20. More Complicated Classes

Topics:
- Example: The class `Fraction`
- Operator Overloading
- Class Invariants
- Example: The class `SimpleDate`
- Class Variables
- `deepcopy`
A Class For Manipulating Fractions

2/3 + 13/6 = (2*6+13*3)/(3*6)
= 51/18
= 17/6

You in Grade School:

Python in College:

```python
>>> x = Fraction(2,3)
>>> y = Fraction(13,6)
>>> z = x+y
>>> print z
17/6
```
A Class For Manipulating Fractions

You in Grade School:

\[
\frac{2}{3} \times \frac{3}{4} = \frac{(2 \times 3)}{(3 \times 4)} = \frac{6}{12} = \frac{1}{2}
\]

Python in College:

```python
>>> x = Fraction(2,3)
>>> y = Fraction(3,4)
>>> z = x+y
>>> print z
1/2
```
Let's Define a Class to Do This Stuff

class Fraction(object):
    """
    Attributes:
        num: the numerator [int]
        den: the denominator [int]
    """

Not good enough. Do not want zero denominators!
Let's Define a Class to Do This Stuff

```python
class Fraction(object):
    """
    Attributes:
        num: the numerator [int]
        den: the denominator [nonzero int]
    """
```

Still not good enough. Fractions should be reduced to lowest terms, e.g., \(-3/2\) not \(-24/16\)
A Note About Greatest Common Divisors

Reducing a fraction to lowest terms involves finding the \textit{gcd} of the numerator and denominator and dividing.

<table>
<thead>
<tr>
<th>$p$</th>
<th>$q$</th>
<th>$\text{gcd}(p,q)$</th>
<th>$p/q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>24</td>
<td>8</td>
<td>2/3</td>
</tr>
<tr>
<td>19</td>
<td>47</td>
<td>1</td>
<td>19/47</td>
</tr>
<tr>
<td>15</td>
<td>25</td>
<td>5</td>
<td>3/5</td>
</tr>
</tbody>
</table>
Computing the Greatest Common Divisor

```python
def gcd(a, b):
    a = abs(a)
    b = abs(b)
    r = a % b
    while r > 0:
        a = b
        b = r
        r = a % b
    return b
```

Euclid’s Algorithm

300BC

We will assume this is given and won’t worry why it works
class Fraction(object):
    ""
    Attributes:
        num: the numerator [int]
        den: the denominator [nonzero int]
        num/den is reduced to lowest terms
    """
The Constructor

def __init__(self,p,q=1):
    d = gcd(p,q)
    self.num = p/d
    self.den = q/d

>>> x = Fraction(10,4)
>>> print x
5/2

>>> x = Fraction(10)
>>> print x
10/1

Whole numbers are fractions too. Handy to use the optional argument feature.
Let’s Look at the Methods Defined in the Class Fraction

Informal synopsis:

<table>
<thead>
<tr>
<th></th>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>negate</td>
<td>2/3</td>
<td>-2/3</td>
</tr>
<tr>
<td>Invert</td>
<td>2/3</td>
<td>3/2</td>
</tr>
<tr>
<td><strong>add</strong></td>
<td>2/3 + 1/6</td>
<td>5/6</td>
</tr>
<tr>
<td><strong>mul</strong></td>
<td>2/3 * 1/6</td>
<td>1/9</td>
</tr>
</tbody>
</table>

The double underscore methods make a nice notation possible. Instead of $f1.add(f2)$ we can just write $f1+f2$. 
def negate(self):
    """ Returns the negative of self """
    F = Fraction(-self.num, self.den)
    return P

>>> x = Fraction(6,-5)
>>> print x
-6/5
>>> y = x.negate()
>>> print y
6/5
The invert Method

def invert(self):
    """ Returns the reciprocal of self
    PreC: self is not zero
    """
    F = Fraction(self.den,self.num)
    return F

>>> x = Fraction(100,95)
>>> print x
20/19
>>> y = x.invert()
>>> print y
19/20
Consider Addition

\[ s = 'dogs' + 'and' + 'cats' \]

\[ x = 100 + 200 + 300 \]

\[ y = 1.2 + 3.4 + 5.6 \]

What “+” signals depends on the operands. Python figures it out. We say that the “+” operation is overloaded.
Let’s Define “+” For Fractions

```python
def __add__(self,f):
    N = self.num*f.den + self.den*f.num
    D = self.den*f.den
    return Fraction(N,D)
```

```python
>>> A = Fraction(2,3)
>>> B = Fraction(1,4)
>>> C = A + B
>>> print C
11/12
```

