Chapter 5, Python Classes and Objects

John M. Morrison

October 30, 2019

Contents

1 Introduction **2**

2 Creating A Class **2**

3 A Case Study: Playing Cards **6**

3.1 A Design Decision 6

3.2 Getting to Know our Number 7

3.3 Adding Regular Methods to the Class 10

4 Resolving our Identity Crisis **11**

5 Using our `Card` Class **12**

6 Shouldn't a Deck of Cards be an Object? **12**

7 Case Study: Fractions **14**

7.1 An Algorithmic Interlude 17

7.2 Creating a gcd function 18

7.3 Now Back to Fractions 18

7.4 Static Methods 19

7.5 Giving Fractions Relational Operators 21

7.6 Adding Arithmetic 23

8 Case Study: A Calendar Date Class **25**

1 Introduction

Python has yet another level of organization we have not yet explored: the *class*. A class is a blueprint for creating objects; these are computational units which know things about themselves and which exhibit various behaviors when sent messages. The objects we create from the class are called *instances* of the class. This apparatus is baked right into the Python language.

Classes allow us to use a wealth of code created by others. Many authors of code enclose their work in classes; this makes it easier and less confusing to use. We will later learn how to program with a graphical user interface using the QT4 framework of classes.

You have already made liberal use of Python objects. For example, the built-in `str` type features methods such as `upper()`, `lower()`, and `contains()`. The class mechanism allows you to create new types and to make the "smart." You can write code to handle messages sent to objects created with your classes.

2 Creating A Class

You can create a Python class at the interactive prompt. Here is how

```
>>> class First(object):
...     pass
...
>>> f = First()
>>> f
<__main__.First object at 0x20c3250>
```

At this stage, this class is rather useless. It knows nothing and does nothing. The line

```
>>> f = First()
```

creates an *instance* of this class. An object is made according to the specifications we placed in our class and now `f` is pointing at that object.

Now let us make another instance of our class

```
>>> g = First()
```

We now have objects `f` and `g` floating around. We can attach a variable to an object as follows.

```
>>> f.x = "This is x"
```

This attaches `x` to tt f but not to `g`, as we see here.

```
>>> g.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'First' object has no attribute 'x'
```

Now let's create a class with some degree of usefulness. We will represent vectors in the plane with integer coördinates. Here we begin by creating an empty class.

```
class intVector(object):
    pass
```

Our question is *How do we endow the vector with two components?* Python has a special method called a *hook* for this. All hooks have this appearance `__hookName__`. The hook we shall use is the `init` hook. This hook runs right after an object is first created. We can use it to attach coördiantes to our vector.

```
class intVector(object):
    def __init__(self):
        self.x = 0
        self.y = 0
```

All of our vectors are now born with an `x` and a `y` that are both 0. You notice the use of the argument `self`. This symbol refers to the object itself (hence the name). All functions (methods) created inside of a class must have `self` as their first argument. We can improve our class further by doing this.

```
class intVector(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

We now drive it as follows.

```
class intVector(object):
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y

p = intVector()
print("p = {0}i + {1}j".format(p.x, p.y))
q = intVector(3, 4)
print("q = {0}i + {1}j".format(q.x, q.y))
```

Now run it.

```
$ python IntVector.py
p = 0i + 0j
q = 3i + 4j
```

The next question is: Can we get it to print nicely? The `__str__` hook comes to the rescue. While you are a it define the `__repr__` hook so it looks nice in an interactive session.

```
class intVector(object):
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y
    def __str__(self):
        out = "" + str(self.x) + "i"
        if self.y < 0:
            out += " - " + str(-self.y) + "j"
        else:
            out += " + " + str(self.y) + "j"
        return out
    def __repr__(self):
        out = "" + str(self.x) + "i"
        if self.y < 0:
            out += " - " + str(-self.y) + "j"
        else:
            out += " + " + str(self.y) + "j"
        return out
print(p)
q = intVector(3, 4)
print(q)
r = intVector(3, -4)
```

Now run this and see the pretty result.

```
$ python IntVector.py
0i + 0j
3i + 4j
3i - 4j
$
```

We will now make a regular method called `magnitude` that computes the vector's magnitude.

```
import math
class intVector(object):
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y
    def __str__(self):
        out = "" + str(self.x) + "i"
        if self.y < 0:
            out += " - " + str(-self.y) + "j"
        else:
            out += " + " + str(self.y) + "j"
        return out
    def __repr__(self):
        out = "" + str(self.x) + "i"
        if self.y < 0:
            out += " - " + str(-self.y) + "j"
        else:
            out += " + " + str(self.y) + "j"
        return out

    def magnitude(self):
        return math.hypot(self.x, self.y)
q = intVector(3, 4)
print("q.magnitude() = {0}".format(q.magnitude()))
r = intVector(3, -4)
print("r.magnitude() = {0}".format(r.magnitude()))
```

Now run this.

