Lab 1: Telling time on the planet “Regulus”

Our idea of a current date (such as January 23, 2004) or time (such as 12:30PM and 0 seconds) are defined in terms of how long it has been since a particular event (related to the start date of the Christian calendar, or the beginning of the day at midnight). In some cases, these definitions involve subtleties, such as the adjustment of clocks for daylight savings time or when crossing time zones, or the adjustment of the calendar in the Gregorian Reformation (compare the calendars printed by the unix calendar program for January 2004 (type “cal 1 2004” in a terminal window) and September 1752 (type “cal 9 1752”)).

Some computer systems represent dates and times in the terms that would be shown on the calendar and clock. Others use a different representation; for example, the date and time can be represented in terms of the number of seconds since the start of January 1, 1970 UTC. Each representation makes some operations easy and others hard (e.g. printing as a string, or finding the number of minutes between two times).

In this lab, you will create classes to represent date and time information. To reduce the number of arbitrary rules you’ll need to program (such as leap years), you will write a class representing time on the hypothetical planet “Regulus”: Regulus has days that are exactly as long as Earth days (24 hours, each of 60 minutes, each of which is 60 seconds long), but the year contains exactly 360 days, which the occupants group into 12 months of 30 days each. Years are counted from the original landing of explorers or Regulus.

Regulus has one long, thin continent, and thus no need for time zones. However, this continent is well north of the equator, and Regulus has significant axial tilt, so daylight savings time is used. Everyone sets their clocks forward by 1 hour on April 1, at 2AM, to go from RST to RDT; everyone sets their clocks back by 1 hour on October 1, the first time the clocks read 3AM.

Get the starter files for the Regulus_time project, and create classes that represent date and time information on Regulus in two different ways: one class (Time_represented_by_seconds) should use a count of seconds, as on the original Unix operating system; the other class (Time_represented_by_clock_and_calendar) should use separate fields for days, hours, minutes, etc.. In the former, operations such as adding a certain number of minutes will be easy, but printing will be hard; in the latter, adding minutes is tricky but printing is easy — note that most of the non-trivial steps will appear somewhere in each version, so you may want to build and debug one class first, and then cut-and-paste various bits of your algorithm as you create the other. The test suite expects that Time will be defined as one of your time classes, for example by setting Time = Time_represented_by_clock_and_calendar as in the DocTest section.

The two classes should respond to the following messages in exactly the same way — it should be impossible to determine which class is in use through any means other than looking at the source code or measuring which operations are fast and which are slow. Each class should have:

- A constructor to create a Time from a number of seconds since the start of Regulus time. It is, of course, more than a little inconvenient for users to create times by entering, say, t1 = Time(03 + 60*(25 + 60*(01 + 24*(((01-1) + 30*(((04-1) + 12*((12-1)))))) for 04/01/12 01:25:03 AM RST, but doing anything better makes for a rather tedious lab exercise that loses focus on the creation of data types. If this bothers you, you are welcome to undertake the optional better constructor (see below), but please do that only after you get the rest of the lab working.
• A `minutes_until` accessor method that takes a second `Time` (as a parameter) and returns the number of minutes from the object receiving the message ("self") to the parameter, rounded down to the nearest integer. For example, if we ask “02/01/12 10:59:00 AM RST” for the the `minutes_until “02/01/12 11:05:00 AM RST”, we should get 6, and if we ask “02/01/12 11:03:15 AM RST” for the `minutes_until “02/01/12 11:05:00 AM RST” should be 1. Note that these time values would be constructed in a somewhat cryptic way, and the `minutes_until` would be asked in an object-oriented notation, so the first question could be phrased `Time(344775540).minutes_until(Time(344775900))`. Negative `minutes_until` would result from a parameter that is a time before the receiver.

• An `add_minutes` mutator method that takes an integer number of minutes and adjusts the receiver to be that many minutes later (or earlier, if the number is negative). For any `Time t` and integer `i`, if we define `old_t = deepcopy(t)` and then do `t.add_minutes(i)`, then `old_t.minutes_until(t)` should give us `i`.

• A `__repr__` accessor method to format the time for printing. Remember that, as on Earth, years, months and days are counted starting at 1 rather than 0. You may print afternoon times as either “PM” or hours between 13 and 23, and print numbers below 10 with one ore two digits (i.e., 02/01/12 11:05 and 2/1/12 11:5 are both acceptable). If you print single digits, you may adjust the test suite correspondingly. (Also, if you allow strings in your optional constructor or an input operator, you should be consistent with the way you print them.)

• **OPTIONAL**: Write an alternate (or additional) constructor that takes a string showing how the date would be printed, e.g. "04/01/12 01:25:03 AM RST" for April 1, year 12, at 1:25 AM and 3 seconds, if you adjust the test suite appropriately (while still using the same date/time examples, so that all tests print the same results).

• **OPTIONAL**: Write an input method.

• **OPTIONAL**: Write additional methods to add hours, seconds, minutes, days, or years, and to find differences in these units too.

Your time classes are not required to work for times before landing (i.e. years before 1).

You should also create files

• `README.txt` (to give a concise description of your classes to anyone who might want to use them without reading your class — for example, how would someone writing a program that needed to manipulate date/time information choose which class to use?), and

• `README-DESIGN.txt` (to give a description of your classes to any programmer who might need to update them in the future and thus understand your classes — this can be a very brief overview, if you used clear names and comments, or a more detailed document).

Remember to submit your work when you're done, and to make backup copies (also via Team->Commit) whenever you achieve significant progress.