Table of Contents

Homework: Timeline of events

Background

For this assignment you will build a text-based timeline by reading events from a file and sorting them in chronological order. Your code should be able to handle dates in a variety of formats.

File format

Your program should be able to read in a text file where each line consists of a date, a tab character, and a description of the event.

Date format

Dates in the input file do not necessarily follow one single format, but they adhere to the following rules:

- The year must always be four digits.
- The month may be a number or a word. If the month is a word, it may or may not be abbreviated; abbreviations must consist of the first three letters of the month name. If the month is a word, the first letter will be capitalized and subsequent letters will be lower case.
- The day must be expressed in digits, optionally followed by letters (e.g., "1st", "2nd", "3rd", "4th").
- Days and numeric months less than 10 may optionally begin with a 0.
- The day never comes directly after the year (so dates like "2014 17 Jan" or "2014/17/1" are disallowed).
- When the month is a number, the day must follow the month; the year may come at the beginning ("2014-01-17") or end ("01-17-2014") of the date.
- There is always some kind of separation between the parts of a date (so, for example, dates such as 01172014 are not allowed).

Some examples of valid dates include:

- January 17th, 2014
- Jan. 17, 2014
- Jan 17 2014
- 17 January 2014
- 1/17/2014
- 2014-01-14

Magic methods for comparison operators

Python defines a set of special method names that are often referred to as "magic methods" or "dunder methods". These methods are rarely called directly, but they define how instances of a class will behave with certain features of Python. We have already seen a few of these:

- `__init__()` determines how an instance will be initialized when it is first created
- `__repr__()` returns a formal string representation of an object that is used when the object is passed to the `repr()` function
- `__str__()` returns an informal string representation of an object that is used when the object is printed or converted to a string (e.g., using the `str()` function)

For this assignment, we will also define some of the magic methods that define the behavior of comparison operators:

- `__lt__()` defines the behavior of the `<` operator; note that sorting in Python is based on this operator
- `__gt__()` defines the behavior of the `>` operator
- `__eq__()` defines the behavior of the `=` operator
- `__ne__()` defines the behavior of the `!=` operator
- `__le__()` defines the behavior of the `<=` operator
- `__ge__()` defines the behavior of the `>=` operator

Once you have defined the behavior of the `==` operator and one of the operators `<`, `>`, `<=`, or `>=`, it's logically possible to derive the behavior of the other operators. `functools.total_ordering` is a class decorator that does this. To use it, import functools (which is part of the Python standard library), then write `@functools.total_ordering` on the line before your class header.
Here is an example class that defines comparison operators with the help of `functools.total_ordering`. The class models elephants and compares them based on their age. It can also compare `Elephant` objects to ints and floats.

```python
import functools


@functools.total_ordering
class Elephant:
    """An elephant object with an age.

    Attributes:
        age (int): the age of the elephant, in years.
    """
    def __init__(self, age):
        """Initialize a new elephant object."""
        self.age = age

    def __lt__(self, other):
        """Determine if this elephant is younger than other.

        Other can be another Elephant object or a number.

        Args:
            other (Elephant, int, or float): value to compare to self.

        Returns:
            bool: True if self is younger than other.
        """
        if isinstance(other, Elephant):
            return self.age < other.age
        elif isinstance(other, int) or isinstance(other, float):
            return self.age < other
        else:
            raise TypeError(f"'<' not supported between instances of 'Elephant'"
                            f" and {repr(type(other))}")

    def __eq__(self, other):
        """Determine if this elephant is the same age as other.

        Other can be another Elephant object or a number.

        Args:
            other (Elephant, int, or float): value to compare to self.

        Returns:
            bool: True if self is the same age as other.
        """
        if isinstance(other, Elephant):
            return self.age == other.age
        elif isinstance(other, int) or isinstance(other, float):
            return self.age == other
        else:
            raise TypeError(f"'==' not supported between instances of"
                            f" 'Elephant' and {repr(type(other))}")
```

For more information on magic methods, I recommend "A guide to Python's magic methods" by Rafe Kettler and the section "Special method names" from the official Python documentation. For more information on `functools.total_ordering`, see the functools documentation.

Provided files

Data file

A file of random events and dates, `events.tsv`, is provided. The dates occur in a variety of formats. The dates and events came from random Wikipedia articles.

Program template

A program template, `timeline.py`, is provided for you. This template imports some code for you, defines two dictionaries (`MONTHS` and `MONTHS_INVERSE`) for your convenience, and implements the functions `main()` and `parse_args()` as well as an `if __name__ == "__main__":` statement.
`parse_args()` takes a list of command-line arguments and extracts a text file containing events, an optional start date, and an optional end date.
`main()` creates an instance of your `Timeline` class and calls its `print_events()` method to print a timeline in chronological order.

