Lecture 06: Classy class class

We've been using a lot of syntax that might seem strange. For example with matrix vector multiply:

```
A.shape,  A.dot(x), A.trace()
```

In [1]:
```
import numpy as np

A = np.array([[1,2],[3,4]])

print('A:')
print(A)
print('')

print('A.shape   =',A.shape)

print('A.trace() =', A.trace())
```

```
A:
[[1 2]
 [3 4]]

A.shape   = (2, 2)
A.trace() = 5
```

`numpy` `array` doesn't have a determinant method. The reason is that determinants can get really complicated, and are better handeled by specialised software.

In `numpy`, the determinant is attached to a linear algebra library. Not the matrix object.

In [2]:
```
np.linalg.det(A).astype(int)
```

Out[2]:
```
-2
```

In [3]:
```
x = np.array([6,7])

print('x:')
print(x)
print('')

print('x.shape   =',x.shape)
print('')

print('A.dot(x):')
print(A.dot(x))

x.shape = (2,1)

print('x:')
print(x)
print('')

print('x.shape   =',x.shape)
print('')

print('A.dot(x):')
print(A.dot(x))

x.shape = (1,2)

print('x:')
print(x)
print('')

print('x.shape   =',x.shape)
print('')

print('x.dot(A):')
print(x.dot(A))
```

```
x:
[6 7]

x.shape   = (2,)

A.dot(x):
[20 46]
x:
[[6]
 [7]]

x.shape   = (2, 1)

A.dot(x):
[[20]
 [46]]
x:
[[6 7]]

x.shape   = (1, 2)

x.dot(A):
[[27 40]]
```

Or with complex numbers

```
c.real, c.imag, c.conjugate()
```

In [4]:
```
c = 3 + 4j

print('c      =',c)
print('c.real =',c.real)
print('c.imag =',c.imag)

print('')

cc = c.conjugate()

print('cc = c.conjugate()')

print('')

print('cc      =',cc)
print('cc.real =',cc.real)
print('cc.imag =',cc.imag)

print('')

print('|c|**2  =', c*cc)
```

```
c      = (3+4j)
c.real = 3.0
c.imag = 4.0

cc = c.conjugate()

cc      = (3-4j)
cc.real = 3.0
cc.imag = -4.0

|c|**2  = (25+0j)
```

What does the `'.'` mean in Python? How come some names that come after a dot have brackets `()` and others don't?

We would also like to make better plots with `matplotlib`. Right now (for example) you can use `plt.plot(x,y,color='red')` to make a red line plot for arrays `x` versus `y`. But what if you want to label the axes? You might try to pass in a keyword the way you pass in the color. But this won't work. If you look on the internet about how to do this, you will see the examples looking a lot more complicated than what you've been doing so far.

There is a reason of this. Plotting in `matplotlib` is ***Object oriented***.

We've already done a lot of **Procedural** and **Functional** styles. To understand more about how `matplotlib` works, it's useful to start to understand object oriented style. This will pay off big time. We'll see in this lecture that we can do some powerful things.

Object oriented programming often seems strange at first. But it's not so bad if know what we do about Python. Remember Python works on a *binded environment* system. The object oriented approach creates seperate environments that can act on their own and with each other in controlled ways.

The idea of a ***Class*** is central to object oriented programming. What is a `class`? It turns out everything we've been doing so far is ultimately a class underneath everything; including functions. A class is the prototype for a mini environment that we can include into out larger environment.

A **class** is a *template*. It lays out the rules for little environments that we will create later (maybe lots of them). The little environs are called *instances* of the class. The environments can have data, and special functions that can act on that data. The class defines what the data we can use and what the functions do.

An **object** is a particular instance of a class. This is usually something with a name, eg, `A`, `x`, `c`, `cc`. If the class is called `dog`, the data in the class might be what kind of dog, how old is the dog, etc. The `object` is a particular dog with particular characteristics.

```
>scooby = dog(name='Scooby-Doo', breed='cartoon', weight=None, food='Scooby snack
s')
```

A **method** is a function that can act on an object. It's a function that is attached to (associated with, aka binded to) the object (which is an enviroment). This means there is a special way of calling it.

```
>scooby.speak('hello world!')
're-row rorld!'
```

The function can also take other arguments.

