# Chapter 5, Creating Custom Objects

John M. Morrison

January 6, 2020

## Contents

## 0 Introduction

We have learned that objects have state, identity, and behavior. We have worked with a wide variety of objects: lists, tuples, integers, and strings are examples. We can leverage these to solve big programming problems with a surprisingly small amount of code.

We have learned that we can create functions, which allow us to isolate a task and store a procedure for solving it under a name. These organizational features give Python a great deal of clarity and they cut clutter in your code.

Now it is time to go to the next level and ask this question *How can we create our own custom objects?* For example we might like to create a card game such as Poker. It would be useful to be able to create our own data type for representing playing cards.

We will begin by discussing a method for creating a custom object on the fly you might only use once. We will build the object using the "stick building" technique. This will allow us to attach state and methods to the object.

All objects in Python are created using the `class` mechanism; a class is a blueprint for an object.

## 1 Stick-Building Python Objects

We will now create a class named `First`.

```
>>> class First(object):
...     pass
...
```

Next, we create an object of this class's type. Since the class just contains a `pass` (do-nothing) statement, the object resulting has no state and no behavior. It does, however, have an identity.

```
>>> f = First()
>>> f
<__main__.First object at 0x104148668>
```

Now let us make another instance of our class

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

We now have objects `f` and `g` floating around. We can attach state 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'
```

We can even attach a method the object `f` like so.

```
>>> f.doSomething = lambda y: print(f.x + " is printing " + str(y))
>>> f.doSomething(42)
This is x is printing 42
```

Our funny little object even has a type.

```
>>> type(f)
<class '__main__.First'>
```

You can continue in this way, adding state and methods to the objects `f` and `g`. However, if you intend to mint lots of objects of your new type, this arrangement is clearly not satisfactory.

## 2 Making a Class

Now let us do something useful. 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 type of method called a *hook* for this purpose. 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.

\begin{minted}{shell-session}
unix> 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.

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

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.

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

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: Calendar Dates from Scratch

As you know calendar dates are a type of data riddled with idosyncracies and funky exceptions. The goal here is to create a nice interface for working with dates and to understand how Python's library works. This is a "look from the inside."

Let us begin. First we will set up our init hook.

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

Certain pieces of information in this class should be *static*; to wit, these are shared data for all objects created from this class. Let us add lists for the month names and the names of days of the week. The reason for this is that when the class is read into memory, only one copy of static items is needed.

```
class Date(object):
    monthNames = ["", "January", "February", "March",
        "April", "May", "June", "July", "August", "September",
        "October", "November", "December"]
    dayNames = ["Sunday", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday"]
    def __init__(self, day, month, year):
        self.day = day
        self.month = month
        self.year = year
```

How do you access these? Observe.

```
>>> from Date import Date
>>> Date.monthNames[2]
'February'
>>> Date.dayNames[5]
'Friday'
```

Static data are called by using the class name. Notice the empty string in the list `monthNames`; we do this so that 1 is associated with January and 2 with February, etc, because it's burnt into our brains.

## 3.1 Put on a strring Performance

As of yet, we can't see our dates print and we can't see them in an interactive session. So, we will get the str and repr hooks working.

```
def __str__(self):
    return "{0} {1} {2}".format(self.day,
        Date.monthNames[self.month], self.year)
def __repr__(self):
    return "'{0} {1} {2}'".format(self.day,
        Date.monthNames[self.month], self.year)
```

Let us now see what things look like

```
>>> d = Date(25, 12, 2019)
>>> print(d)
25 December 2019
>>> d
'25 December 2019'
```

## 3.2 A Leap of Faith

The next bear we must wrestle with is that of the business of leap years. In case you don't know, here are ther rules.

1. If a year is divisible by 4 it leaps.
2. Every 100 years, there is an exception. For instance, 29 February 1900 does not exist.
3. Every 400 years there is an exception to the exception! The day 29 February 2000 was a very special day and almost no one knew about that!

Let us write a function that returns 1 if a year leaps and 0 otherwise and name it `leapAdjust`.

