Lab 10: Virtual Art

Copied from: https://www.cs.hmc.edu/twiki/bin/view/CS5/Lab10 on 3/1/2017

a.k.a. building a `Date` class…

One of Prof. Art Benjamin's abilities is to compute (*in his head*) the day of the week that any past date fell on. For example, if you tell him you were born on June 13, 1997 (6/13/1997), he'll be able to tell you that you were born on a Friday (the 13th, no less!)

This week's lab will guide you through creating a class named `Date` from which you will be able to create objects of type `Date`. Your objects will have the ability to find the day of the week they fell on (or will fall on)…hence, *virtual Art*!

The starting file…

Grab this starting zip file: **hw10pr1.zip** from this link. You'll want to start with the `hw10pr1.py` file inside it. Windows users: be sure to *unzip* the archive—Windows will let you browse the files even without unzipping, but it won't let you open or run them…

The `Date` class

Take a moment to look over the `hw10pr1.py` file as it stands so far…

Notice that in this `Date` class there are three ***data members***:

- A data member holding the month (this is `self.month`)
- A data member holding the day of the month (this is `self.day`)
- A data member holding the year (this is `self.year`)

Note that `self` is used to denote ***any*** object (that is, any variable or value) of class `Date`!

***Methods* are just functions...**

Object-oriented programming tends to have some of its own names for familiar things. For example, *method* is the "OOP" name for *function*. In particular, a *method* is a function whose first argument is `self`!

Note that the `Date` class has an `__init__` method and a `__repr__` method. As we've discussed in class, Python expects to see these special methods in virtually every class. The double underscores before and after these method names indicate that these methods are special ones that Python knows to look for. In the case of `__init__`, this is the method that Python looks for when making a new `Date` object. In the case of `__repr__`, this is the method that Python looks for when it needs to `repr`esent the object as a string.

Notice the line

```
        s =  "%02d/%02d/%04d" % (self.month, self.day,
self.year)
```

in the `repr` method. This constructs a string with the month, day, and the year, formatted very nicely to have exactly two digits places for the month, two digit places for the day, and four for the year.

We've also defined our own `isLeapYear` method. There are no double-underscores here, because Python didn't "expect" this method, but it certainly doesn't "object" to it either. (Clearly our puns have no *class*!)

Note on *"method"*:
Traditionally, functions called by objects are called *methods*. There is no really good reason for this. They are functions - the only thing special about them is that they are defined in a class and they are called after a dot (period) following the name of an object. For example, you might try these:

```
In [1]: d = Date(11, 9, 2016)

In [2]: d.isLeapYear()
Out[2]: True

In [3]: ny = Date(1, 1, 2017)

In [4]: ny.isLeapYear()
Out[4]: False

In [5]: Date(1, 1, 1900).isLeapYear()    # no variable needed!
```

```
Out[5]: False
```

What's up with `self`?

One odd thing about the above example is that **three different objects** of type `Date` are calling the *same* `isLeapYear` code. How does the `isLeapYear` method tell the different objects apart?

The method **does not** know the name of the variable that calls it!

In fact, in the third example, there is *no* variable name! The answer is `self`. The `self` variable holds **the object that calls the method**, including all of its data members.

This is why `self` is always the first argument to all of the methods in the `Date` class (and in any class that you define!): because `self` is how the method can access the individual data members in the object that called it.

Please notice also: this means that a method always has at least one argument, namely `self`. However, this value is passed in *implicitly* when the method is called. For example, `isLeapYear` is invoked in the example above as `Date(1,1,1900).isLeapYear()`, and Python automatically passed `self`, in this case the object `Date(1,1,1900)`, as the first argument to the `isLeapYear` method.