A **attribute** is a little bit of data that is attached to the object.

```
>print(scooby.age)
'49 in people years'
```

Complex numbers are probably a better example of a class. Every complex number *object* contains a pair of real numbers; the real and imaginary part. Asking for those real and imag part uses the `.` operator. You can also add, subtract, multiply and divide complex numbers; and more. These are all methods that know how to operate on complex objects.

We will have a closer look at a particular class. We want one that is just interesting enough to show the power of classes. But hopefully you know enough about how rational numbers work to know what we need.

EXAMPLE: RATIONAL NUMBERS

Rational numbers are pairs of integers

$$\frac{n}{d} \longleftrightarrow (n, d)$$

where n = numerator, d = denominator.

But we rememebr from the difficulty of learning to add fractions, that this pair of integers doesn't satisfy a simple additions rule. Rather

$$\frac{a}{b} + \frac{c}{d} = \frac{a\,d + b\,c}{b\,d}$$

Or

$$(a, b) + (c, d) = (a\,d + b\,c, b\,d)$$

Subtracting them is similar

$$(a, b) - (c, d) = (a\,d - b\,c, b\,d)$$

Multiplication is simpler

$$(a, b) \times (c, d) = (a\,c, b\,d).$$

We can also divide rational numbers

$$(a, b) \div (c, d) = (a\,d, b\,c).$$

We also know that rational number are not unique. For any integer, n

$$(n\,a, n\,b) = (a, b)$$

This means we can *reduce* a rational number by dividing out the *greatest common divisor* of a and b; $n = gcd(a, b)$

Euclid devloped the algorithm for finding greatest common divisor of two integers. It is a important very algorithm even today. Euclid first published the method in his $Elements$ did this approximately 2,300 years ago!

And in a very interesting twist, the scheme is recursive. You can read a lot of good info about it on wikipedia:

https://en.wikipedia.org/wiki/Euclidean_algorithm (https://en.wikipedia.org/wiki/Euclidean_algorithm)

It may or may not be a surpirse, but the algorithm is how you compute the continued fraction representation of numbers.

Here is a first attempt at creating a `rational class`:

In [5]:
```
class rational():

    def __init__(self,num=0,den=1):
        self.num     = int(num)
        self.den     = int(den)
        self.decimal = num/den
```

This is the way you tell a `class` what it's going to be. The `__init__` is a very basic function that tells a class how to initialise itself. The double underscore is reserved for special words. We'll talk more about this soon.

The `__init__`

The `__init__` function is here you put all the initial `attributes` associated with the class.

The `self`

To reiterate an important point about classes: Every single different instance of a class is a new *enviroment* setting in the main python enviroment. This means that ever single thing in the new (sub) enviroment is *attached* aka *binded* to the name of the enviroment. In the definition of the class, we don't know what the specific name is going to be yet. So we give it the name `self` for now. The first arguement of every function in a class is `self`.

And since we pass `self` to every function in a class, we have all of it's attributes that we can use anytime.

This probably seems weird at this point. Let's try some examples and see how it goes.

Here's how you ***instantiate*** a ***instance*** of the `calls rational`.

In [6]:
```
r = rational(3,2)

print(r)
```

```
<__main__.rational object at 0x7f18dd6f06a0>
```

An ***instance*** is a particular member of the class. The varible `r` is now the `object`. It is its own little enviroment. In this case the rational number (3,2). The particual member is distinguished by it's ***attributes***, in this case `num`, and `den`.

In [7]:
```
print(r.num)
print(r.den)
print(r.decimal)
```

```
3
2
1.5
```

At this point, the `rational` number `r` only has it numerator and denominator, and decimal. It doesn't have anything else that comes with rationals. Like addition, multiplication, etc.

Let's define some more.

Remember, a rational number is the same if you multiply its it numerator and denominator by the same integer. We want to be able to `reduce` a rational to the lowest common denominator. And we also want to have a `attribute` that tells us it it is reduced.