```
$ python IntVector.py
q.magnitude() = 5.0
r.magnitude() = 5.0
$
```

You may add as many regular methods to your class as you wish. Python has a rich collection of hooks for overriding the behavior of operators such as +, -, *, / and **.

paragraphProgramming Exercises

1. Implement the hook

   `def __add__(self, other):`

   so a vector will add itself to the vector `other`.
2. Implement the hook

   `def __sub__(self, other):`

so a vector subtract `other` from itself and return the result

3. Implement the hook

   ```
   def __eq__(self, other):
   ```

   and have a the the vector `self` report if it is equal to the vector `other`.

3 A Case Study: Playing Cards

Let us imagine that we are writing a game involving playing cards. Using the class apparatus, we can create a new data type that represents a playing card.

We will adopt the Java naming convention: each class we create will reside in a file with the same name as our class. Also, we will capitalize all class names.

We begin by creating an empty class like so.

```
class Card(object):
    pass
```

Observe that the class statement is a boss statement, and therefore has a colon at the end. It owns a block of code. Recall that if you want to have an empty block of code, you must place a `pass` statement in it.

3.1 A Design Decision

What does a card need to know to be a card? We will deal with the standard Bridge deck of 52 cards here. Each card is determined by a rank of 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack (J), Queen (Q), King (K) and Ace (A). We have listed the ranks in ascending order here. There are four suits, Clubs, Diamonds, Hearts and Spades. A card's identity is determined completely by its suit and its rank.

What does a card need to know? We could keep track of cards by recording their ranks and suits. However there is a simpler and nicer way to do this. What we shall do here is to connect each card to an integer in `range(0,52)`. This is an implementation detail; we will enable our cards separately to tell their ranks and suits. We will call this integer `number`.

You can accomplish this task several ways; the one shown here is nice and compact. However, you can try using another way to keep track of cards. You have the freedom to choose here!

What should a card be able to do? A card needs to be able to tell us its suit and rank. A card should know whether or not it outranks another card. We

should be able to make a card either by specifying its number or by specifying nothing and getting a random card.

In a class we specify state variables, which constitute the things objects created from the class know and methods, which constitute what instance of the class can do.

This kind of design process occurs in all object-oriented languages including Ruby, Python, Java and C++.

3.2 Getting to Know our Number

Our `Card` class will have a state variable named number and it will know the ranks and suits of cards. Since we are coding inside of the `Card` class, we "are" a card. Our name inside the class is self. This is a special Python language keyword.

When a `Card` is first created, a special method called `__init__` swings into action; we use this to teach the class what it needs to know. First let us teach the class its number.

```
class Card(object):
def __init__(self, number):
    self.number = number
```

This worked but the default method of printing out an instance of a class is pretty uninformative. In the session below we created an instance of `Card` number 5, but not much shows when we try to print it.

```
>>> from Card import *
>>> c = Card(5)
>>> print c
<Card.Card instance at 0xb7eacf4c>
>>>
```

Now we will add a the string hook to our class, which will create a string representation of a `Card` object.

```
class Card(object):
def __init__(self, number):
    self.number = number
def __str__(self):
    return "Card#" + str(self.number)
```

Whenever we use print, we will see this string representation put to `stdout`.

```
>>> from Card import *
>>> c = Card(5)
>>> print c
Card#5
>>> c
<Card.Card instance at 0xb801af4c>
>>>
```

It is also nice to have a card print nicely right at the interactive prompt. To do this, implement the `__repr__` method just as you did `__str__`. Whatever you print should be a valid Python expression. Here we will show our card as a string literal. We show the implementation here.

```
def __repr__(self):
    return "\"Card#" + str(self.number) + "\""
```

Here is the result.

```
>>> from Card import *
>>> c = Card(5)
>>> c
"Card#5"
>>>
```

Implementing the `Card` Class Now let us make instances of of our class cognizant of ranks and suits. We will also "show our cards" about our implementation.

```
class Card(object):
    def __init__(self, number):
        self.ranks = ['2', '3', '4', '5', '6', '7', '8', '9',
        '10', 'J', 'Q', 'K', 'A']
        self.suits = ['clubs', 'diamonds', 'hearts', 'spades']
        self.number = number
    def __str__(self):
        return self.ranks[self.number % 13] + " of "
        + self.suits[self.number//13]
    def __repr__(self):
        return "' + self.ranks[self.number % 13] + " of "
        + self.suits[self.number//13] + "'"
```

Notice that our cards are determined by an integer and how this simple arabesque with lists make it quick and easy to print out a card's name.