Testing your initial `Date` class:

Just to get a feel for how to test your new datatype, try out the following calls:

```
# create an object named d with the constructor
In [1]: d = Date(11, 9, 2016)  # use another day if you
prefer...

# show d's value
In [2]: d
Out[2]: 11/09/2016

# a printing example
In [3]: print('My favorite day is', d)
My favorite day is 11/09/2016

# create another object named d2
# of *the same date*
```

```
In [4]: d2 = Date(11, 9, 2016)

# show its value
In [5]: d2
Out[5]: 11/09/2016   # _looks_ the same...

# are they the same?
In [6]: d == d2
Out[6]: False

# look at their memory locations
In [7]: id(d)   # return memory address
Out[7]: 413488      # your result will be different...

In [8]: id(d2)  # again...
Out[8]: 430408      # and this should differ from above!

# check if d2 is in a leap year...
In [9]: d2.isLeapYear()
Out[9]: True

# yet another object of type Date
# a distant New Year's Day
# Question: where will you be on this date?
In [10]: d3 = Date(1, 1, 2020)

# check if d3 is in a leap year
In [11]: d3.isLeapYear()
Out[11]: True
```

copy and **equals**

For this part, you should paste the following two methods (**code provided**) into your `Date` class and then test them.

Here are those two methods; we are providing the code so that you have several more examples of defining functions inside a class:

copy(self) Here is the code for the `copy` method:

```
    def copy(self):
        """ Returns a new object with the same month, day, year
```

```
            as the calling object (self).
        """
        dnew = Date(self.month, self.day, self.year)
        return dnew
```

This `copy` method returns a newly constructed object of type `Date` with the same month, day, and year that the calling object has. Remember that the calling object is named `self`, so the calling object's month is `self.month`, the calling object's day is `self.day`, and so on.

Since you want to create a newly constructed object, you *need to call the constructor*! This is what you see happening in the `copy` method.

Try out these examples, which use last year's New Year's Day. First we ***don't*** use copy:

```
In [1]: d = Date(1, 1, 2016)

In [2]: d2 = d

In [3]: id(d)
Out[3]: 430542      # your memory address will differ

In [4]: id(d2)
Out[4]: 430542      # but d2 should have the SAME mem. address
as d!

In [5]: d == d2
Out[5]: True        # so this should be True...
```

Next, you'll show that `copy` does make a deep copy (instead of a copy of only the reference, or "shallow" copy):

```
In [2]: d = Date(1, 1, 2016)    # starting fresh...

In [3]: d2 = d.copy()

In [4]: d
Out[4]: 01/01/2016

In [5]: d2
Out[5]: 01/01/2016

In [6]: id(d)
```

```
Out[6]: 430568      # your memory address will differ

In [7]: id(d2)
Out[7]: 413488      # but d2 should be different from d!

In [8]: d == d2
Out[8]: False       # thus, this should be false...
```

equals(self, d2) Here is the code for this method:

```
    def equals(self, d2):
        """ Decides if self and d2 represent the same calendar
date,
            whether or not they are the in the same place in
memory.
        """
        if self.year == d2.year and self.month == d2.month and
self.day == d2.day:
            return True
        else:
            return False
```

This method should return `True` if the calling object (named `self`) and the argument (named `d2`) represent the same calendar date. If they do not represent the same calendar date, this method should return `False`. The examples above show that the same calendar date may be represented at multiple locations in memory—in that case the `==` operator returns`False`. This method can be used to see if two objects represent the same calendar date, regardless of whether they are at the same location in memory.

Try these examples (after reloading) to get the hang of how this `equals` method works.

```
In [1]: d = Date(1, 1, 2016)

In [2]: d2 = d.copy()

In [3]: d
Out[3]: 01/01/2016

In [4]: d2
Out[4]: 01/01/2016
```

```
In [5]: d == d2
Out[5]: False       # this should be False!

In [6]: d.equals(d2)
Out[6]: True        # but this should be True!

In [7]: d.equals(Date(1, 1, 2016))  # this is OK, too!
Out[7]: True

In [8]: d == Date(1, 1, 2016)       # this tests memory
addresses
Out[8]: False                        # so it should be False
```

Redefining the `==` operator in addition

In Python you can also define operators for your own classes. For example, since the above `equals` method is how we ***want*** to express double-equals, `==`, you can do so by adding a method named `__eq__`. Note that that name has ***two*** underscores on each side of the `eq`

```
    def __eq__(self, d2):
        """ Overrides the == operator so that it declares two of
the same dates in history as ==
            This way , we don't need to use the awkward
d.equals(d2) syntax...
        """
        if self.year == d2.year and self.month == d2.month and
self.day == d2.day:
            return True
        else:
            return False
```

Be sure to include this in your `Date` class as well. Now, `==` should work with objects of class `Date`!