Reducing: gcd

In [8]:
```
class rational():

    def __init__(self,num,den=1.0):
        self.num     = int(num)
        self.den     = int(den)
        self.decimal = num/den
        self.reduced = self.is_reduced()
        self.decimal = num/den

    # Euclid's algorithm
    def gcd(self):
        n, d = self.num, self.den
        while (d > 0): n, d = d, n%d
        return n

    def is_reduced(self): return self.gcd() == 1

    def reduce(self):
        n = self.gcd()
        self.num, self.den = self.num//n, self.den//n
        self.reduced = True
```

Now we can much more easliy ***instantiate*** two `rational` objects

In [9]:
```
r = rational(3,2)
s = rational(35,91)
t = rational(51,34)
```

Instantiate means "make an instance".

`r` and `s` have ***attributes***

In [10]:
```
print('r.num     =', r.num)
print('r.den     =', r.den)
print('r.decimal =', r.decimal)
print('r.reduced =', r.reduced)
print('------------')
print('s.num     =', s.num)
print('s.den     =', s.den)
print('s.decimal =', s.decimal)
print('s.reduced =', s.reduced)
print('------------')
print('t.num     =', t.num)
print('t.den     =', t.den)
print('t.decimal =', t.decimal)
print('t.reduced =', t.reduced)
```

```
r.num     = 3
r.den     = 2
r.decimal = 1.5
r.reduced = True
------------
s.num     = 35
s.den     = 91
s.decimal = 0.38461538461538464
s.reduced = False
------------
t.num     = 51
t.den     = 34
t.decimal = 1.5
t.reduced = False
```

`r`, `s` and `t` have the ***method*** `gcd()`. Remember, these are functions attached to the instance of the class `rational`; aka objects. The objects are defalut arguements of the function. Because the function is attached to the objects. In this case, the function `gcd()` takes no other arguements. It could in general.

In [11]:
```
print('r.gcd() = ', r.gcd())
print('s.gcd() = ', s.gcd())
print('t.gcd() = ', t.gcd())
```

```
r.gcd() =  1
s.gcd() =  7
t.gcd() =  17
```

We can see that $(3, 2)$ are relatively prime, $(35, 91)$ have a $gcd = 7$; and $(51, 34)$ have $gcd = 17$.

The method `gcd()` doesn't change `r` and/or `s` in any way. It only reports what it sees. This is a `functional` method within a class. It leaves no side effects.

We can also have methods that change the object itself. These are "side effects". They are not always bad. In this case it's just what we want.

In [12]:
```
print('s.num, s.den, s.decimal = ',s.num, s.den, s.decimal)
print('t.num, t.den, t.decimal = ',t.num, t.den, t.decimal)
```

```
s.num, s.den, s.decimal =  35 91 0.38461538461538464
t.num, t.den, t.decimal =  51 34 1.5
```

Here is another function. This will have side effects. In this case, the only effect is a side effect.

In [13]:
```
s.reduce()
t.reduce()
```

The function didn't `return` anything. It just altered the numerator and denominator. This didn't change the actual *value* of `s`. It only *reduced* the fraction. The `.reduce()` method is all side effects.

In [14]:
```
print('s.num, s.den, s.decimal = ',s.num, s.den, s.decimal)
print('s.reduced =',s.reduced)
print(5/13==35/91)
print('')
print('t.num, t.den, t.decimal = ',t.num, t.den, t.decimal)
print('t.reduced =',t.reduced)
print(51/34==3/2)
```

```
s.num, s.den, s.decimal =  5 13 0.38461538461538464
s.reduced = True
True

t.num, t.den, t.decimal =  3 2 1.5
t.reduced = True
True
```

Adding

Defining and reducing rational numbers is fun. But we really want to ba able to *compute* with them. For this we need to tell the computer how to do this. Right now, it doesn't know.

In [15]:
```
r+s
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-a956ec62072b> in <module>()
----> 1 r+s

TypeError: unsupported operand type(s) for +: 'rational' and 'rational'
```

This makes an add method.