By defining `__add__` this way we can say `A+B` instead of `A.__add__(B)`

Underlying math:

$$a/b + c/d = (ad+bc)/bd$$
Likewise for Multiplication

```python
def __mul__(self,f):
    N = self.num*f.num
    D = self.den*f.den
    return Fraction(N,D)
```

```python
>>> A = Fraction(2,3)
>>> B = Fraction(1,4)
>>> C = A*B
>>> print C
1/6
```

By defining `__mul__` this way we can say

```
A*B
```

instead of

```
A.__mul__(B)
```
Sometimes we would like to add an integer to a fraction:

\[ \frac{2}{3} + 5 = \frac{17}{3} \]

To make this happen Python needs to know the type of the operands, i.e., “who is to the right of the “+” and who is to the left of the “+”? 
Using the Built-In Boolean-Valued Function `isinstance`

```
>>> x = 3/2
>>> isinstance(x,Fraction)
False
>>> y = Fraction(3,2)
>>> isinstance(y,Fraction)
True
```

Feed `isinstance` it the “mystery” object and a class and it will tell you if the object is an instance of the class.
def __add__(self,f):
    if isinstance(f,Fraction):
        N = self.num*f.den + self.den*f.num
        D = self.den*f.den
    else:
        N = self.num + self.den*f
        D = self.den
    return Fraction(N,D)

If f is a Fraction, use \((a/b + c/d) = (ad+bc)/(bd)\)
A More Flexible __add__

def __add__(self,f):
    if isinstance(f,Fraction):
        N = self.num*f.den + self.den*f.num
        D = self.den*f.den
    else:
        N = self.num + self.den*f
        D = self.den
    return Fraction(N,D)

If f is an integer, use \((a/b + f) = (a+bf)/b\)
A More Flexible `__mul__`

def __mul__(self, f):
    if isinstance(f, Fraction):
        N = self.num * f.num
        D = self.den * f.den
    else:
        N = self.num * f
        D = self.den
    return Fraction(N, D)

If f is a Fraction, use \((a/b)(c/d) = (ac)/(bd)\)
def __mul__(self, f):
    if isinstance(f, Fraction):
        N = self.num * f.num
        D = self.den * f.den
    else:
        N = self.num * f
        D = self.den
    return Fraction(N, D)

If f is an int, use \((a/b)(f) = (af)/b\)
Be Careful!

>>> F = Fraction(2,3)
>>> G = F + 1
>>> print G
5/3
>>> H = 1 + F
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'instance'

When you add an int to a Fraction, the int must be on the right side of the +.
An Example

Let's compute \(1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{15}\)

\[
\begin{align*}
n &= 15 \\
s &= \text{Fraction}(0) \\
\text{for } k \text{ in range}(1,n+1): \\
    &\quad s = s + \text{Fraction}(1,k) \\
\text{print } s
\end{align*}
\]

\(\frac{1195757}{360360}\)

This “+” invokes \texttt{__add__}.
Next, a Class that Supports Computations with Dates
If Today is July 4, 1776, then What is Tomorrow’s Date?

>>> D = SimpleDate('7/4/1776')
>>> print D
July 4, 1776
>>> E = D.Tomorrow()
>>> print E
July 5, 1776
The Check is in the Mail and will Arrive in 1000 Days

```python
>>> D = SimpleDate('1/1/2016')
>>> A = D + 1000
>>> print A
September 27, 2018
```
How Many Days from Pearl Harbor to 9/11?

```python
>>> D1 = SimpleDate('9/11/2001')
>>> D2 = SimpleDate('12/7/1941')
>>> NumDays = D1-D2
>>> print NumDays
21828
```
Class Variables

To pull this off, it will be handy to have a “class variable” that houses information that figures in date-related computations...

nDays = [0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
class SimpleDate(object):
    """
    Attributes:
    m: index of month [int]
    d: the day [int]
    y: the year [int]
    m, d, and y identify a valid date.
    """
The Leap Year Problem

An integer \( y \) is a leap year if it is not a century year and is divisible by 4 or if is a century year and is divisible by 400.

def isLeapYear(self):
    """ Returns True if and only if self encodes a date that part of a leap year. """
    thisWay = ((y%100>0) and y%4==0)
    thatWay = ((y%100==0) and (y%400==0))
    return thisWay or thatWay
Visualizing a SimpleDate Object

```python
>>> D = SimpleDate('7/4/1776')
```

