This week's homework will guide you through creating a class named `Date` which will allow you to do computations on dates.

The `Date` class

Grab this starting file date.py.

Take a moment to look over this `date.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 input is `self`!

Note that the `Date` class has an `__init__` method and a `__str__` 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 methods 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 `__str__`, this is the method that Python looks for when it needs to `represent` the object as a string.

Notice the line

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

in the `str` 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. If you would like to learn more about fancy string formatting, you can look at the string documentation page.

We've also defined our own `isLeapYear` method. There are no double-underscores here, so Python didn't "expect" this method, but it certainly doesn't "object" to it either (sorry for the bad pun - we clearly 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, 8, 2011)
>>> d.isLeapYear()
False


>>> d2 = Date(3, 15, 2012)
>>> 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 input 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 input 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 input 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, 9, 2011)

# show d's value
>>> d
11/09/2011

# a printing example
>>> print 'Wednesday is', d
Wednesday is 11/09/2011
```

```
# create another object named d2
# of *the same date*
>>> d2 = Date(11, 9, 2011)

# show its value
>>> d2
11/09/2011

# 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's not!
>>> d2.isLeapYear()
False

# yet another object of type Date
# next year's NYDay
>>> d3 = Date(1, 1, 2012)

# 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...

```
>>> d = Date(1, 1, 2011)
>>> d2 = d.copy()
>>> d
01/01/2011
>>> d2
01/01/2011
```

```
>>> 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.'''
    return self.year == d2.year and self.month == d2.month and \
           self.day == d2.day
```

This method should return `True` if the calling object (named `self)` and the input (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 your file) to get the hang of how this `equals` method works:

```
>>> d = Date(1, 1, 2011)
>>> d2 = d.copy()
>>> d
01/01/2011
```

```
>>> d2
01/01/2011
```

```
>>> d == d2
```

```
False         # this should be False!


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


>>> d.equals(Date(1, 1, 2011))       # no name needed!
True


>>> d == Date(1, 1, 2011)         # tests memory addresses
False                             # so it should be False
```

Now, the next part of the homework 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`

First, add the following method to your `Date` class (you may have 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 may find the list `DIM = (0,31,28,31,30,31,30,31,31,30,31,30,31)` a useful line to have at the very top of your function... !

That list makes it easy to determine how many days there are in any particular month (self.month).

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, 2010)
>>> d
12/31/2010


>>> d.tomorrow()
>>> d
```

```
01/01/2011


>>> d = Date(2, 28, 2012)
>>> d.tomorrow()
>>> d
02/29/2012
>>> d.tomorrow()
>>> d
03/01/2012
```

`yesterday`

Next, add the following 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:

```
>>> d = Date(1, 1, 2011)
>>> d
1/1/2011
>>> d.yesterday()
>>> d
12/31/2010



>>> d = Date(3, 1, 2012)
>>> d.yesterday()
>>> d
02/29/2012
>>> d.yesterday()
>>> d
02/28/2012
```

`addNDays`

Next, add the following method to your `Date` class:

- `addNDays(self, N):`

  This method only needs to handle nonnegative integer inputs `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, 9, 2011)
>>> d.addNDays(3)
11/09/2011
11/10/2011
11/11/2011
11/12/2011
>>> d
11/12/2011
```

```
>>> d = Date(11, 11, 2011) # the near future
>>> d.addNDays(1283)
11/11/2011
11/12/2011
... lots of dates skipped ...
5/16/2015
5/17/2015
>>> d
5/17/2015# graduation! (if not before...)
```

You can check your own additions 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!
However, your `Date` class does not need to handle these unusual situations (and shouldn't do so, so that we can test things consistently!).

`subNDays`

Next, include the following method in your `Date` class:

- `subNDays(self, N):`

  This method only needs to handle nonnegative integer inputs `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. You might 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!

`isBefore`

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 input 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 returnFalse.

It might be worth mentioning that you could pass a string or a float or an integer as the input to this `isBefore` method. In this case, your code will likely raise a `TypeError`. We won't do this to your code! Python relies on the user to keep track of the types being used.

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:

```
>>> d = Date(11, 11, 2011)
>>> d2 = Date(1, 1, 2012)
>>> d.isBefore(d2)
True
>>> d2.isBefore(d)
False
>>> d.isBefore(d)
False
```

`isAfter`

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 input 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 returnFalse.

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`

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
```

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

One crucial point: this method should NOT change `self` NOR should it change `d2!` You should create and manipulate copies of `self` and `d2,` so 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 should return `0.`
- If `self` is before `d2,` this method should return a negative integer equal to the number of days between the two dates.
- If `self` is after `d2,` this method 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 `tomorrowand` loops.

How do we do this?

- You will want to use the `tomorrow` and `yesterday` methods you've already written -- inside a `while` loop!
- The test for the while loop may be something like `while day1.isBefore( day2 ):` or it may use `isAfter...`
- Use `yesterday` or `tomorrow` within the loop and count the number of times you need to loop before it finishes...

Testing `diff`

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

```
>>> d = Date(11,9,2011)
>>> d2 = Date(12,16,2011)
>>> d2.diff(d)
37
>>> d.diff(d2)
-37
>>> d
11/09/2011
>>> d2# make sure they did not change...
12/16/2011
```

```
# Here are two that pass over a leap year...
>>> d = Date(11,9,2011)
>>> d3 = Date(5, 18, 2012)
>>> d3.diff(d)
191
```

And here are two relatively distant pairs of dates:

```
>>> d = Date(11, 9, 2011)
>>> d.diff(Date(1, 1, 1899))
41219
```

```
>>> d.diff(Date(1, 1, 2101))
-32560
```

You can check other differences at www.timeanddate.com/date/duration.html .

dow

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, 2011? 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 www.timeanddate.com/calendar/ by entering the year you're interested in.

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