In [ ]:
```
class rational():

    def __init__(self,num=0,den=1):
        self.num     = int(num)
        self.den     = int(den)
        self.reduced = self.is_reduced()
        self.decimal = num/den

    def gcd(self):
        n, d = self.num, self.den
        while (d > 0): n, d = d, n%d
        return n

    def is_reduced(self): return self.gcd() == 1

    def reduce(self):
        n = self.gcd()
        self.num, self.den = self.num//n, self.den//n
        self.reduced = True

    # Let's make a better way to see the numbers.
    def string(self): return ('%i/%i' % (self.num, self.den))

    # This is the new adding function
    # It takes an optional arguement, 'reduce'
    # The defalut vaule is 'True'.
    # It's will assume this if you don't say anything about it.
    def add(self,other,reduce=True):
        a, b = self.num,  self.den
        c, d = other.num, other.den
        q = rational(a*d+b*c,b*d)
        if reduce: q.reduce()
        return q
```

Hopefully the syntax is clear. The first two lines get the numerator and denominator of the numbers to be added.

One of the numbers is called "`self`" and the other is called "`other`". The reason for this naming convention is that methods are attached to objects. That means that the function `add` (in this case) is attached to the first rational number in the additions. The argument of this function also requires a partner.

$r + s$

In [ ]:
```
r = rational(3,2)
s = rational(35,91)

# The function `add` is attacted the the `r` enviroment.
# It takes an arguement, in this case, `s`.

q = r.add(s)

print('q         =', q.string())
print('q.decimal =', q.decimal)

print(r.num/r.den + s.num/s.den == q.num/q.den)
```

We could do it the other way around: $s + r$

In [ ]:
```
q = s.add(r)

print('q         =', q.string())
print('q.decimal =', q.decimal)

print(r.num/r.den + s.num/s.den == q.num/q.den)
```

What about adding rationals to integers?

In [ ]:
```
q = r.add(1)
```

The enviroment bouncer didn't like this becuase `intergers` don't have the same attributes as `rational`, and `rational` only knows how to add things of its own kind.

To fix this, we can put a catch on the input of `add` and *convert* `integers` to `rationals` if they come in:

In [ ]:
```
class rational():

    def __init__(self,num,den=1.0):
        self.num     = int(num)
        self.den     = int(den)
        self.reduced = self.is_reduced()
        self.decimal = num/den

    def gcd(self):
        n, d = self.num, self.den
        while (d > 0): n, d = d, n%d
        return n

    def is_reduced(self): return self.gcd() == 1

    def reduce(self):
        n = self.gcd()
        self.num, self.den = self.num//n, self.den//n
        self.reduced = True

    def string(self): return ('%i/%i' % (self.num, self.den))

    # This converts other to a rational if other is an integer, and then
 adds them.
    def add(self,other,reduce=True):
        if isinstance(other,int): other = rational(other) # This is the n
ew line.
        a, b = self.num,  self.den
        c, d = other.num, other.den
        q = rational(a*d+b*c,b*d)
        if reduce: q.reduce()
        return q
```

Now it should work:

In [ ]:
```
r = rational(3,2)
s = rational(35,91)

q = r.add(1)

print('r         =', r.string())
print('q         =', q.string())
print('r.decimal =', r.decimal)
print('q.decimal =', q.decimal)
```

It doesn't work the other way around becuase methods are attached to `rational` objects

In [ ]:
```
q = (1).add(r)
```

Of course not, Python does know how to add integers and rational numbers! We just made up the latter.

```
Remember: the method is always attached to the thing on the left of the ".";
          that is an integer in this case.
```

So there much be a way to make the interpreter realise what's going on. The answer is very similar to what we saw with `init` versus `__init__`.

We'll fix the issue about adding with `int` later. For now, let's see what the errors can tell us about how things work.