```
>> from Card import *
>>> c = Card(5)
>>> print c
7 of clubs
>>> c = Card(44)
>>> c
"7 of spades"
>>>
```

Now that we have a simple working class, let us make some observations. When a card is created, the `__init__` method is called. The code

```
c = Card(5)
```

causes a card to be created and its number to be set to 5. The variable `self` inside of the class represents the card's identity. So, `self.number` means `self`'s number. The variable number is just a parameter for the `__init__` method.

Also, you should notice that self is a required argument in all of the class's methods. Syntactically, notice that all method headers below the class statement are indented; this reflects the class's ownership over all of its methods.

We shall take advantage of default arguments to give a default method for creating instance of `Card`. It will choose a card at random.

```
import random
class Card(object):
    def __init__(self, number = -1):
        random.seed()
        self.ranks = ['2', '3', '4', '5', '6', '7', '8', '9',
        '10', 'J', 'Q', 'K', 'A']
        self.suits = ['clubs', 'diamonds', 'hearts', 'spades']
        self.number = number if number >= 0 else random.randint(0,51)
    def __str__(self):
        return self.ranks[self.number % 13] + " of "
        + self.suits[self.number/13]
    def __repr__(self):
        return "\"" + self.ranks[self.number % 13] + " of "
        + self.suits[self.number/13] + "\""
if __name__ == '__main__':
    for k in range(1,10):
    c = Card()
    print c
```

Running this program results in the following. If you run this yourself, you will very likely get ten cards different from those shown here.

```
$ python Card.py
K of clubs
J of diamonds
J of hearts
K of hearts
10 of spades
4 of diamonds
10 of clubs
K of hearts
2 of diamonds
$
```

3.3 Adding Regular Methods to the Class

Now we add in methods for displaying suit and rank of a `Card`. These are regular methods which we name and define ourselves. You can add as many regular methods as you wish to a class. The rules for naming methods are exactly the same as they are for variables or regular functions.

```
import random
class Card(object):
    def __init__(self, number = -1):
        random.seed()
        self.ranks = ['2', '3', '4', '5', '6', '7', '8', '9',
        '10', 'J', 'Q', 'K', 'A']
        self.suits = ['clubs', 'diamonds', 'hearts', 'spades']
        self.number = number if number >= 0 else random.randint(0,51)
    def __str__(self):
        return self.ranks[self.number % 13] + " of "
        + self.suits[self.number/13]
    def __repr__(self):
        return "\"" + self.ranks[self.number % 13] + " of "
        + self.suits[self.number/13] + "\""
    def rank(self):
        return self.ranks[self.number % 13]
    def suit(self):
        return self.suits[self.number/13]
```

This interactive session provides a simple test of our work.

```
>>> from Card import *
>>> c = Card(51)
>>> print c
A of spades
>>> c.rank()
```

```
'A'
>>> c.suit()
'spades'
>>>
```

4 Resolving our Identity Crisis

Here is a surly little problem.

```
>>> d = Card(51)
>>> c == d
False
>>> print c
A of spades
>>> print d
A of spades
>>>
```

However, this problem is akin to the problem we had with printing. By default, when we create a class, the operator == checks for equality of identity, not of value. We fix this with the `__eq__` hook.

```
import random
class Card(object):
    def __init__(self, number = -1):
        random.seed()
        self.ranks = ['2', '3', '4', '5', '6', '7', '8', '9',
        '10', 'J', 'Q', 'K', 'A']
        self.suits = ['clubs', 'diamonds', 'hearts', 'spades']
        self.number = number if number >= 0 else random.randint(0,51)
    def __str__(self):
        return self.ranks[self.number % 13] + " of " +
        self.suits[self.number/13]
    def __eq__(self, other):
        return self.number == other.number
    def rank(self):
        return self.ranks[self.number % 13]
    def suit(self):
        return self.suits[self.number//13]
```

We now have the makings of a new, general–purpose type for programming with playing cards. Now you will perform some simple tasks with this class.

5 Using our `Card` Class

This little sample program shows how to program with the class. Place this code in a file called `exercise.py`.

```
from Card import Card
c = Card()
print("c = {0}".format(c))
d = Card(23)
print("d = {0}".format(d)
print("c.rank() = {0}".format(c.rank()))
print("d.suit() = " + str(d.suit()))
```

This is what happens when we run the code.

```
$ python exercise.py
c = Q of hearts
7
d = Q of diamonds
c.rank() = 7
d.suit() = diamonds
$
```

Programming Exercises

1. Make a list containing a full deck of cards. Look in the random library and figure out how to shuffle the deck.
2. A poker hand is a sample without replacement of five cards from a bridge deck. Make a function dealHand() that generates a poker hand (5 cards).
3. Make a function isFlush() that checks if a poker hand contains cards all of the same suit.
4. Make a function isStraight() that checks if a poker hand contains cards with five consecutive ranks.
5. Make a function isPair() that checks if a poker hand contains cards two cards of equal rank and three other cards of different ranks.

6 Shouldn't a Deck of Cards be an Object?

The short answer is: yes. When card games are played in casinos, several decks are combined to create a reservoir of cards called a *shoe*. We will create a class for shoes of cards and have the shoe deal cards (in a list).

The main information we need is the number of decks in the shoe, and the order of the cards in the decks.

