Problem 1: Virtual Art

One of Harvey Mudd Professor 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 October 13, 1995, 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*!

To get started, make a copy of this trinket:

The `Date` class

Take a moment to look over the Homework 10, Problem 1 code 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 (line 29). 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 or period following the name of an object. For example, you might try these:

```
>>> d = Date(11, 12, 2014)
>>> d.isLeapYear()
False

>>> d2 = Date(3, 15, 2016)
>>> d2.isLeapYear()
True

>>> Date(1, 1, 1900).isLeapYear()    # no variable
needed!
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
>>> d = Date(11, 12, 2014)  # use day 11 if you
prefer

# show d's value
>>> d
11/12/2014

# a printing example
>>> print 'Wednesday is', d
Wednesday is 11/12/2014

# create another object named d2
# of *the same date*
>>> d2 = Date(11, 12, 2014)

# show its value
>>> d2
11/12/2014

# are they the same?
>>> d == d2
False

# look at their memory locations
>>> id(d)   # return memory address
413488      # your result will be different

>>> id(d2)  # again
430408      # this should differ from above!

# check if d2 is in a leap year—it is!
>>> d2.isLeapYear()
False

# yet another object of type Date
# a distant New Year's Day
>>> d3 = Date(1, 1, 2020)

# check if d3 is in a leap year
>>> d3.isLeapYear()
True
```

`copy` and `equals`

For this part, you should paste the following two methods (**code provided**) into your `Date` class and then test them. We are providing the code so that you have several more examples of what it is like to define functions inside a class:

`copy(self)`

Here is the code for this 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 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 this year's New Year's Day. First we ***don't*** use copy:

```
>>> d = Date(1, 1, 2015)
>>> d2 = d
>>> id(d)
430542      # your memory address may
differ
>>> id(d2)
430542      # but d2 should be the SAME as
d!
>>> d == d2
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):

```
>>> d = Date(1, 1, 2015)
>>> d2 = d.copy()
>>> d
01/01/2015
>>> d2
01/01/2015

>>> id(d)
430568      # your memory address may differ
>>> id(d2)
413488      # but d2 should be different from
d!
>>> d == d2
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 to get the hang of how this `equals` method works:

```
>>> d = Date(1, 1, 2015)
>>> d2 = d.copy()
>>> d
01/01/2015
>>> d2
01/01/2015
>>> d == d2
False       # this should be False!

>>> d.equals(d2)
True        # but this should be True!

>>> d.equals(Date(1, 1, 2015))   # this is OK, too!
True

>>> d == Date(1, 1, 2015)        # tests memory
addresses
False                            # so it should be
False
```

Methods to Write

Now, the next part of this problem asks you to implement a few methods for the `Date` class from scratch.

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.)

`tomorrow(self)`

Add the `tomorrow(self)` method to your `Date` class (you may have class notes that help):

- 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 define `fdays = 28 + self.isLeapYear()`, or you might avoid that trick for fear of writing unreadable code, and instead write a proper `if`-`else` statement to accomplish the same thing.
- Then, the 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?

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:

```
>>> d = Date(12, 31,
2014)
>>> d
12/31/2014
>>> d.tomorrow()
>>> d
01/01/2015

>>> d = Date(2, 28, 2016)
>>> d.tomorrow()
>>> d
02/29/2016
>>> d.tomorrow()
>>> d
03/01/2016
```

`yesterday(self)`

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

- 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:

```
>>> d = Date(1, 1,
2015)
>>> d
01/01/2015
>>> d.yesterday()
>>> d
12/31/2014

>>> d = Date(3, 1,
2016)
>>> d.yesterday()
>>> d
02/29/2016
>>> d.yesterday()
>>> d
02/28/2016
```

`addNDays(self, N)`

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

- 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 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 `addNDays`

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

```
>>> d = Date(11, 12, 2014)
>>> d.addNDays(3)
11/12/2014
11/13/2014
11/14/2014
11/15/2014
>>> d
11/15/2014

>>> d = Date(11, 12, 2014)
>>> d.addNDays(1278)
11/12/2014
11/13/2014
 lots of dates skipped
5/12/2018
5/13/2018
>>> d
5/13/2018    # graduation! (if you're now in your first
year)
```

You can check your own date arithmetic with this website.

Note that 1752 was a **weird** year for the United States/colonies' calendar—especially September! However, your `Date` class does **not** need to handle these unusual situations—and it **shouldn't** do so, so that we can test things consistently!

`subNDays(self, N)`

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

- 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.

Testing `subNDays`

Try reversing the above test cases!

`isBefore(self, d2)`

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

- 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`.

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:

```
>>> ny = Date(1,1,2015)     # New Year's
>>> d2 = Date(11,12,2014)
>>> ny.isBefore(d2)
False
>>> d2.isBefore(ny)
True
>>> d2.isBefore(d2)         # should be
False!
False
```

`isAfter(self, d2)`

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

- 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 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, you might reverse the examples shown above for `isBefore`.

`diff(self, d2)`

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

- 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`. But dates are more complicated than integers!! So, implementing `diff` will be more involved. See below for some hints.
- **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.
- 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:

```
>>> d = Date(11,12,2014)     # now
>>> d2 = Date(12,19,2014)   # break!
>>> d2.diff(d)
37
>>> d.diff(d2)
-37
>>> d
11/12/2014
>>> d2       # make sure they did not
change
12/19/2014


# Here are two that pass over a leap year
>>> d = Date(12,1,2015)
>>> d3 = Date(3,15,2016)
>>> d3.diff(d)
105
```

And here are two relatively distant pairs of dates:

```
>>> d = Date(11, 12, 2014)
>>> d.diff(Date(1, 1,
1899))
42318
>>> d.diff(Date(1, 1,
2101))
-31461
```

Use your `diff` method to compute your own age (or someone else's age) in days!

You can check other differences at this website.

`dow(self)`

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

- 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 12, 2014? 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:

```
>>> d = Date(12, 7, 1941)
>>> d.dow()
'Sunday'

>>> Date(10, 28, 1929).dow()     # dow is appropriate: crash
day!
'Monday'

>>> Date(10, 19, 1987).dow()     # ditto!
'Monday'

>>> d = Date(1, 1, 2100)
>>> d.dow()
'Friday'
```

You can check your days of the week at this website by entering the year you're interested in.

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!

You might see what day of the week your birthday (or New Year's Day) is most likely to be for the next century (see the Extra Challenges for this week) or, feel free to submit your Homework 10, Problem 1 code as-is.

Submit Homework 10, Problem 1

30.0/30.0 points (graded)

To submit your Homework 10, Problem 1 code, you'll need to copy it from the trinket above and paste it into the box below. After you've pasted your code below, click the "Check" button.

IMPORTANT: Make sure that there aren't spaces at the beginning of your code, and that you copied all of the characters. If there are extra spaces or you are missing spaces, our server won't be able to run your code and we won't be able to give you any of the points you deserve for your hard work.

1

2

```
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
```

Press ESC then TAB or click outside of the code editor to exit
correct

correct

Test results
CORRECT See full outputSee full output

You have used 1 of 3 attempts Some problems have options such as save, reset, hints, or show answer. These options follow the Submit button.