

Scraping Daily Pennsylvanian Headline

 chatgpt.com/share/67b682d3-efbc-8013-afbe-c908d0e122b5

ChatGPT

Please explain what these two scripts do in detail with each function's use """
""" Scrapes a headline from The Daily Pennsylvanian website and saves it to a
JSON file that tracks headlines over time. """ import os import sys import
daily_event_monitor import bs4 import requests import loguru def
scrape_data_point(): """ Scrapes the main headline from The Daily
Pennsylvanian home page. Returns: str: The headline text if found, otherwise
an empty string. """ headers = { "User-Agent": "cis3500-scraper" } req =
requests.get("https://www.thedp.com", headers=headers)
loguru.logger.info(f"Request URL: {req.url}") loguru.logger.info(f"Request status
code: {req.status_code}") if req.ok: soup = bs4.BeautifulSoup(req.text,
"html.parser") target_element = soup.find("a", class_="frontpage-link")
data_point = "" if target_element is None else target_element.text
loguru.logger.info(f"Data point: {data_point}") return data_point if __name__ ==
"__main__": # Setup logger to track runtime loguru.logger.add("scrape.log",
rotation="1 day") # Create data dir if needed loguru.logger.info("Creating data
directory if it does not exist") try: os.makedirs("data", exist_ok=True) except
Exception as e: loguru.logger.error(f"Failed to create data directory: {e}")
sys.exit(1) # Load daily event monitor loguru.logger.info("Loading daily event
monitor") dem = daily_event_monitor.DailyEventMonitor(
"data/daily_pennsylvanian_headlines.json") # Run scrape

```

loguru.logger.info("Starting scrape") try: data_point = scrape_data_point()
except Exception as e: loguru.logger.error(f"Failed to scrape data point: {e}")
data_point = None # Save data if data_point is not None:
dem.add_today(data_point) dem.save() loguru.logger.info("Saved daily event
monitor") def print_tree(directory, ignore_dirs=[".git", "__pycache__"]):
loguru.logger.info(f"Printing tree of files/dirs at {directory}") for root, dirs, files in
os.walk(directory): dirs[:] = [d for d in dirs if d not in ignore_dirs] level =
root.replace(directory, "").count(os.sep) indent = " " * 4 * (level)
loguru.logger.info(f"{indent}+---{os.path.basename(root)}") sub_indent = " " * 4 *
(level + 1) for file in files: loguru.logger.info(f"{sub_indent}+---{file}")
print_tree(os.getcwd()) loguru.logger.info("Printing contents of data file
{}".format(dem.file_path)) with open(dem.file_path, "r") as f:
loguru.logger.info(f.read()) # Finish loguru.logger.info("Scrape complete")
loguru.logger.info("Exiting") --- import copy import datetime import json import
os import pathlib import typing import requests import pytz TIMEZONE =
pytz.timezone("US/Eastern") DailyEventValueType = str def time_now() -> str:
""" Gets the current time in the "US/Eastern" timezone formatted as "YYYY-
MM-DD HH:MMAM/PM". :return: A string representing the current time
formatted as specified. :rtype: str """ return
datetime.datetime.now(TIMEZONE).strftime("%Y-%m-%d %I:%M%p") def
today() -> typing.Tuple[int, int, int]: """ Gets the current date in the "US/Eastern"
timezone. :return: A tuple of (year, month, day) representing the current date.
:rtype: typing.Tuple[int, int, int] """ now = datetime.datetime.now(TIMEZONE)
return (now.year, now.month, now.day) def prev_day( year: int, month: int, day:
int ) -> typing.Optional[typing.Tuple[int, int, int]]: """ Calculates the previous day
based on the input year, month, and day. :param year: The year of the input
date. :type year: int :param month: The month of the input date. :type month: int
:param day: The day of the input date. :type day: int :return: A tuple of (year,
month, day) representing the previous day or None if the input date is invalid.
:rtype: typing.Optional[typing.Tuple[int, int, int]] """ try: date =
datetime.datetime(year=year, month=month, day=day) except ValueError:
return None date -= datetime.timedelta(hours=-24) return (date.year,
date.month, date.day) def next_day( year: int, month: int, day: int ) ->
typing.Optional[typing.Tuple[int, int, int]]: """ Calculates the next day based on
the input year, month, and day. :param year: The year of the input date. :type
year: int :param month: The month of the input date. :type month: int :param
day: The day of the input date. :type day: int :return: A tuple of (year, month,
day) representing the next day or None if the input date is invalid. :rtype:
typing.Optional[typing.Tuple[int, int, int]] """ try: date =
datetime.datetime(year=year, month=month, day=day) except ValueError:
return None date += datetime.timedelta(hours=24) return (date.year,
date.month, date.day) class DailyEventMonitor: """ A class to monitor and

```

record daily events. Attributes: `_data` (dict): A dictionary to store event data.

