project dependencies to prevent conflicts.

Examples:

- Working on two different Python projects, where one requires requests library version 2.20 and the other requires version 2.28, without them clashing.
- Ensuring that a project's dependencies are precisely defined and can be easily replicated on another developer's machine or a deployment server.
- Experimenting with new library versions for one project without affecting the stability of other ongoing projects.

Packages

```
# Create a virtual environment named 'my_env'
python3 -m venv my_env

# Activate the environment (Linux/macOS)
source my_env/bin/activate

(my_env) $ pip install flask

# Activate the environment (Windows)
my_env\Scripts\activate

(my_env) > pip freeze > requirements.txt

# Deactivate the environment
deactivate
```

Questions