Problem statement

Write a program containing two classes, `Event` and `Timeline`, and one function, `parse_date()`.
`Event` objects represent an event. They have a string representation and can be compared to other `Event` objects, strings containing dates, or tuples consisting of a year, a month, and a day.
`Timeline` objects represent a collection of events. That collection can be filtered to a specific date range, and either the full set of events or a filtered subset can be printed out.
`parse_date()` takes a string containing a date as described above in the Date format section and returns a tuple consisting of a year, a month, and a day.

Instructions

Filename

Your program should be called `timeline.py`. You should implement your program using the provided template file.

`Event` class

Define a class called `Event`. Put the decorator `@functools.total_ordering` above the class header (see Magic methods for comparison operators for an explanation).
Instances of your class will have the following attributes:

- `year` (an integer indicating the year the event took place)
- `month` (an integer indicating the month the event took place)
- `day` (an integer indicating the day the event took place)
- `desc` (a string describing the event)

Define the following methods within this class:

`__init__()` method

Write an `__init__()` method with three parameters:

- `self`
- a date expressed as a string; you can assume that the date follows the rules listed in the Date format section
- a description of the event (a string)

Use your `parse_date()` function to convert the date to a year, a month, and a day. Store these in attributes called `year`, `month`, and `day`, respectively. Store the description in an attribute called `desc`.

`__str__()` method

Write a `__str__()` method with one parameter (`self`). This method should return a string containing the date and the description of the event, in the format `DD MMM YYYY: DESC` (where `DD` is the day, `MMM` is a three-letter abbreviation for the month, `YYYY` is the year, and `DESC` is the description of the event). Below is an example string representation of an event in the expected format.

```
17 Jan 2014: Jerry Brown, governor of California, declares a drought
```

You can look up month abbreviations in the MONTHS dictionary declared near the top of the template.
Overachievers: if you want dates to line up nicely when you print multiple events, you can build your date in an f-string and specify that the day should be right-aligned within a field that is two characters long (see this article for details and examples). This is optional.

Comparison methods

Define an `__eq__()` method and one of the following methods: `__gt__()`, `__lt__()`, `__ge__()`, `__le__()` (see Magic methods for comparison operators for more information). Each method should have two parameters (`self` and another object).
The other object could be any of the following (your methods should be able to handle all of these):

- another instance of Event
- a date expressed as a string (subject to the rules listed in the Date format section)
- a tuple of numbers, where the first number would represent a year, the second number (if any) would represent a month, and the third number (if any) would represent a day; note: the number of items in the tuple may be more than or less than three

If the other object belongs to some other data type, your methods should raise a TypeError.
Otherwise, these methods should return a boolean value. `__eq__()` should return `True` if the two values represent the same date, and `False` otherwise. The other method should return a boolean value consistent with the principle that an earlier date should be considered to be less than a later date.
Hint: it's pretty easy to compare tuples. You may want to build a tuple containing the date components of `self` and a similar tuple containing the date components of `other`.

`Timeline` class

Define a class called `Timeline`. Instances of this class will have an attribute called `events` which will be a list of `Event` objects, which are the events in the timeline.
Define the following methods within this class:

`__init__()` method

Write an `__init__()` method with two parameters:

- `self`
- a string that is a path to a file of events; each line in the file should consist of a date (subject to the rules listed in the Date format section), a tab character, and a description of an event that occurred on that date

The `__init__()` method should convert each line from the file of events into an `Event` object with the appropriate date and description. It should store a list of all these `Event` objects in an attribute called `events`. The list should be sorted chronologically, with older events preceding more recent ones.
Hint: the default sort order of objects is determined by the `<` operator. You don't need to specify a `key` function to use the default sort order.

`events_in_range()` method

Write a method called `events_in_range()` that has three parameters (please use the names specified below):

- `self`
- `start_date`, the earliest date of interest (default value: `None`)
- `end_date`, the latest date of interest (default value: `None`)

`start_date` and `end_date` could take any of the forms allowed by the comparison methods of the `Event` class (see Comparison methods), or they could be `None`. If `start_date` is `None`, set it to the special value `(float("-Inf"),)` (a tuple containing negative infinity). If `end_date` is `None`, set it to `(float("Inf"),)` (a tuple containing positive infinity).
Use a list comprehension to create a list of events from `self.events` that fall within the specified `start_date` and `end_date` (note: any events that fall on `end_date` should be included in the list). Return this list.

`print_events()` method

Write a method called `print_events()` that has the same three parameters as `events_in_range()` (please use the same names; please also use `None` as the default value for both `start_date` and `end_date`).
Call the `events_in_range()` method to get a list of events within the specified date range. Print each event on its own line.
Hint: Event objects have a `__str__()` method, so they should print beautifully if you print them individually. (Printing a list of `Event` objects is a different story.)