```
import random
from Card import Card

class Shoe:
    def __init__(self, howMany = 1):
        self.howMany = howMany
        self.cards = []
        for k in range(howMany):
            for j in range(51):
                self.cards.append(Card(j))
```

We will now see how this creates a shoe of cards. We begin by making our shoe know how many decks it contains. That is done by this line of code.

```
        self.howMany = howMany
```

The programmer using this code will say something like

```
deck = Shoe(2)
```

and this will create a two-deck shoe. If no value is given to `Shoe()`, it will create by default a one-deck shoe.

Now we make our shoe of cards.

```
        self.cards = []
        for k in range(howMany):
            for j in range(51):
                self.cards.append(Card(j))
```

Our shoe is learning its cards. At first it is empty. The loop populates it with the appropriate number of decks. At the end of this code, all of the cards are sorted in numeric order. That is not desirable and could get us shot in a less than friendly card game. To shuffle the deck using `random`'s `shuffle` mechanism, we create a method `shuffle`.

```
    def shuffle(self):
        random.shuffle(self.cards)
```

We now have a shoe of cards with the specified number of decks, nicely shuffled and ready for the dealer's table.

Now we are going to have the shoe deal cards from itself. We will return the cards (even one card) in a list of cards. The list method `pop()` comes in handy. It takes an item off the list, removes it from the list and returns the item. This comes in very handy here.

```
    def deal(self, n = 1):
        cardsToBeGiven = []
        for k in range(n):
            cardsToBeGiven.append(self.cards.pop())
        return cardsToBeGiven
```

Programming Exercises It is useful to know if a card is a face card (J, K, Q) or if it is an ace. This is true if you wish to write a blackjack game. Knowing a card's color is important for solitaire games. Add these methods to your card class.

1. Implement a method `isFace` that returns `True` if a card is a face card and `False` otherwise.
2. Implement a method `isAce` that returns `True` if a card is an ace and `False` otherwise.
3. Implement a method `isRed` that returns `True` if a card is a red card (hearts or diamonds) and `False` otherwise.
4. Implement a method `isBlack` that returns `True` if a card is a black card (spades or clubs) and `False` otherwise.

7 Case Study: Fractions

Python has a built-in class for these but we will create an example class here to do extended–precision rational arithmetic and use it do do some interesting things such as producing very close rational approximations of roots of numbers.

To begin we ask: what does a fraction need to know? It needs to know its numerator and denominator. So we might begin like so.

```
class Fraction(object):
    def __init__(self, num = 0, denom = 1):
        self.num = num
        self.denom = denom
    def __str__(self):
        return "%s/%s" %(self.num, self.denom)
    def __repr__(self):
        return "\"%s/%s\"" %(self.num, self.denom)
```

```
f = Fraction(1,2)
print(f)
```

Now run this.

```
$ python Fraction.py
1/2
$
```

Now we see an irritation coming. Add more cases.

```
class Fraction(object):
    def __init__(self, num = 0, denom = 1):
        self.num = num
        self.denom = denom
    def __str__(self):
        return "%s/%s" %(self.num, self.denom)
    def __repr__(self):
        return "\"%s/%s\"" %(self.num, self.denom)
f = Fraction(1,2)
print(f)
g = Fraction(5,10)
print(g)
print(f == g)
h = Fraction(1,-2)
print(h)
```

Now run this.

```
$ python Fraction.py
1/2
5/10
False
1/-2
$
```

Notice how 5/10 is not equal to 1/2. Then notice the ugliness of having a negative in the denominator. We get the clue. *All fractions must be born fully reduced and with any negative in the numerator.* This means some work.

It is easy to fix the negative sign problem in the `__init__` method. We do this as follows.

```
    def __init__(self, num = 0, denom = 1):
        if denom < 0:
```

```
            num = -num
            denom = -denom
        self.num = num
        self.denom = denom
```

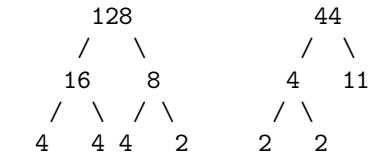
Et Voila! One headache is gone.