Next, the lab will ask you to implement a few of your own methods for the `Date` class from scratch. We talked about these in class...

Be sure to add a docstring to each of the methods you write! (Recall that the term *method* refers to a function that is a member of a user-defined class.)

isBefore(self, d2) Next, add the following method to your `Date` class:

- `isBefore(self, d2)`:

  This method should return `True` if the calling object is a calendar date ***before*** the argument named `d2` (which will always be an object of type `Date`). If `self` and `d2` represent the same day, this method should return `False`. Similarly, if `self` is *after* `d2`, this should return `False`.

- **Hint**: There are many approaches. A reasonable approach is to first compare the years: `self.year < d2.year`, then the months, then the days. But *be sure to compare months only when the years are equal*. There is a similar constraint on the days (only compare them when *both* the months and years are equal).

Testing `isBefore` To test your `isBefore` method, you should try several test cases of your own design. Here are a few to get you started:

```
In [1]: ny = Date(1,1,2017)     # New Year's

In [2]: d = Date(11,9,2016)

In [3]: ny.isBefore(d)
Out[3]: False

In [4]: d.isBefore(ny)
Out[4]: True

In [5]: d.isBefore(d)         # should be False!
Out[5]: False
```

Similar to what we did with `__eq__`, you can optionally redefine the `<` operator. Its name is `__lt__`.

isAfter(self, d2) Next, add the following method to your `Date` class:

- `isAfter(self, d2)`:

  This method should return `True` if the calling object is a calendar date ***after*** the argument named `d2` (which will always be an object of type `Date`). If `self` and `d2` represent the same day, this method should return `False`. Similarly, if `self` is *before* `d2`, this should return `False`.

You can emulate your `isBefore` code here **OR**

you might consider how to *use* both the `isBefore` and `equals` methods to write `isAfter`.

Testing isAfter To test your `isAfter` method, you should try several test cases of your own design. For example, reverse the examples shown above for `isBefore`.

Also, you can optionally redefine the `&gt` operator. Its name is `__gt__`.

tomorrow(self) Next, add the following method to your `Date` class (you may have class notes that help…)

- `tomorrow(self)`:

  This method should **NOT RETURN ANYTHING**! Rather, it should ***change*** the calling object so that it represents one calendar day *after* the date it originally represented. This means that `self.day` will definitely change. What's more, `self.month` and `self.year` might change.

- You might use the "Luke" trick, which seems Jedi-like to us, and define `fdays = 28 + self.isLeapYear()` ... or you might avoid that trick for fear that Prof. Kuenning will become apocalyptically angry for writing unreadable (too-clever) code [**important disclaimer**: Prof. Kuenning himself acknowledged this last bit], and instead write a proper `if-else` statement to accomplish the same thing.

- Then, list `DIM = [0,31,fdays,31,30,31,30,31,31,30,31,30,31]` of days-in-each-month is useful to have! It makes it easy to determine how many days there are in any particular month (`self.month`).
  - Do you see why the initial 0 is helpful here? It is!

Testing tomorrow To test your `tomorrow` method, you should try several test cases of your own design. Here are a couple of randomly chosen ones to get you started:

```
In [1]: d = Date(12, 31, 2016)

In [2]: d
Out[2]: 12/31/2016

In [3]: d.tomorrow()

In [4]: d
Out[4]: 01/01/2017

In [5]: d = Date(2, 28, 2016)

In [6]: d.tomorrow()

In [7]: d
Out[7]: 02/29/2016

In [8]: d.tomorrow()

In [9]: d
Out[9]: 03/01/2016
```

yesterday(self) Next, add the following this complementary method to your `Date` class:

- `yesterday(self)`:

  Like tomorrow, this method should not return anything. Again, it should change the calling object so that it represents one calendar day *before* the date it originally represented. Again, `self.day` will definitely change, and `self.month` and `self.year` might change.

