CSC108H Assignment 3

Due April 6, 2011 at 10 pm

Update 1: Change Summary

- corrected typo in reference to 4.2.1 (non-existent section). It now reads 3.2.1.
- docstring for the `Event.py __cmp__` function updated and clarified to remove ambiguity.
- some test code included in `test_event.py` and `test_database.py`

Overview

Given that task management and scheduling is often a challenge in university, in this assignment you will create a command-line task management and scheduling application called CAL.

CAL will be an application that:

- Shows a nice calendar view of any month or year the user requests
- Maintains a database of `Events` which the user has scheduled, along with their start times and durations
- Displays the user's schedule for any given day
- Reads user input and parses out dates, times and descriptions for events

Code Organization and Requirements

Code is separated into the following three files, which are what you should hand in:

`event.py`	Contains the `Event` class and all its methods
`database.py`	Contains the `Database` class, keeps track of `Events`
`cal.py`	Reads user commands from the console and parses them, retrieving and storing information in the database.

Code requirements:

- **Correctness:** Your functions should perform as specified. Correctness, as measured by our tests, will count for the largest single portion of your marks.
- **Formatting style:** Make sure you follow the Python style guidelines that applied for Assignment 1!
- **Programming style:** Your variable names should be meaningful and your code as simple and clear as possible. You should use efficient methods to store and retrieve information.
- **Commenting**: for each function you write, you should write a docstring that describes its parameters, what the function does, and what is returned by the function, if anything. Supply comments if you believe they are necessary for code readability.
- **Reuse**: If you find yourself repeating a task, you should add a helper function and call that function instead of duplicating the code. This will count for a solid portion of your mark as well due to the possibility for repeating code when parsing.

Part 0: `calendar`, `date` and `datetime`

You can print a nice calendar representation of a month using the `printmonth(y,m)` method in `cal.py`, passing it the year and month you would like to print. You can also print a nice calendar representation of a year using the `printyear(y)` method in `cal.py`. If you are curious, these two methods call methods `pryear()` and `prmonth()` of an instance of the `calendar.TextCalendar` class. This part is already implemented for you, all you have to do is call the appropriate method.

Python uses the `datetime` module to reason about date and time. `datetime` has three useful classes: `date`, `datetime` and `timedelta`.

`datetime.date` has the attributes `year`, `month` and `day` and represents a date. It has a few helpful methods, such as `weekday()` which returns the day of the week (Monday = 0, Sunday = 6), and `today()` which returns a date object representing the current date. A sample date constructor call is `datetime.date(1975,11,25)`

`datetime.datetime` has all of date's attributes and methods, as well as `hour`, `minute`, `second`, and `microsecond`. In this assignment, you won't have to worry about seconds or microseconds, and they are optional arguments. Some of its useful methods include `.now()` which returns a `datetime` object representing the current date and time and `.strftime()` which returns a string representation of the `datetime` according to a supplied formatting string. A sample `datetime` constructor call is `datetime.datetime(1975,11,25,7,05)`

`date` and `datetime` are mutually comparable, with the object depicting the earlier date/time considered the smaller object.

`datetime.timedelta()` is used to create a difference in times/dates between two date or `datetime` objects. It is created with named attributes (`days`, `hours`, `minutes`, etc.) that represent the difference.

For example, `datetime.timedelta(days=3)` can be subtracted from a `date` or `datetime` to yield a date that's three days earlier. `timedelta` objects can be added or subtracted from `date` and `datetime` objects.

Make sure you understand how `datetime`, `date` and `timedelta` work. You can experiment with them in the console, and read more about them online. You will need them later in the assignment.

`datetime.datetime.strftime()` takes a formatting string which can have any of these attributes:

`%Y`	year as a decimal integer (4-digits)
`%m`	padded month as a decimal integer (01-12)
`%d`	padded day of the month as a decimal number (01-31)
`%H`	hour as a decimal integer (24-hour clock) (00-23)
`%M`	minute as a decimal integer (00-59)

Sample usage:

```
>>> x = datetime.datetime(2007,1,7,13,05)
>>> print x.strftime("Year: %Y, Month: %m, Day: %d, Hour: %H, Minute: %M")
Year: 2007, Month: 01, Day: 07, Hour: 13, Minute: 05
```

Part 1: The `Event` Class

The `Event` class is the blueprint for creating `Event` objects. Each `Event` object should keep track of the following:

- a textual description of the event
- a start date and time as a `datetime` object
- an integer duration in minutes (default is 60)
- whether the event is an all-day event (in which case the start time should be 0:00)

Open the file `event.py` and complete the methods outlined there according to their docstring specification.

Part 2: The `Database` Class

At the start of our program, we will create a `Database` instance to keep track of `Events`.