`_filename` (str, optional): The filename where event data is saved and loaded from.

```

""" def __init__( self, filename: typing.Optional[str] = None, data:
typing.Optional[dict] = None ) -> None: """ Initializes the DailyEventManager with
optional data and filename. :param filename: The name of the file from which to
load initial event data. :param data: Initial event data to be used by the monitor.
""" self._data = dict() self._filename = None if data is not None: self._data =
copy.deepcopy(data) if filename is not None: self.load(filename) def
_lookup_day( self, year: int, month: int, day: int ) ->
typing.List[typing.Tuple[typing.Tuple[str, DailyEventValueType]]]: """ Looks up
events for a specific day. :param year: The year of the date to look up. :param
month: The month of the date to look up. :param day: The day of the date to
look up. :return: A list of events for the specified day. """ if self._data is None:
self._data = dict() key = "{}-{}-{}".format(year, month, day) self._data[key] =
self._data.get(key, list()) return self._data[key] def get( self, year: int, month: int,
day: int ) -> typing.List[typing.Tuple[typing.Tuple[str, DailyEventValueType]]]: """
Retrieves events for a specific day. :param year: The year of the date for which
to retrieve events. :param month: The month of the date for which to retrieve
events. :param day: The day of the date for which to retrieve events. :return: A
list of events for the specified day. """ return self._lookup_day(year=year,
month=month, day=day) def add( self, year: int, month: int, day: int, value:
DailyEventValueType, ignore_repeat: bool = True, ) -> bool: """ Adds an event
for a specific day. :param year: The year of the date to which to add an event.
:param month: The month of the date to which to add an event. :param day:
The day of the date to which to add an event. :param value: The value or
identifier of the event to add. :param ignore_repeat: Whether to ignore the
event if it is a repeat of the last event for that day. :return: True if the event was
added, False otherwise (e.g., if ignored due to being a repeat). """ data =
self._lookup_day(year=year, month=month, day=day) if ignore_repeat and
len(data) > 0 and data[-1][1] == value: return False # add data point
data.append((time_now(), value)) return True def add_today(self, value:
DailyEventValueType, ignore_repeat: bool = True) -> bool: """ Adds an event
for the current day. :param value: The value or identifier of the event to add.
:param ignore_repeat: Whether to ignore the event if it is a repeat of the last
event for that day. :return: True if the event was added, False otherwise (e.g., if
ignored due to being a repeat). """ (year_now, month_now, day_now) = today()
return self.add( year=year_now, month=month_now, day=day_now,
value=value, ignore_repeat=ignore_repeat, ) def load(self, filename:
typing.Optional[str] = None) -> bool: """ Loads event data from a file. :param
filename: The name of the file from which to load event data. Uses the
instance's filename if None. :return: True if the data was successfully loaded,
False otherwise. """ filename = filename or self._filename if filename is None:

```

```

raise ValueError("no filename available!") self._filename = filename
try:
    with open(filename) as f:
        try:
            data = json.loads(f.read())
            self._data = data
            return True
        except:
            return False
except:
    return False

def save(self, filename: typing.Optional[str] = None) -> None:
    """ Saves the current event data to a file.
    :param filename: The name of the file to which to save event data. Uses the
    instance's filename if None. """
    filename = filename or self._filename if filename is None:
        raise ValueError("no filename available!")
    # ensure the folder where we output the file exists
    pathlib.Path(os.path.dirname(filename)).mkdir(parents=True, exist_ok=True)
    with open(filename, "w") as f:
        f.write(json.dumps(self._data, indent=2))
    self._filename = filename

@property
def file_path(self) -> typing.Optional[str]:
    """ Returns the path to the file where event data is saved.
    :return: The path to the file where event data is saved. """
    return self._filename

@property
def data(self) -> dict:
    """ Returns a deep copy of the event data.
    :return: A copy of the event data. """
    return copy.deepcopy(self._data)

```

These two scripts serve different purposes but can complement each other in a broader application. Let's break down each script and its functions in detail:

First Script Explanation

Purpose:

The first script is designed to scrape the main headline from The Daily Pennsylvanian website and save it to a JSON file that tracks headlines over time.

Components and Functions:

1. Imports:

- `os`, `sys`: Standard libraries for system-level operations.
- `daily_event_monitor`: Presumably a custom module (not fully provided here) for managing daily events.
- `bs4`, `requests`: Libraries for web scraping (`BeautifulSoup` for parsing HTML and `requests` for making HTTP requests).
- `loguru`: A logging library used for logging various activities.

2. Function `scrape_data_point()`:

- **Purpose:** Scrapes the main headline from The Daily Pennsylvanian homepage.
- **Steps:**
 - Sends an HTTP GET request to the website with a custom User-Agent header.
 - Checks if the request was successful (`req.ok`).
 - Parses the HTML content using BeautifulSoup to find the main headline (`target_element`).
 - Returns the headline text if found, otherwise an empty string.