Testing yesterday To test your `yesterday` method, you should try several test cases of your own design. Here are the reverses of the previous tests:

```
In [2]: d = Date(1, 1, 2016)

In [3]: d
Out[3]: 01/01/2016
```

```
In [4]: d.yesterday()

In [5]: d
Out[5]: 12/31/2015

In [6]: d = Date(3, 1, 2016)

In [7]: d.yesterday()

In [8]: d
Out[8]: 02/29/2016

In [9]: d.yesterday()

In [10]: d
Out[10]: 02/28/2016
```

addNDays(self, N) Next, add the following method to your `Date` class:

- `addNDays(self, N)`:

  This method only needs to handle nonnegative integer arguments `N`. Like the `tomorrow` method, this method should not return anything. Rather, it should ***change*** the calling object so that it represents `N` calendar days *after* the date it originally represented.

Don't copy any code from the tomorrow method!

Instead, consider how you could ***call*** the `tomorrow` method inside a `for` loop in order to implement this!

In addition, this method should *print* all of the dates from the starting date to the finishing date, inclusive of both endpoints. Remember that the line `print(self)` can be used to print an object from within one of that object's methods. See below for examples of output.

Testing To test your `addNDays` method, you should try several test cases of your own design. Here are a couple to start with:

```
In [1]: d = Date(11, 9, 2016)
```

```
In [2]: d.addNDays(3)
11/09/2016  # printing the first one is optional...
11/10/2016
11/11/2016
11/12/2016

In [3]: d
Out[3]: 11/12/2016

In [4]: d = Date(11, 9, 2016)  # re-create the original Date

In [5]: d.addNDays(1278)
11/09/2016   # printing the first one is optional
11/10/2016
... lots of dates skipped ...
05/09/2020
05/10/2020

In [6]: d
Out[6]: 05/10/2020    # graduation! (if you're now a first-
year...)
```

You can check your own date arithmetic with this website: http://www.timeanddate.com/date/dateadd.html. Note that 1752 was a ***weird*** year for the United States/colonies' calendar—especially September! (And 1712 saw the only Feb. 30th - and only in Sweden!) Note that your `Date` class does **not** need to handle these unusual situations— in fact, it *shouldn't* do so, so that we can test things consistently!

subNDays(self, N) Next, include the following method in your `Date` class:

- `subNDays(self, N)`:

  This method only needs to handle nonnegative integer arguments `N`. Like the addNDays method, this method should not return anything. Rather, it should ***change*** the calling object so that it represents `N` calendar days *before* the date it originally represented. Analogous to the previous case, consider using `yesterday` and a `for` loop to implement this!

In addition, this method should **print** all of the dates from the starting date to the finishing date, inclusive of both endpoints. Again, this mirrors the `addNDays` method. See below for examples of the output.

Testing subNDays: try reversing the above test cases!

If you like redefining Python's operators, you could create

- `__iadd__(self,N)` for `addNDays(self,N)` - this creates the ability to use `d += 1` or `d += 1000`
- `__isub__(self,N)` for `subNDays(self,N)` - this creates the ability to use `d -= 1` or `d 1= 1000`

---

diff(self, d2) Next, add the following method to your `Date` class:

- `diff(self, d2)`:

  This method should return an integer representing the *number of days* between `self` and `d2`. You can think of it as returning the integer representing

```
self - d2
```

In fact, `__sub__` is the subtraction operator - feel free to define it, in addition to `diff`.

Dates are more complicated than integers!! So, implementing `diff` will be more involved. See below for some hints….

One crucial point: **this method should NOT change `self` NOR should it change `d2`**!

Rather, you should create and manipulate ***copies*** of `self` and `d2`—this will ensure that the originals remain unchanged.