Note: There will only ever be **one** `Database` instance in each run of the program.

You are free to decide how you want to store `Events` in the `Database`, but keep in mind that the database should be able to:

- associate each `Event` with a search key that references its date
- return lists of sorted `Events`
- search `Events` by a substring of their description
- filter `Events` by date

You are required to write a **class docstring** which details the storage strategy you have chosen. TA's will read and mark how well the strategy is described, as well as its robustness. With your database storage strategy, you should demonstrate a good understanding of the various data structures that Python has, and select the best suited ones for each task.

Part 3: The CAL Module – Parsing User Commands

This is the hardest part of the assignment: getting your program to recognize complex user commands.

Google Calendar has an ingenious Quick Add feature which lets users type in natural language expressions like:

> *"Tennis with Mike tomorrow at 2 in the rec centre"*
>
> *"Wisdom teeth out May 25"*
>
> *"Big presentation Wednesday"*

The Quick Add software teases apart event descriptions, location, date and time from these strings and creates appropriate calendar events.

On the surface, this seems pretty impressive, and it is, but it's really based on a very simple idea: the idea of a grammar.

3.1 Grammars

Natural languages have grammars (what we're tortured with in middle school) which govern how sentences are formed in order to be understood and unambiguous. Natural language grammars are fairly fuzzy, and their rules can be bent quite a bit, for instance by tiny (but decidedly battle-worthy) green Jedi Knights, without losing meaning or becoming unintelligible.

Formal (or artificial) languages (such as Python) are less forgiving, and their grammars are much more rigid.

The computational definition of a formal language states that given all possible strings that can be formed using the letters of a particular alphabet, the language accepts a subset of these strings as 'correct' and rejects all others.

To determine whether a string should be accepted or not, every formal language has a grammar: a set of rules for generating correct strings. If a string can be generated using the language's grammar, then it is accepted by the language.

We will use the concept of a grammar to describe what kinds of strings our program will accept as valid commands.

3.2 The CAL Grammar

This section details the possible commands a user may enter. Each command is a string obtained by `raw_input()` in `cal.py`'s main method which consists of several parts, separated by spaces.

Note: For this assignment, we will **only** be testing with valid CAL commands. You don't need to worry about what to do if a command is badly formatted.

3.2.1 The DATE String

The DATE string is a string that denotes a particular date. It is parsed by the `parsedate()` function. DATE strings are not case-sensitive. Your program should take the DATE string given and return which day, month and year it is referring to. A DATE string can be:

`MONTH-dd-yyyy`	where MONTH is the first three characters of a month's name, dd is an integer from 01 to 31 and yyyy is an integer from 1000 to 9999.
`MONTH-dd`	where MONTH is the first three characters of a month's name and dd is an integer from 01 to 31. The year is implied to be this year.

`today`
`tomorrow`
`yesterday`
	which all denote dates relative to today's date in Python.

WEEKDAY	where WEEKDAY starts with one of '`mon`', '`tue`', '`wed`', '`thu`', '`fri`', '`sat`' or '`sun`'. This date returns the closest such weekday in the **future**. A weekday that matches today's weekday is one week in the future. Note that full weekday strings (such as '`saturday`') should also be accepted.

Representative examples of a DATE string:

`today`	the current date in Python
`tomorrow`	a day after the current date
`yesterday`	a day before the current date
`thu`	the closest Thursday in the future from the current date
`sunday`	the closest Sunday in the future from the current date
`Mar-03`	March 3 of the current year
`sep-19-2007`	September 19, 2007

3.2.2 The TIME String

The TIME string is a string that denotes a particular time of day. It is parsed by the parsetime() function. TIME strings are not case-sensitive. Your program should take the TIME string given and return which hour (0-23) and minute (0-59) it denotes.

A TIME string can be:

`12NUMam` `12NUMpm`	where 12NUM is an unpadded integer from 0 to 12.
`24NUMh`	where 24NUM is an unpadded integer from 0 to 23.
`12NUM:MINam` `12NUM:MINpm`	where 12NUM is an integer from 0 to 12, MIN is a padded integer from 00 to 59.
`24NUM:MIN`	where 24NUM is an integer from 0-23, MIN is a padded integer from 00 to 59. This is time on a 24-hour clock.

Representative examples of a TIME string:

`3am`	3 hours and 0 minutes (0300)
`8pm`	20 hours and 0 minutes (2000)
`15h`	15 hours and 0 minutes (1500)
`21:05`	21 hours and 5 minutes (2105)
`7:15am`	7 hours and 15 minutes (0715)
`5:30pm`	17 hours and 15 minutes (1715)

3.2.3 The DESC String

The DESC string is a multi-word string enclosed in double quotes that denotes either a search string, or the description of an Event. A DESC string can be a single word (`"drycleaning"`) or a sequence of words (`"pick up drycleaning"` or `"meet Joe"`). Note that DESC strings are collectively enclosed in double quotes.