![Diagram of SimpleDate object with breakdown of components: m=7, d=4, y=1776]
The SimpleDate Constructor

def __init__(self, s):
    """ Returns a reference to a SimpleDate representation of the date encoded in s.
    """
    PreC: s is a date string of the form 'M/D/Y' where M, D and Y encode the month index, the day, and the year.
    """
    v = s.split('/')
    m = int(v[0]), d = int(v[1]), y = int(v[2])
    self.m = m, self.d = d, self.y = y

If s = '7/4/1776' then v = ['7', '4', '1776']
The SimpleDate Constructor

Note that

\[
D = \text{SimpleDate}('7/32/1776')
\]

and

\[
D = \text{SimpleDate}('2/29/2015')
\]

produce SimpleDate objects that encode invalid dates.
The SimpleDate Constructor

```python
def __init__(self, s):
    """ Returns a reference to a SimpleDate
    representation of the date encoded in s.
    """

    v = s.split('/')
    m = int(v[0]); d = int(v[1]); y = int(v[2])
    self.m = m; self.d = d; self.y = y
```

A good place to guard against "bad" input using `assert`. 
Use Class Variable nDays

nDays = [0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]

v = s.split('/')
m = int(v[0]); d = int(v[1]); y = int(v[2])
assert 1 <= m <= 12, 'Invalid Month'
assert 1 <= d <= self.nDays[m], 'Invalid Day'

Needs more work. Does not handle leap year situations.
Nothing wrong with SimpleDate('2/29/2016')
Some Simple `DateTime` Methods

Informally...

`Tomorrow` the next day’s date

`__eq__` when are two dates the same?

`__add__` `'7/4/1776' + 364 is '7/3/1777'

`__sub__` `'3/2/2016' - '2/28/2016' is 3
class SimpleDate(object):

    nDays = [ blah ]

    def __init__(self, s):
    
    def __str__(self):
    
    def __eq__(self, other):
    
    def __add__(self, other):
    
    def __sub__(self, other):
    
    def Tomorrow(self):
    
    def isLeapYear(self):
The Method Tomorrow

```python
>>> D = SimpleDate('7/4/1776')
>>> T = D.Tomorrow()
>>> print T
July 5, 1776
```

Pretty printing via `__str__`
The Method Tomorrow

Need a bunch of if constructions to handle end-of-month and end-of-year situations with possible leap year issues:

`'7/4/1776' ----> '7/5/1776'`
`'2/28/1776' ----> '2/29/1776'`
`'2/28/1777' ----> '3/1/1777'`
`'7/31/1776' ----> '8/1/1776'`
`'12/31/1776' ----> '1/1/1777'`
The **__eq__** Method

def __eq__(self, other):
    """ Returns True if and only if other encodes the same date as self """
    B1 = self.m == other.m
    B2 = self.d == other.d
    B3 = self.y == other.y
    return B1 and B2 and B3

>>> D1 = SimpleDate('7/4/1776')
>>> D2 = SimpleDate('4/1/1066')
>>> D1==D2
False
The __add__ Method

def __add__(self,n):
    """ Returns a date that is n days later than self.
    PreC: n is a nonegative integer.
    """
    Day = self
    for k in range(n):
        Day = Day.Tomorrow()
    return Day

>>> D = SimpleDate('1/1/2016')
>>> E = D + 365
>>> print E
December 31, 2016
```python
def __sub__(self, other):
    """D2-D1 returns the number of days from D1 to D2. D2 must be the later date."""

    k = 0
    Day = other
    while not (Day == self):
        k += 1
        Day = Day.Tomorrow()
    return k

>>> D1 = SimpleDate('9/11/2001')
>>> D2 = SimpleDate('12/7/1941')
>>> D1-D2
21828
```
def Tomorrow(self):
    m = self.m
    d = self.d
    y = self.y
    Last = self.nDays[m]
    if isLeapYear(y) and m==2:
        Last+=1
    nDays = [0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
More on Copying Objects

A subtle issue is involved if you try to copy objects that have attributes that are objects themselves.
More on Copying Objects

To illustrate consider this class

class MyColor:
    """
    Attributes:
        rgb: length-3 float list
        name: str
    """
    def __init__(self,rgb,name):
        self.rgb = rgb
        self.name = name
More on Copying Objects

```python
>>> A = MyColor([1,0,0], 'red')
```
More on Copying Objects

>>> B = copy(A)
More on Copying Objects

>>> B = copy(A)

Now let’s make a yellow
More on Copying Objects

```python
>>> A.rgb[1] = 1
>>> A.name = 'yellow'
```

Unintended Effect

B.Rgb refers to a yellow triple
More on Copying Objects

```python
>>> B = deepcopy(A)
```

decopy copies everything