Thus, make a copy of each of `self` and `d2` and then only use and change those copies! For example,

- `self_copy = self.copy()`
- `d2_copy = d2.copy()`

Also, **The sign of the return value is important!** Consider these three cases:

- If `self` and `d2` represent the same calendar date, this method `diff(self, d2)` should return `0`.
- If `self` is ***before*** `d2`, this method `diff(self, d2)` should return a **negative** integer equal to the number of days between the two dates.
- If `self` is ***after*** `d2`, this method `diff(self, d2)` should return a **positive** integer equal to the number of days between the two dates.

Two approaches ***not*** to use!

- First, don't try to subtract years, months, and days between two dates: this is *way* too error-prone.
- By the same token, however, don't use `addNDays` or `subNDays` to implement your `diff` method. Checking all of the possible difference amounts will be too slow! Rather, implement `diff` in the same *style* as those two methods: namely, using `yesterday` and/or `tomorrow` and loops.

What to do?

- So, you will want to use the `tomorrow` and `yesterday` methods you've already written—now, they will be inside a `while` loop!
- The test for the while loop could be something like `while day1.isBefore(day2):` or it may use `isAfter`…
- Use a counter variable to count the number of times you need to loop before it finishes: that is your answer (perhaps with a negative sign)!

Testing diff To test your `diff` method, you should try several test cases. Here are two relatively nearby pairs of dates:

```
In [1]: d = Date(11,9,2016)      # now...

In [2]: d2 = Date(12,16,2016)     # winter break!

In [3]: d2.diff(d)
Out[3]: 37
```

```
In [4]: d.diff(d2)
Out[4]: -37

In [5]: d          # make sure they did not change!
Out[5]: 11/9/2016

In [6]: d2         # make sure they did not change!
Out[6]: 12/16/2016

# Here are two that pass over a leap year...
In [7]: d3 = Date(12,1,2015)

In [8]: d4 = Date(3,15,2016)

In [9]: d4.diff(d3)
Out[9]: 105

# And here are two relatively distant pairs of dates:
In [10]: d = Date(11, 9, 2016)

In [11]: d.diff(Date(1, 1, 1899))
Out[11]: 43046

In [12]: d.diff(Date(1, 1, 2101))
Out[12]: -30733
```

Use your `diff` method to compute your own age - or someone else's age - in days!
You can check other differences
at www.timeanddate.com/date/duration.html.

dow(self) Next, add the following method to your `Date` class:

- `dow(self)`:

  This method should return a string that indicates the day of the week (`dow`) of the object (of type `Date`) that calls it. That is, this method returns one of the following strings: "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", or "Sunday".

Hint: How might it help to find the `diff` from a *known* date, like Wednesday, November 9, 2016? How might the mod (`%`) operator help?

Testing dow To test your `dow` method, you should try several test cases of your own design. Here are a few to get you started:

```
In [1]: d = Date(12, 7, 1941)

In [2]: d.dow()
Out[2]: 'Sunday'

In [3]: Date(10, 28, 1929).dow()     # dow is appropriate for
the Dow Jones's crash!
Out[3]: 'Monday'

In [4]: Date(10, 19, 1987).dow()     # ditto: another crash!
Beware October Mondays!
Out[4]: 'Monday'

In [5]: d = Date(1, 1, 2100)

In [6]: d.dow()
Out[6]: 'Friday'
```

Submission

Congratulations—you have created a `Date` class whose objects can compute the differences and the days of the week for any calendar dates at all!

Ans, you've completed Lab 10. Be sure to submit in the usual place!

If you'd like, you can now compute what day of the week your birthday is most likely to be for the next century... or NY's day...

Optional Using your Date class...

To take advantage of your `Date` class, we will put it to use! This will include an investigation of your birthday's most likely day-of-the-week, as well as our calendar's statistics for the 13th of each month.

Try the following code and then answer the three questions below.

Consider the following function—you will want to paste it at the bottom of your `hw10pr1.py` file. Be sure to paste it ***OUTSIDE*** the `Date` class. That is, make sure this function is indented all the way to the left (not one indentation rightward, because this is not a method of the `Date` class)!