```
$ python Fraction.py
1/2
5/10
False
-1/2
$
```

Now we have another problem: getting fractions reduced. It's time for a trip back to Mrs. Wormwood's classroom and computing prime factorizations. To do this, we compute the greatest common divisor (gcd) of the numerator and denominator, then divide this out. Consider the fraction

$$\frac{128}{44.}$$

We create factor trees.



From these we see that $128 = 2^7$ and $44 = 2^2 \cdot 11$. The largest power of 2 that is a common factor to both numbers is 4. There is no common power of 11, so $\gcd(128, 44) = 4$. The gcd function is the *greatest common divisor* for two integers.

Thus we have

$$\frac{128}{44} = \frac{128/\gcd(128,44)}{44/\gcd(128,44)} = \frac{32}{11}.$$

Another interesting observation to make is that changing signs does not affect divisibility, so for any integers a and b that are not both zero,

$$\gcd(\pm a, \pm b) = \gcd(a, b).$$

Take note that all integers divide 0 evenly, so if $a \neq 0$, $\gcd(a, 0) = |a|$. This function is not defined if $a = b = 0$. To wit, the domain of gcd is the set of all ordered pairs of integers save for $(0, 0)$. Finally, by the symmetry of this definition, it is easy to see that $\gcd(a, b) = \gcd(b, a)$ provided a and b are not both 0.

7.1 An Algorithmic Interlude

The Wormwood method for reducing fractions is costly and slow. Its fault lies in the computation of the gcd. There is a faster way. If you don't see the flaw yet, consider this problem. Find

$$\gcd(39803419043890, 34198913098105).$$

That looks mighty ugly. It is time to fade back and try another strategy. Here is the key idea.

Theorem. *Let a, b, q, and r be any integers. Then if $b = aq + r$, $\gcd(b, a) = \gcd(a, r)$.*

Before we discuss why this works, let us deploy it. You have

$$128 = 44(2) + 40,$$

so

$$\gcd(128, 44) = \gcd(44, 40).$$

Observe that we now have a smaller problem. Hey, if this works once, let us this again. You have

$$44 = 40(1) + 4,$$

so

$$\gcd(44, 40) = \gcd(40, 4).$$

One more time ...

$$40 = 4(10) + 0,$$

so

$$\gcd(40, 4) = \gcd(4, 0) = 4.$$

We now have

$$\gcd(128, 44) = 4.$$

This appears to be a process that is readily controllable be a loop. We can grid this down until r is 0 in a `while` loop.

Before we do this, let us learn why the theorem works. If a and b are integers, we will write $a \mid b$ if a divides b evenly, i.e if there is some integer a so that $b = aq$.

Suppose that d is common divisor of a an b and that $b = aq + r$. Since d is a common divisor of a and b there are integers s and t so $a = sd$ and $b = td$. Then

$$r = b - aq = td - sdq = d(t - sq).$$

But $t - sq$ is an integer, so $d \mid r$. We have just show that ever common divisor of a and b is a common divisor of a and r.

Now suppose that d is a common divisor of a and r. Then we can choose integers u and v so $a = ud$ and $r = vd$. Then

$$b = aq + r = udq + dv = d(uq + v).$$

We see that d is a common divisor of a and b.

Since a and b have exactly the same common divisors as a and r, our result follows.

7.2 Creating a gcd function

Let us look at our previus calculation in tabular form. Since we will use `q = b // a`, we never really need to know about `q`. In the end we use the bottom row to see that $\gcd(128, 44) = \gcd(4, 0) = 4$.

b	a	r
128	44	40
44	40	4
40	4	0
4	0	0

So, while `r` is not 0, keep going. At the end return a. Before we begin, we discard information about signs so everything stays positive.

```
def gcd(b,a):
    if b < 0:       #discard signs
        b = -b
    if a < 0:
        a = -a
    r = 1  ##fool the loop into running
    while r > 0:
        r = b % a
        b, a = a, r
    return b
```

7.3 Now Back to Fractions

Let us now insert our gcd function and avail ourselves of it. Our fractions will now be born fully reduced and with any negative in the numerator.

```
def gcd(b,a):
    if b < 0:
        b = -b
```

```
    if a < 0:
        a = -a
    r = 1  ##fool the loop into running
    while r > 0:
        r = b % a
        b, a = a, r
    return b
class Fraction(object):
    def __init__(self, num = 0, denom = 1):
        if denom < 0:
            num = -num
            denom = -denom
        #do not forget to use // not /
        self.num = num//gcd(num, denom)
        self.denom = denom//gcd(num,denom)
    def __str__(self):
        return "%s/%s" %(self.num, self.denom)
    def __repr__(self):
        return "\"%s/%s\"" %(self.num, self.denom)
f = Fraction(1,2)
print(f)
g = Fraction(5,10)
print(g)
print(f == g)
h = Fraction(1,-2)
print(h)
print(gcd(128,44))
```

Now run this. All of the problems with the negative and reduction are gone, save for the lingering problem of equality.