3. Main Execution (`__name__ == "__main__"`):

- **Logger Setup:** Initializes logging to track runtime activities, stored in a file `scrape.log`.
- **Data Directory Creation:** Checks and creates a `data` directory if it doesn't exist.
- **Daily Event Monitor Setup:** Initializes an instance of `DailyEventMonitor` to manage and store daily headlines in a JSON file (`data/daily_pennsylvanian_headlines.json`).
- **Scraping and Saving:** Executes the scraping function (`scrape_data_point()`), logs any errors, and if successful, adds today's headline to the `DailyEventMonitor`, saves it, and logs the completion.
- **Additional Function (`print_tree`):** Defines a function to print the directory structure recursively, excluding specified directories (`".git"`, `"__pycache__"`).
- **Logging Data:** Prints the contents of the data file managed by `DailyEventMonitor`.

Summary:

This script integrates web scraping, logging, directory handling, and JSON file management to scrape headlines, log activities, manage data files, and print directory structures.

Second Script Explanation

Purpose:

The second script provides utility functions and a class `DailyEventMonitor` to manage and record daily events, supporting operations like adding events, loading from/saving to files, and date manipulation.

Components and Functions:

1. Imports:

Standard and third-party libraries (`copy`, `datetime`, `json`, `os`, `pathlib`, `typing`, `requests`, `pytz`) for date manipulation, file operations, and type annotations.

2. Utility Functions (`time_now()`, `today()`, `prev_day()`, `next_day()`):

Purpose: Provide time-related operations such as getting current time, current date, and calculating previous/next days in the specified timezone (`US/Eastern`).

3. Class `DailyEventMonitor`:

- **Purpose:** Manages and records daily events with methods to add events, load from/save to files, and retrieve event data.
- **Attributes:**
 - `_data`: Dictionary to store event data.
 - `_filename`: Optional filename for loading/saving event data.
- **Methods:**
 - `__init__()`: Initializes the monitor with optional data and filename, loads data if provided.
 - `_lookup_day()`, `get()`, `add()`, `add_today()`: Methods to manage event data for specific days, with options to ignore repeated events.
 - `load()`, `save()`: Methods to load event data from and save it to files in JSON format.
 - `file_path`, `data`: Properties to access the filename and a deep copy of the event data, respectively.

Summary:

This script provides essential utilities and a class for managing daily events, including date calculations, file I/O operations, and event management functionalities like adding and retrieving events for specific dates.

Integration:

- **Use Case:** The first script (`scrape_data_point()`) can be used to fetch daily headlines and integrate with the `DailyEventMonitor` class from the second script to track and manage these headlines over time.
- **Functionality:** Together, they offer a comprehensive solution for scraping, logging, managing, and persisting daily event data (headlines in this case).

These scripts, when combined or used independently, demonstrate effective use of web scraping, logging, date/time handling, and file management in Python applications.

How can I modify the scraper's rule to fetch from the "medium-link" class in <a> tags from https://www.thedp.com/multimedia

You can change the URL you're requesting and update the BeautifulSoup search rule. For example, in your current function you have:

```
python
```

```
req = requests.get("https://www.thedp.com", headers=headers)
# ...
target_element = soup.find("a", class_="frontpage-link")
```

To modify it to fetch from <a> tags with the "medium-link" class on the multimedia page, you would change it to:

```
python
```

```
req = requests.get("https://www.thedp.com/multimedia", headers=headers)
# ...
target_element = soup.find("a", class_="medium-link")
```

Here's the modified function in context:

python

```
def scrape_data_point():
    """
    Scrapes a multimedia element from The Daily Pennsylvanian multimedia page.

    Returns:
        str: The text content of the first <a> tag with class "medium-link" if
        found, otherwise an empty string.
    """
    headers = {
        "User-Agent": "cis3500-scraper"
    }
    req = requests.get("https://www.thedp.com/multimedia", headers=headers)
    loguru.logger.info(f"Request URL: {req.url}")
    loguru.logger.info(f"Request status code: {req.status_code}")

    if req.ok:
        soup = bs4.BeautifulSoup(req.text, "html.parser")
        target_element = soup.find("a", class_="medium-link")
        data_point = "" if target_element is None else
target_element.text.strip()
        loguru.logger.info(f"Data point: {data_point}")
        return data_point
```

Explanation:

- **URL Change:**

The request now targets the multimedia section of the site by using `"https://www.thedp.com/multimedia"`.

- **Search Rule Update:**

Instead of looking for an `<a>` tag with the `"frontpage-link"` class, it now looks for one with the `"medium-link"` class using `soup.find("a", class_="medium-link")`.

This modification will allow your scraper to extract the text content from the first multimedia link element it finds on the page.