Integers do have an add attribute. But it's *hidden*. The way to hide it is with `__add__`. Now watch the ***magic***:

In [ ]:
```
class rational():

    def __init__(self,num=0,den=1):
        self.num     = int(num)
        self.den     = int(den)
        self.reduced = self.is_reduced()
        self.decimal = num/den

    def gcd(self):
        n, d = self.num, self.den
        while (d > 0): n, d = d, n%d
        return n

    def is_reduced(self): return self.gcd() == 1

    def reduce(self):
        n = self.gcd()
        self.num, self.den = self.num//n, self.den//n
        self.reduced = True

    # I changed this from `string` to `__str__`
    def __str__(self): return ('%i/%i' % (self.num, self.den))

    # I just changed `add` to `__add__`
    def __add__(self,other,reduce=True):
        if isinstance(other,int): other = rational(other)
        a, b = self.num,  self.den
        c, d = other.num, other.den
        q = rational(a*d+b*c,b*d)
        if reduce: q.reduce()
        return q
```

Both "`__add__`" and "`__str__`" are things Python knows to expect.

In [ ]:
```
r = rational(3,2)
s = rational(35,91)
```

It doesn't seem to know `add` anymore.

In [ ]:
```
q = r.add(s)
```

Of course not! It's not called "`add`" anymore. But what about:

In [ ]:
```
q = r.__add__(s)

print('  r = %+5s = %f' %(r.__str__(),r.decimal))
print('  s = %+5s = %f' %(s.__str__(),s.decimal))

print('+ --------------------')

print('  q = %+5s = %f' %(q.__str__(),q.decimal))
```

It seesm like we've made it look ugly with nothing gained. But integers *do* know about `__add__`, and `__str__`

In [ ]:
```
n = (1).__add__(2)
print(n.__str__())
```

But we've never had to use `__add__` before with `integers`. Why not just try:

In [ ]:
```
n = 1+2
print(n)
```

The big news: magic methods

With integers, `a.__add__(b)` gave the same answer as `a + b`.

And `print(q.__str__())` gave the same thing as `print(q)`

Why?

Becuase every time Python sees

```
a + b, or print(q)
```

It replaces it with

```
a.__add__(b), or print(q.__str__())
```

Everytime. It doesn't care what `a`, `b` and `q` are. It just does a string replacement. It tries to evaluate after the replacement

If the `object` `a` knows how to `__add__` and make a `__str__` (because you or someone else told it how) then it will work.

We can *reuse* aka ***overload*** the `__add__` and `__str__` methods, ***becuse methods are binded to their objects***.

And Python makes it nice for everyone make their code act like math.

You can read all about it here:

https://docs.python.org/3/reference/datamodel.html (https://docs.python.org/3/reference/datamodel.html)

We've made print know how to work with `rational` numbers.

In [ ]:
```
print(r)
```

It turns out the Python *ever* only knew how to `print` `string` data. Every time you see a `print` statement of something that is not a string (eg, `float`, `int`, `complex`), it happens that the thing (eg, `float`, `int`, `complex`) knows how to convert itself into a `string` and `print` that instead.

```
print(thing) is replaced with print(thing.__str__())
```

Printing also works if it's formatted nicely.

We've made rational numbers that know how to +

In [ ]:
```
q = r + s

print('  r = %+5s = %f' %( r, r.decimal ) )
print('  s = %+5s = %f' %( s, s.decimal ) )

print('+ --------------------')

print('  q = %+5s = %f' %( q, q.decimal ) )
```

Python replaced `r + s` with `r.__add__(s)`, and it worked because we defined what `__add__` does within `rational`

Always read the error message. It's saying that the class `int` doesn't know how to + rationals.

Well, nice try.

Hopefully we are starting to realise the everything is Python is a method attached to an object. The number `1` doesn't know how to convert itself to rationals.

But rationals do know how to convert integers

In [ ]:
```
print('r     =', r)

q = r + 3


print('q     =', q)
print('q.num =',q.num)
print('q.den =',q.den)
print('(q == r + 3) = ',q.decimal == r.decimal + 3)
```

One last specific example: `__repr__`

Remember from before when we typed just the name of an object we got some junk about the fact that it was an object?

In [ ]:
```
r
```

Wouldn't it be nice if this did the same thing as typing an integer?

In [ ]:
```
n = 42
```

In [ ]:
```
n
```

It can with the `__repr__` method. Which does what ever you want when you just enter the object with nothing else.