```
$python Fraction.py
1/2
1/2
False
-1/2
4
$
```

7.4 Static Methods

It would probably would be better if our `gcd` function were are part of our class. Notice that is makes no use of the state of a `Fraction` object. It would be silly to have `self` as an argument.

Were we to do so, a copy of this function's code would have to be included in every instance of `Fraction`. Can we create just once instance of this resource and share it among all instances of `Fraction`? Happily, the answer is "yes."

What we do is use the `@staticmethod` declaration. So, we modify our class as follows. Notice how we use the `@staticmethod` declaration just before the static method (`gcd`) and that we preface all calls to `gcd` with the class name `Fraction`.

```
class Fraction(object):
    def __init__(self, num = 0, denom = 1):
        if denom < 0:
            num = -num
            denom = -denom
        #do not forget to use // not /
        ##include a Fraction. before calls to gcd
        self.num = num//Fraction.gcd(num, denom)
        self.denom = denom//Fraction.gcd(num,denom)
    def __str__(self):
        return "%s/%s" %(self.num, self.denom)
    def __repr__(self):
        return "\"%s/%s\"" %(self.num, self.denom)
    @staticmethod
    def gcd(b,a):
        if b < 0:
            b = -b
        if a < 0:
            a = -a
        r = 1  ##fool the loop into running
        while r > 0:
            r = b % a
            b, a = a, r
        return b
f = Fraction(1,2)
print(f)
g = Fraction(5,10)
print(g)
print(f == g)
h = Fraction(1,-2)
print(h)
print(Fraction.gcd(128,44))
```

Note that if you use the `@staticmethod` declaration, you *cannot* use `self` as an argument in the method. Class state variables cannot be visible to static methods.

7.5 Giving Fractions Relational Operators

There is a Python 2/3 fork in the road here. In Python 2, you would define a special method `__cmp__` to define the relational operators. This method would have this appearance.

```
def __cmp__(self, other):
    ## return a negative number if self<other
    ## return a positive number if self >other
    ## return a 0 if self and other are equal
```

We will, instead, do things the Python 3 way. Here are the hooks for the relational operators. We will implement all of these in our fraction class.

`__lt__`	This defines the `<` operator
`__gt__`	This defines the `>` operator
`__le__`	This defines the `<=` operator
`__ge__`	This defines the `>=` operator
`__ne__`	This defines the `!=` operator
`__eq__`	This defines the `==` operator

Suppose we have fractions a/b and c/d and we wish to compare them. Our `Fraction` class puts the negative in the numerator, so we can assume that $b > 0$ and $d > 0$. If we wish to test for

$$\frac{a}{b} < \frac{c}{d},$$

we multiply both sides by bd and simply test for

$$ad < bc$$

So if we are implementing `<` we do so as follows. Using $a =$ `self.num`, $b =$ `self.denom`, $c =$ `other.num`, and $d =$ `other.num`, we get this implementation.

```
def __lt__(self, other):
    return self.num*other.denom < self.denom*other.num
```

Now we implement all of the relational operators. Note how we do this for equality and inequality. No cross-multiplication is required for these. We also inserted test code for our relational operators.

```
class Fraction(object):
    def __init__(self, num = 0, denom = 1):
        if denom < 0:
            num = -num
```

```
            denom = -denom
        #do not forget to use // not /
        ##include a Fraction. before calls to gcd
        self.num = num//Fraction.gcd(num, denom)
        self.denom = denom//Fraction.gcd(num,denom)
    def __str__(self):
        return "%s/%s" %(self.num, self.denom)
    def __repr__(self):
        return "\"%s/%s\"" %(self.num, self.denom)
    def __lt__(self, other):
        return self.num*other.denom < self.denom*other.num
    def __gt__(self, other):
        return self.num*other.denom > self.denom*other.num
    def __le__(self, other):
        return self.num*other.denom <= self.denom*other.num
    def __ge__(self, other):
        return self.num*other.denom >= self.denom*other.num
    def __eq__(self, other):
        return self.num == other.num and self.denom == other.denom
    def __ne__(self, other):
        return self.num != other.num or self.denom != other.denom
    @staticmethod
    def gcd(b,a):
        if b < 0:
            b = -b
        if a < 0:
            a = -a
        r = 1  ##fool the loop into running
        while r > 0:
            r = b % a
            b, a = a, r
        return b
##main routine with test code
if __name__ == "__main__":
    f = Fraction(1,2)
    g = Fraction(5,10)
    h = Fraction(1,3)
    print("%s < %s: %s" % (f, g, f < g))
    print("%s > %s: %s" % (f, g, f > g))
    print("%s <= %s: %s" % (f, g, f <= g))
    print("%s >= %s: %s" % (f, g, f >= g))
    print("%s == %s: %s" % (f, g, f == g))
    print("%s != %s: %s" % (f, g, f != g))
    print("%s < %s: %s" % (f, h, f < h))
    print("%s > %s: %s" % (f, h, f > h))
    print("%s <= %s: %s" % (f, h, f <= h))
```

```
    print("%s >= %s: %s" % (f, h, f >= h))
    print("%s == %s: %s" % (f, h, f == h))
    print("%s != %s: %s" % (f, h, f != h))
```

Now run this code at the command line.