Representative examples of a DESC string:

```
"hello"
"meet Mary"
"find Jeff a coat"
```

3.2.4 The EVENT String

The EVENT string describes an `Event` that the user would like created. It contains the description of the event (as a DESC string), and optionally its date, its time and duration.

An EVENT string has to contain a DESC string. If a DATE string is specified, the event occurs on that date, otherwise the event is assumed to occur today. If a TIME string is specified, the event starts at that time, otherwise the event is an all-day event. These three can occur in any order. Additionally, it can have an optional DURATION string (an integer enclosed in square brackets) **at the end**. If the event is determined to be all-day, the event's duration is still set, but it is never used.

Representative examples of an EVENT string:

`"grocery store" today 9:30am [75]`	Event with description "grocery store" that occurs on the current Python date at 0930h and lasts 75 minutes
`3am tomorrow "party hard"`	Event with description "party hard" that occurs a day after the current date at 0300 and lasts 60 minutes (default)
`"call grandma" May-19-2012`	Event with description "call grandma" that occurs on May 19, 2012 and lasts all day (no time specified)
`"nap" 5:30pm [20]`	Event with description "nap" that occurs on the current Python date at 1730 and lasts 20 minutes

3.3 Messages

In order to standardize everyone's output, we are supplying you with a numbered list of messages that your program should output in certain situations. You will find details about which message to use when in part 4.4 and 4.5.

Messages and Message Numbers:

MESSAGE01	`'EVENTS MATCHING "%s":' % searchstring`
MESSAGE02	`'NO EVENTS MATCH "%s"!' % searchstring`
MESSAGE03	`"DELETED:"`
MESSAGE04	`"NO EVENTS TO DELETE!"`
MESSAGE05	`"ADDED: %s" % event`
MESSAGE06	`"EVENT NOT ADDED! CONFLICTS WITH:"`
MESSAGE07	`"EVENTS FOR %s:" % argument`

3.4 Keywords

Now that we've seen some commonly occurring string types, let's have a look at our main keywords. A command in CAL can start with one of the following:

`exit` This keyword is followed by nothing and indicates the program should terminate. It is already implemented in your main method. CAL will keep asking for commands and doing work until the user types in exit.

`clear` This keyword is followed by a DATE string and indicates that all `Events` on the date described by the DATE string (see ~~4.2.1~~ 3.2.1) should be removed from the database. If there are any `Events` to delete, remove them and print MESSAGE03, followed by the time representations of all deleted `Events` on separate lines. If there are no `Events` to delete, print MESSAGE04.

`see` This keyword indicates the user would like information without changing anything. It is followed by:

MONTH (jan-dec)	Prints a calendar representation of MONTH of this year using the `printmonth()` method
YEAR (1000-9999)	Prints a calendar representation of YEAR using the `printyear()` method
MONTH YEAR	Prints a calendar representation of MONTH of the year specified by the four digits in YEAR using the `printmonth()` method
DATE	Prints MESSAGE07 (with the DATE string supplied by the user as argument), followed by a sorted list of `Events` associated with the day denoted by this DATE string (see **3.2.1**). `Events` are printed using the `timerep()` method (no dates)

`event` This keyword indicates the user will be adding, deleting or querying Events. It is followed by:

`search "searchstring"`	Prints MESSAGE01 followed by a sorted list of all `Events` in the database whose description contains `searchstring`. `Events` are printed with their dates as specified by `__str__`. If no `Events` match `searchstring`, prints MESSAGE02
`all`	Prints a sorted list of all `Events` in the database.
EVENT	Adds a new `Event` to the database with the attributes described by the EVENT string (see **3.2.4**) if no conflicts found (see **3.4**)

3.5 Conflicts

When an `event` command is executed, an `Event` should be added to the database. However, CAL will require that the `Event` not conflict with any other events before being added.

What should happen when an event command is executed is that CAL should check the candidate `Event` against every known `Event` in the database and ensure they do not conflict.

A conflict is defined as an overlap between two `Events` with one `Event`'s start time falling between the start and end times of another. Note that an `Event` that starts at the exact same time that another `Event` ends is not in conflict with that `Event`.

All-day events do not conflict with one another or with any time-scheduled event.

Your program should check for conflicts. If there are none, it should print MESSAGE05 and add the `Event` to the database. If there are conflicts, it should print MESSAGE06, followed by a sorted list of `Events` that conflict with the candidate `Event`. The `Event` should not be added.