This `nycounter` function uses a *dictionary* data structure. You may or may not have seen these (we will use them this Thursday and on this week's homework).

Briefly, a dictionary `d` is initialized with the code `d = {}` (note the curly braces), but it is used very much like a normal list. Its key difference is that *strings* can act as the indices of the list! As a result, dictionaries are great for counting the number of times a string appears. Here is an example of just that:

```
def nycounter():
    """Looking ahead to 100 years of NY celebrations..."""

    dowd = {}               # dowd == 'day of week dictionary'
    dowd["Sunday"] = 0      # a 0 entry for Sunday
    dowd["Monday"] = 0      # and so on...
    dowd["Tuesday"] = 0
    dowd["Wednesday"] = 0
    dowd["Thursday"] = 0
    dowd["Friday"] = 0
    dowd["Saturday"] = 0

    # live for another 100 years...
    for year in range(2016, 2116):
        d = Date(1, 1, year)   # get ny
        print('Current date is', d)
        s = d.dow()         # get day of week
        dowd[s] += 1        # count it

    print('totals are', dowd)

    # we could return dowd here
    # but we don't need to right now
    # return dowd
```

First, try this `nycounter` function out:

```
In [1]: nycounter()
```

Question 1

In a comment above this `nycounter` function in your `hw10pr1.py` file, write one or two sentences describing what this `nycounter` computes.

Question 2

Based on the `nycounter` example, write a function that will compute the same information for ***your next 100 birthdays***. Include the results in a comment below your function.

Question 3 [What are the most common days of the week?]

Based on these two examples, write a function that will compute the same information for the ***13th of every month*** for the next 400 years. Since our current calendar cycles every 400 years, your results will be the overall frequency distributions for the 13th of the month for as long as people retain our current calendar system. What day of the week is the 13th most and least likely to fall on? Is it a tie?

Note that you will be taking *full* advantage of your computer's processor for this problem. You might want to print out a status line, e.g., each year, so that you know that it's making progress. **Even so, this will be too slow!** So after you've watched the dates slow down and get tired of waiting, consider the suggestions below and make the indicated changes to speed things up (Control-C will kill your program so you don't have to wait for it to finish).

The problem is with `dow`—it's checking longer and longer intervals. How might you change how `dow` works to speed up the processing in this case?

One way to do it would be to write an alternative method in your `Date` class, named `dow2(self, refDate)`. The second argument to `dow2` would be ***any reference date of the appropriate day of the week***. You would call

```
d.dow2(refDate)
```

instead of `d.dow()`.

The idea is this: create a variable called `refDate` at the top of your "thirteenthcounter" function. Initially give that variable the value of the reference date used in your original `dow`function. However, as you seek into the future, check each thirteenth to see if it's the *same day of the week as your original* `refDate`. If it is the same day of the week as the original `refDate`, then update the value of the `refDate` variable to be that new date—namely, the thirteenth of the month you just found.

That way, you will never be calling `dow2` on spans of dates of more than a year—if you do this, it should finish in less than a minute. You'll notice that the thirteenth is a Friday more often than several other days of the week...!

Starting code for the Date class

In case it's useful to have available from this webpage, here is the start of the `Date` class that can be cut-and-pasted into a `hw10pr1.py` file.

```
class Date:
    """ a user-defined data structure that
        stores and manipulates dates
    """

    def __init__(self, month, day, year):
        """ the constructor for objects of type Date """
        self.month = month
        self.day = day
        self.year = year


    def __repr__(self):
        """ This method returns a string representation for the
            object of type Date that calls it (named self).

             ** Note that this _can_ be called explicitly, but
                it more often is used implicitly via the print
                statement or simply by expressing self's value.
        """
        s =  "%02d/%02d/%04d" % (self.month, self.day,
self.year)
        return s


    def isLeapYear(self):
        """ Returns True if the calling object is
            in a leap year; False otherwise. """
        if self.year % 400 == 0: return True
        elif self.year % 100 == 0: return False
        elif self.year % 4 == 0: return True
        return False
```