Let's make it just turn the retult into a string:

```
def __repr__(self): return str(self)
```

In [ ]:
```
class rational():

    def __init__(self,num,den=1):
        self.num     = int(num)
        self.den     = int(den)
        self.reduced = self.is_reduced()
        self.decimal = num/den

    def gcd(self):
        n, d = self.num, self.den
        while (d > 0): n, d = d, n%d
        return n

    def is_reduced(self): return self.gcd() == 1

    def reduce(self):
        n = self.gcd()
        self.num, self.den = self.num//n, self.den//n
        self.reduced = True

    def __str__(self): return ('%i/%i' % (self.num, self.den))

    def __repr__(self): return str(self)

    def __float__(self): return self.num/self.den

    def __int__(self): return self.num//self.den

    def __add__(self,other):
        if isinstance(other,int): other = rational(other)
        a, b = self.num,  self.den
        c, d = other.num, other.den
        q = rational(a*d+b*c,b*d)
        q.reduce()
        return q
```

In [ ]:
```
r = rational(13,2)
```

In [ ]:
```
r
```

I also did this:

In [ ]:
```
float(r)
```

and this:

In [ ]:
```
int(r)
```

WYSIWYG

Now the sky's the limit. We can build out a lot of good features that we want `rational` numbers to have.

In [ ]:
```
class rational():

    def __init__(self,num=0,den=1):
        self.num     = int(num)
        self.den     = int(den)
        self.reduced = self.is_reduced()
        self.decimal = num/den

    def gcd(self):
        n, d = self.num, self.den
        while (d > 0): n, d = d, n%d
        return n

    def continued_fraction(self):
        n, d = self.num, self.den
        L = []
        while (d>0):
            L += [n//d]
            n , d = d, n%d
        return L

    def is_reduced(self): return self.gcd() == 1

    def reduce(self):
        n = self.gcd()
        self.num, self.den = self.num//n, self.den//n
        self.reduced = True

    def __str__(self):
        if self.den == 1: return str(self.num)
        return '%i/%i' % (self.num, self.den)

    def __repr__(self): return str(self)

    def __float__(self): return self.num/self.den

    def __int__(self): return self.num//self.den

    def __add__(self,other):
        if isinstance(other,int): other = rational(other)
        a, b = self.num,  self.den
```

```
        c, d = other.num, other.den
        q = rational(a*d+b*c,b*d)
        q.reduce()
        return q

    def __neg__(self): return rational(-self.num,self.den)

    def __abs__(self): return rational(abs(self.num),abs(self.den))

    def __sub__(self,other): return self + (-other)

    def __mul__(self,other):
        if isinstance(other,int): other = rational(other)
        a, b = self.num,  self.den
        c, d = other.num, other.den
        q = rational(a*c,b*d)
        q.reduce()
        return q

    def __invert__(self): return rational(self.den,self.num)

    def __truediv__(self,other):
        if isinstance(other,int): other = rational(other)
        return self*(~other)

    def __pow__(self,n):
        if n == 0: return rational(1,1)
        if n  > 0: return self*(self**(n-1))
        return self*((~self)**(1-n))

    def __eq__(self,other):
        if isinstance(other,int): other = rational(other)
        return self.num*other.den == self.den*other.num

    def __ne__(self,other):
        return not (self == other)

    def __lt__(self,other):
        if isinstance(other,int): other = rational(other)
        self.reduce()
        other.reduce()
        return self.num*other.den < self.den*other.num

    def __le__(self, other):
        if isinstance(other,int): other = rational(other)
        self.reduce()
        other.reduce()
        return self.num*other.den <= self.den*other.num

    def __gt__(self,other):
        if isinstance(other,int): other = rational(other)
        self.reduce()
        other.reduce()
        return self.num*other.den > self.den*other.num

    def __ge__(self,other):
        if isinstance(other,int): other = rational(other)
```

```
        self.reduce()
        other.reduce()
        return self.num*other.den >= self.den*other.num
```

Notice how most of these methods are of the form: `__method__` ? This means you can google the and find out what they represent. In every case, something standard you type will be replaces with one of these functions if the object is a `rational`.