```
$ python Fraction.py
1/2 < 1/2: False
1/2 > 1/2: False
1/2 <= 1/2: True
1/2 >= 1/2: True
1/2 == 1/2: True
1/2 != 1/2: False
1/2 < 1/3: False
1/2 > 1/3: True
1/2 <= 1/3: False
1/2 >= 1/3: True
1/2 == 1/3: False
1/2 != 1/3: True
$
```

7.6 Adding Arithmetic

Next, we insert code for the arithmetic hooks. Let us begin with addition. Mrs Wormwood tells us that for fractions a/b and c/d, we have

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}.$$

So, when implementing `__add__`, we proceed as follows.

```
def __add__(self, other):
    return Fraction(self.num*other.denom + self.denom*other.num, self.denom*other.denom)
```

While we are here, can define `+=` using `__iadd__`.

```
def __iadd__(self, other):
    self.num, self.denom = self.num*other.denom + self.denom*other.num, self.denom*other.den
    return self
```

We proceed similarly for subtraction

```
def __sub__(self, other):
    return Fraction(self.num*other.denom - self.denom*other.num, self.denom*other.denom)
```

While we are here, can define += using `__iadd__`.

```
def __isub__(self, other):
    self.num, self.denom = self.num*other.denom - self.denom*other.num, self.denom*other.den
    return self
```

You will also want the unary prefix operator - to work.

```
def __neg__(self):
    return Fraction(-self.num, self.denom)
```

We then add in multiplication and division

```
def __mul__(self, other):
    return Fraction(self.num*other.num, self.denom*other.denom)
def __imul__(self, other):
    self.num, self.denom = self.num*other.num, self.denom*other.denom
    return self
def __truediv__(self, other):
    return Fraction(self.num*other.num, self.denom*other.denom)
def __itruediv__(self, other):
    self.num, self.denom = self.num*other.num, self.denom*other.denom
    return self
```

Finally, we add powers.

```
def __pow__(self, n):
    if n < 0:
        n = -n
        return Fraction(self.denom**n, self.num**n)
    Fraction(self.num**n, self.denom**n)
def __ipow__(self, n):
    if n < 0:
        n = -n
        self.denom, self.num = self.num*n, self.denom**n
    self.num, self.denom = self.num*other.num, self.denom*other.denom
    return self
```

Programming Exercises

1. You can control how a `Fraction` casts to an integer using the hook `__int__(self)`. See how floats cast to integers and implement this hook in a similar manner.

2. We define the *harmonic numbers* H_n as follows.

$$H_n = \sum_{k=1}^{n} H_k.$$

   Write a static method `harmonic(n)` that computes the nth harmonic number. Can you compute $H1000$? H_{5000}?

3. How might you create a `__float__` method to convert a `Fraction` to a floating point number? This is not as easy as it looks. For example, if both numerator and denominator of a `Fraction` have a very large number of digits, attempting to cast numerator and denominator to floats and dividing will not do, since this cast will return `infinity` for both? You will also have to decide when to return 0 and when to return `infinity`. Floating point numbers carry about 17 digits of precision. Look up the standard and understand how big a floating point number can be. Test the result mercilessly.

8 Case Study: A Calendar Date Class

This is of interest because it will have a wide variety of hooks and special methods. First we ask, "What should a date know?" It should know a day, month, and year. We will force the user to specify all three when specifying `Date`. No defaults will be used

```
class Date(object):
    def __init__(self, day, month, year):
        self.day = day
        self.month = month
        self.year = year
```

Next, we add a method so we can see a date get printed. Note the addition of tuples to represent month names and the names of the days of the week.

```
class Date(object):
    def __init__(self, day, month, year):
        self.monthNames = ("", "January", "February", "March", "April", "May", "June", "July
        self.dayNames = ("Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "
        self.day = day
        self.month = month
        self.year = year
    def __str__(self):
        return "%s %s %s" % (self.day, self.monthNames[self.month], self.year)

print(Date(22, 10, 2015))
```

Now run this program and see our shiny new date print out.