```
    def leapAdjust(year):
        out = 0
        if year % 4 == 0:
            out += 1
            if year % 100 == 0:
                out -= 1
                if year % 400 == 0:
                    out += 1
        return out
```

This method does not depend on any object's state so we will make it a static (shared) method. When we do so, it is a good idea to apply the *decorator* `@staticmethod`.

```
class Date(object):
    monthNames = ["", "January", "February",
    "March", "April", "May", "June", "July",
    "August", "September", "October", "November",
    "December"]
    dayNames = ["Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday", "Friday", "Saturday"]
    @staticmethod
    def leapAdjust(year):
        out = 0
        if year % 4 == 0:
            out += 1
            if year % 100 == 0:
                out -= 1
                if year % 400 == 0:
                    out += 1
        return out
    def __init__(self, day, month, year):
        self.day = day
        self.month = month
        self.year = year
```

Note that the month lengths cannot be static, since the lengths of months in a given year depend upon whether the year leaps or not. We now add this to the init hook.

```
    self.monthLengths = [0, 31, 28 + Date.leapAdjust(self.year),
        31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
```

Here we see our init hook.

```
    def __init__(self, day, month, year):
        self.day = day
        self.month = month
        self.year = year
        self.monthLengths = [0, 31, 28 +
            Date.leapAdjust(self.year), 31, 30, 31,
            30, 31, 31, 30, 31, 30, 31]
```

## 3.3 I Take Exception!

Here is a stupid thing an end-user might do with Our Lovely Code.

```
>>> d = Date(32, 12, 2019)
>>> print(d)
32 December 2019
```

Heaven forfend! We shall defend the integrity of our work here. Add these two lines to the init hook.

```
        if self.month < 1 or self.month > 12:
            raise ValueError;
        if self.day < 1 or
        self.day > self.monthLengths[self.month]:
            raise ValueError
```

Punishment will be meted out to offenders.

```
>>> from Date import Date
>>> d = Date(32, 12, 2019)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/morrison/book/ppp/Date.py",
    line 44, in __init__
    raise ValueError;
ValueError
>>>
```

**Programming Exercies**

1. Write a method called `def tomorrow(self)` that computes for any given date the date of the next day.
2. Write a method called `def yesterday(self)` that computes for any given date the date of the previous day.

### 3.4 Fraternite, Egalite, Liberte!

Another piece of low-hanging fruit comes in the forms of the eq and neq hooks. We just check equality of state.

```
    def __eq__(self, other):
        return self.day == other.day and self.month == other.month
            and self.year == other.year

    def __neq__(self, other):
        return self.day != other.day or self.month != other.month
            or self.year != other.year
```

## 4 If it is Saturday, this must be Paris!

Now we staring a serious problem right in the eye. Given a date, how do we get the day of the week? We will restrict ourselves to the modern calender, which has been in force since 1752.

Ancillary to this, it will be helpful to know the ordianl position of a given date in its year. We will write a method `dayInYear` for this purpose. The idea is as follows. Add up the lengths of the previous months, then add on the day. Et Voila! For example for 26 May 2019, you do the following.

```
Jan   31
Feb   28
Mar   31
Apr   30
May   26
---------
     146
```

We have everything in place!

```
def dayInYear(self):
    return sum(self.monthLengths[:self.month]) + self.day
```

Now it's time to get shifty. Notice that `365%7 = 1`, so the passage of a year means that the calendar shifts one day. We are going to compute the accumulated shift for a date, keeping it between 0 and 6 by modding by 7.

We first get all of the shift up to the previous year. If we have a date `d`, let us put `y = d.year - 1`. Then, ignoring leap years we get `y` units of shift. Factoring in the exceptions and the exceptions to the exception, we get

```
    y + y//4 - y//100 + y//400
```

units of shift. We then mod this by 7. We get the rest of the shift from the ordinal position of the day in the year, which we already know how to compute. Our result can be computed like so. The variable `out` holds the total shift.

```
y = self.year - 1
out = (y + y//4 - y//100 + y//400)% 7
out += self.dayInYear()
out = out % 7
```

To set the shift, run it on a day. If your day is Tuesday, you will get 2. So, Tuesday is a 2, Wednesday is a 3, Thursday is a 4, Friday is a 5, Saturday is a 6, and . . . , Sunday is a 0! Here is our `dayOfWeek` method.