`parse_date()` function

Define a function called `parse_date()`. (This is a function, not a method; define it outside of your classes.) This function should take one parameter: a string containing a date, subject to the rules in the Date format section.
Use one or more regular expressions to extract the year, month, and day from the date string. Convert these to integers and return them as a tuple with the year first, then the month, then the day. If you are unable to extract the year, month, and day from the string, raise a `ValueError`.
You may find it helpful to use the `MONTHS_INVERSE` dictionary to help you deal with months expressed as words (or abbreviations of words). Its keys are three-letter month abbreviations ("Jan", "Feb", etc.) and its values are month numbers (1 for January, 2 for February, etc.).

Other instructions

Module restrictions

The provided template imports all or parts of the `argparse`, `functools`, `re`, and `sys` modules. For this assignment, you may not import any other modules or components of any other modules. You may not take any other actions that are functionally equivalent to importing other modules or components of other modules.

Length of individual lines of code

Please keep your lines of code to 80 characters or less. If you need help breaking up long lines of code, please see https://umd.instructure.com/courses/1299872/pages/how-to-break-up-long-lines-of-code.

Docstrings

Please write docstrings for the `Event` and `Timeline` classes, the `events_in_range()` and `print_events()` methods, and the `parse_date()` function. Docstrings were covered in the first week's lecture videos (https://youtu.be/jHTv83PlQYw?t=1415) and revisited in the OOP lecture videos (https://youtu.be/Oq9ssywHMPg). There's an ELMS page about them here: https://umd.instructure.com/courses/1299872/pages/docstrings. Please follow the specified docstring format; I encourage you to pattern your docstrings after the docstrings in the `main()` and `parse_args()` functions.
Docstrings are not comments; they are statements. Python recognizes a string as a docstring if it is the first statement in the body of the method, function, class, or script/module it documents. Because docstrings are statements, the quotation mark at the start of the docstring must align exactly with the start of other statements in the method, function, class, or module.

▶ General instructions for class docstrings

▶ General instructions method and function docstrings

Running your program at the command line

The template is designed to use command-line arguments. To run your program within the VS Code built-in terminal, first make sure you have opened (in VS Code) the folder where your program is saved. If necessary, you can go to the VS Code File menu and select "Open…" on macOS or "Open Folder…" on Windows, and navigate to the directory where your program is.
Then, open the VS Code built-in terminal. Type `python3` (on macOS) or `python` (on Windows) followed by a space, the name of the program, another space, and the name of the CSV file containing a customer's electricity usage over time (a sample file, `events.tsv`, is provided with the assignment). Below is an example. The example assumes that `events.tsv` is in the same directory as your program.

```
python3 timeline.py events.tsv
```

You can filter events using a start date and/or an end date. A start date should be preceded by `-s` and a space; and end date should be preceded by `-e` and a space. Here are some examples:

```
python3 timeline.py events.tsv -s "1/1/1990"
python3 timeline.py events.tsv -e "Jul 1 2015"
python3 timeline.py events.tsv -s "1980-05-01" -e "2010-12-31"
```

Submitting your work

Submit your work using Gradescope. Please upload only `timeline.py` (do not upload any test scripts or CSV files). `timeline.py` will be partially auto-graded by Gradescope. If you are not happy with the results, you may revise your code and resubmit as many times as you like until the deadline.

Extensions

If you need an extension on the deadline, please email the instructor prior to the deadline, indicating when you propose to turn in the assignment. There is no penalty for requesting or receiving an extension, and no explanation is required. Once granted, an extension will not be extended. For full details on the policy for homework extensions are provided in the syllabus.

Grading

This assigment is worth 30 points in the homework category, allocated as follows:
14 points are allocated to automatic tests of your code functionality. 4 points are allocated to automatic tests of your docstrings. The remaining 12 points are awarded based on the degree of completeness of your program and docstrings.

Category	Points	Notes
Automatic tests of code functionality	14	Tests will evaluate instance attributes; return values; side effects; and whether errors are raised when expected
Automatic tests of docstrings	4	Tests will look for existence of docstrings and presence of expected sections in each docstring
Manual evaluation of code completeness	8	Does the code contain the correct classes, methods, and functions? Was a recognizable attempt made to write code to the specifications of the assignment?
Manual evaluation of docstrings	4	Do the docstrings contain the expected information? Do they conform to the expected format?

Academic integrity

This assignment is to be done by you individually, without outside help of any kind (including, but not limited to, help from classmates, tutors, or the internet, including help websites). Disseminating these instructions in whole or in part without written permission of the instructor is considered an infraction of academic integrity.