```
$ python Date.py
22 October 2015
```

Next, we make enable out `Date` objects to tell us their ordinal position in the year (Feb 12 is the 43rd day of the year), and how many days are left in the year. While we are under the hood, let us also create a static method that checks a year to see if it leaps. Recall the rule.

1. A leap year occurs in years divisible by 4, EXCEPT
2. when a year is divisible by 100 when it does not, EXCEPT
3. when a year is divisible by 400 it does!

```
class Date(object):
    def __init__(self, day, month, year):
        self.monthNames = ("", "January", "February", "March", "April", "May", "June", "July
        self.dayNames = ("Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "
        self.day = day
        self.month = month
        self.year = year
    @staticmethod
    def isLeap(y):
        out = 0
        if y % 4 == 0:
            out += 1
            if y % 100 == 0:
                out -= 1
                if y % 400 == 0:
                    out += 1
        return out
    def __str__(self):
        return "%s %s %s" % (self.day, self.monthNames[self.month], self.year)
    def dayInYear(self):
        monthLengths = (0, 31, 28 +  self.isLeap(self.year), 31, 30, 31, 30, 31, 31, 30, 31,
        return self.day + sum(monthLengths[:self.month])
    def daysLeftInYear(self):
        return 365 + self.isLeap(self.year) - self.dayInYear()
print(Date(22, 10, 2015))
print(Date(26,5,2015).dayInYear())
```

We will now create a "service function" that computes for any given date its number of days since the fictitious date 1 January 1. If this bugs you a whole lot, you can base the calculation on 1 January 1753; this marks the first whole year on the modern calendar.

Let us sketch out our thinking. Suppose today is 23 October 2015. To begin we compute the number of days from 1 January 1 to 31 December 2014. Here is what we do.

Non leap days	`2015*365`	735475
Years divisible by 4	`2015//4`	500
Years divisible by 100	`-2015//100`	-20
Years divisible by 400	`2015//400`	4
Total		735955

We now see how to write this function. To get to the current date, we just add `dayInYear()` to this result.

```
    def dayIndex(self):
        y = self.year - 1
        out = 365*y
        out += y//4 - y //100 + y//400 #leap adjustments
        out += self.dayInYear()
        return out
```

We now will computer the dayIndex for 26 October 2015 and mod out by 7. This date is a Monday (authoritative, since this is being typed on 26 October 2015). Running this produces a result of 1. So, if we compute the day index of a date and mod out by 7, a 1 gives a Monday, a 2 gives a Tuesday, etc. We can now write `dayInWeek()`.

```
    def dayInWeek(self):
        self.dayNames[self.dayIndex()%7]
```

Next, we will define subtraction for two dates. The difference will be the number of days from one date to the other. The `dayIndex` method makes this simple

```
    def __sub__(self, other):
        return self.dayIndex() - other.dayIndex()
```

Here is the complete code.

```
class Date(object):
    def __init__(self, day, month, year):
        self.monthNames = ("", "January", "February", "March", "April", "May", "June", "July
        self.dayNames = ("Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "
        self.day = day
        self.month = month
        self.year = year
    @staticmethod
    def isLeap(y):
```

```
        out = 0
        if y % 4 == 0:
            out += 1
            if y % 100 == 0:
                out -= 1
                if y % 400 == 0:
                    out += 1
        return out
    def __str__(self):
        return "%s %s %s" % (self.day, self.monthNames[self.month], self.year)
    def __sub__(self, other):
        return self.dayIndex() - other.dayIndex()
    def dayIndex(self):
        y = self.year - 1
        out = 365*y
        out += y//4 - y //100 + y//400 #leap adjustments
        out += self.dayInYear()
        return out
    def dayInYear(self):
        monthLengths = (0, 31, 28 +  self.isLeap(self.year), 31, 30, 31, 30, 31, 31, 30, 31,
        return self.day + sum(monthLengths[:self.month])
    def dayOfWeek(self):
        return self.dayNames[self.dayIndex()%7]
    def daysLeftInYear(self):
        return 365 + self.isLeap(self.year) - self.dayInYear()
print(Date(22, 10, 2015))
print(Date(26,5,2015).dayInYear())
print(Date(26,10,2015).dayIndex())
print(Date(26,10,2015).dayIndex()%7)
print(Date(26, 5, 1957).dayOfWeek())
print(Date(26,10,2015) - Date(4,7,1776))
```

Programming Exercise

1. Add a method `tomorrow()` that returns the date for the next day.
2. Add a method `yesterday()` that returns the date for the previous day.
3. Add a method `nextWeekDay()` that returns the date for the next weekday day.
4. Add a method `yesterday()` that returns the date for the previous weekday day.
5. Here is an ugly challenge. Add a static method

   ```
   def index2Date(index):
       return Date()
   ```

which returns the `date` corresponding to the index passed it

6. Now define + for a `Date` and an integer `n` as follows.

```
def __add__(self, index, n):
    return Date()
```

Have this return the date `n` days from `self`.