```
def dayOfWeek(self):
    y = self.year - 1
    out = (y + y//4 - y//100 + y//400)% 7
    out += self.dayInYear()
    out = out % 7
    return Date.dayNames[out]
```

## 5 I am afraid your `number` is up!

Inspired by what we have just done, we are going to serially number all dates by computing how many days have elapsed since a fictional "beginning." We will do the routine in the `dayOfTheWeek` method, but we will not mod out by 7.

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

Here is a fun thing we can do with this `number` method. We can write a method to subtract dates using the sub hook.

```
def __sub__(self, other):
    self.number() - other.number()
```

**Progrmaming Exercises** Use number to implement the following.

1. `__gt__`
2. `__lt__`
3. `__ge__`
4. `__le__`

**OK, Hotshots! A Serious Challenge** Here is a right hairy question to ponder: *Given a date's number, how do you reconstruct the date?* Make a static method called `dateFromNumber(num)` that computes a date from its number

**NC-17 Programming Exercises** Use number to implement the following.

1. Use `dateFromNumber` to define the hook

   ```
   def ___add__(self, n):
       #return the date n ndays from this date
       #n can be an integer positive or negative
   def __radd(self, n):
       ## allows you to wrte n + date as well as date + n.
   ```

2. Upgrade the subtract hook so it detects if a number is passed in as `other` instead of a date and which does the right thing.
3. Create the hook `__rsub__`.

# 6 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.

## 6.1 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++.

## 6.2 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#{0}".format(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.

```python
def __repr__(self):
    return "'Card#{0}'".format(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. We need lists that hold ranks and suits for cards. Observe that the list of ranks and the list of suits is the same of for all cards, so we will make those *static* or shared, variables. Here is how to do that. Notice that the static variables are "unselfish."

```python
class Card(object):
    #shared data for all cards
    ranks = ['2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K', 'A']
    suits = ['clubs', 'diamonds', 'hearts', 'spades']
    def __init__(self, number):
        self.number = number

Now let us see how to gain access to these static variables.
\begin{minted}{python}
>>> Card.ranks
['2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K', 'A']
>>> Card.suits
['clubs', 'diamonds', 'hearts', 'spades']
```

You can access them via the class name since they are static. The next step is to get the `__str__` and `__repr__` hooks up and running so we can see what we are doing. You can access them via the class name since they are static. The next step is to get the `__str__` and `__repr__` hooks up and running so we can see what we are doing.

```python
class Card(object):
    #static variables shared by all cards
    ranks = ['2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K', 'A']
    suits = ['clubs', 'diamonds', 'hearts', 'spades']
    def __init__(self, number):
        self.number = number
    def __str__(self):
        return "{0} of {1}".format(Card.ranks[self.number%13], Card.suits[self.number//13])
    def __repr__(self):
        return "'{0} of {1}'".format(Card.ranks[self.number%13], Card.suits[self.number//13]
```

It's time to test this. First we will test our `__str__` hook. Add this to our code.

```
def main():
    c = Card(0)
    d = Card(1)
    print(c)
    print(d)
if __name__ == "__main__":
    main()
```

Run it and see this.

```
unix> python Card.py
2 of clubs
3 of clubs
```

Next test the `__repr__` hook by opening a Python session and importing our class.

```
>>> from Card import Card
>>> for k in /ange(0,15):
...     print(Card(k))
...
2 of clubs
3 of clubs
4 of clubs
5 of clubs
6 of clubs
7 of clubs
8 of clubs
9 of clubs
10 of clubs
J of clubs
Q of clubs
K of clubs
```

```
A of clubs
2 of diamonds
3 of diamonds
>>>
```

We will now add some regular metods, one to return rank, one to return suit, and a third method that will return a numerical ranking for a card that is a number 0-12. This will be useful in comparing card ranks. First, let's write the methods reporting rank and suit.

```
    def rank(self):
        return Card.ranks[self.number % 13]
    def suit(self):
        return Card.suits[self.number//13]
```

Now for the numerical value.

```
   def numericalRank(self):
        return self.number % 13
```

## 6.3 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.

```
    def __eq__(self, other):
        return self.number == other.number
```

# 7 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(35)
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.

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

### 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.

## 8 A Little Interior Decoration

The `@property` decorator can shorten code and make it easer to understand. Let us apply it to our `rank`, `suit`, and `numericalRank` methods as follows.

```
    @property
    def rank(self):
        return Card.ranks[self.number % 13]
    @property
    def suit(self):
        return Card.suits[self.number//13]
    @property
    def numericalRank(self):
        return self.number % 13

    print(Card(51).rank)
    print(Card(51).numericalRank)
    print(Card(51).suit)
```

Now these methods can be called as if they were properties. Here is the result of runnning the Card class with these three lines.

```
A
12
spades
```

## 9 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.

**Hey, a Shoe is a collection! I want to to use the `for` loop!** To do this, we must implement two hooks, `__len__` and `__getitem__`. The first defines the builtin `len` function for our `Shoe` object. The second allows us to index into the `Shoe` object. If you have these two in place, you can use a `for` loop.

Add 'em.

```
    def __getitem__(self, n):
        return self.cards[n]
    def __len__(self):
        return len(self.cards)
```

Now do this and watch your cards print

```
s = Shoe()
for k in s:
    print (k)
```

## 10 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 "{}/{}".format(self.num, self.denom)
    def __repr__(self):
        return "'{}/{}'".format(self.num, self.denom)
f = Fraction(1,2)
print(f)
```

Now run this.

```
unix> python Fraction.py
1/2
unix>
```

Let us now add these four lines to our code. We will be unhappy.

```
g = Fraction(2,4)
print(g)
h = Fracton(-3, -6)
print(h)
```

Run this.

```
unix> python Fraction.py
1/2
2/4
-3/-6
```

Blecch. We find fauult here. Firstly, any self-respecting fraction allowed to be seen in public should be in fully-reduced form. Secondly, that negative on the bottom is ugly and we can easily get rid of it.

So, we will add a function to compute the greatest common divisor of two integers and we will make a provision to kick any negative sign upstairs. We will do this in the `__init__` hook so our Fractions are “born” fully reduced and with any negative in the numerator. While we are here, we will pitch a fit if some hapless client programmer passes in a zero denominator.

```
def gcd(b, a):
    while a > 0:
        b, a = a, b%a
    return b
class Fraction:
    def __init__(self, num = 0, denom=1):
        if denom == 0:
            raise ValueError
        d = gcd(num, denom)
        self.num = num//d
        self.denom = denom//d
        if self.denom < 0:
            self.denom = -self.denom
            self.num = -self.num

    def __str__(self):
        return "{}/{}".format(self.num, self.denom)
    def __repr__(self):
        return "'{}/{}'".format(self.num, self.denom)
```

Run again

```
unix> python Fraction.py
1/2
1/2
1/2
```

A glimmer of happiness can now ensue. Now let us add fractions. Recall from Mrs. Wormwood that

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

The header of the `__add__` hook looks like this.

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

Now we do this:

- $a \leftarrow$ `self.num`
- $b \leftarrow$ `self.denom`
- $c \leftarrow$ `other.num`
- $d \leftarrow$ `other.denom`

Now we can implement the method.

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

It would be cool to be able to add an integer to a Fraction (Hey... they are both numbers), so we proceed as follows.

```
def __add__(self, other):
    if type(other) == type(1):
        other = Fraction(other)
    return Fraction(self.num*other.denom +
        self.denom*other.num, self.denom*other.denom)
```

Now take it for a spin. Add these lines

```
print(f + g)
print(f + 5)
```

Run.

```
unix> python Fraction.py
1/2
1/2
1/2
1/1
11/2
```

Doing subtraction is simple.

```
    def __sub__(self, other):
        if type(other) == type(1):
            other = Fraction(other)
        return Fraction(self.num*other.denom -
            self.denom*other.num, self.denom*other.denom)
```

Next come multiply, divide, and power.

```
    def __mul__(self, other):
        if type(other) == type(1):
            other = Fraction(other)
        return Fraction(self.num*other.num, self.denom*other.denom)
    def __truediv__(self, other):
        if type(other) == type(1):
            other = Fraction(other)
        return Fraction(self.num*other.denom, self.denom*other.num)
    def __pow__(self, n):
        return Fraction(self.num**n, self.denom**n)
```