Anything without the __ is not standard and is something unique to `rational`. How many there are should give an idea about how much stuff shares the same properties with `rational` numbers.

You can also see that as soon as something like '`__add__`' is defined, I can switch to using '+' in its place.

This class has some fun stuff in it. For example:

In [ ]:
```
r = rational(3,2)
s = rational(35,91)
print('r   =',r)
print('s   =',s)
s.reduce()
print('s   =',s)
print('r+s =',r+s)
print('r-s =',r-s)
print('r*s =',r*s)
print('r/s =',r/s)
```

In [ ]:
```
r = rational(2)
print(r)
print(r**(-10))
-r/10
```

In [ ]:
```
print('r**2  =',r**2)
print('r**-2 =',r**-2)

print('')

q = (r**3 + (r**(-17))*3)/2
print('q = (r**3 + ( r**(-17) )*3)/2')

print('q = %s = %f' %(q,q.decimal))
```

We can use Euclid's algorythim to compute the continued fraction sequence of a number. You'll have to trust me on why this works for now.

In [ ]:
```
print(q.continued_fraction())
```

In [ ]:
```
one = rational(1)

a = rational(4)
b = rational(87381)
c = rational(3)

print(q == a + one/(b + one/ c))
```

Recursive functions with `rational`

We can even define recursive continued fraction functions as before. But this time it's in *exact* rational arithmetic.

Recall:

$$r_{n+1} = \frac{r_n + 2}{r_n + 1} \quad \Longrightarrow \quad \lim_{n\to\infty} r_n \ = \ \sqrt{2}$$

$$r_{n+1} = \frac{r_n + 1}{r_n} \quad \Longrightarrow \quad \lim_{n\to\infty} r_n \ = \ \frac{1+\sqrt{5}}{2} \ = \ \phi$$

In [ ]:
```
def root_two(k):
    if k == 0: return rational(1)
    d = root_two(k-1) + 1
    return (d+1)/d

def phi(k):
    if k == 0: return rational(1)
    d = phi(k-1)
    return (d+1)/d
```

First look at $\sqrt{2}$:

In [ ]:
```
x = root_two(10)
print(x)
```

We can see that we are doing well:

In [ ]:
```
print('r(10)     = ',x.decimal)
print('r(10)**2  = ',x.decimal**2)
```

Now with exact arithmetic we can really see how well:

In [ ]:
```
e = abs(x**2 - 2)
print('x**2 - 2 = ',e)
```

In [ ]:
```
for k in range(42):
    x = root_two(k)
    e = abs(x**2 - 2)
    print("error(%02i): %e = %s" %(k,e.decimal,e))
```

In [ ]:
```
x = phi(3)
e = abs((x+x - 1)**2 - 5 )
print(x)
print(e)
```

In [ ]:
```
for k in range(42):
    x = phi(k)
    e = abs((x+x - 1)**2 - 5 )
    print("error(%02i): %e = %s" %(k,e.decimal,e))
```

We can see clearly that the sequence for ϕ is converging slower than the sequence for $\sqrt{2}$.

The error after 41 iterations is $\approx\ 5\times 10^{-17}$, versus $\approx\ 5\times 10^{-32}$.

Remember that I said Newton's method and Halley's method converged faster that the original continued fraction sequence?

Recall Newton's and Halley's methods work by solving an equation of the form
$$f(x) = 0$$
This

Newton works by iterating the function
$$F(x) \ = \ x \ - \ \frac{f(x)}{f'(x)} \quad \longrightarrow \quad x_{n+1} \ = \ F(x_n)$$

Halley works by iterating the function
$$H(x) \ = \ x \ + \ \frac{2f(x)f'(x)}{f(x)f''(x) - 2f'(x)^2} \quad \longrightarrow \quad x_{n+1} \ = \ H(x_n)$$

Computing ϕ

In the case of $\phi = (1+\sqrt{5})/2$,
$$f(x) = x^2 - x - 1 \quad \Longrightarrow \quad F(x) \ = \ \frac{x^2+1}{2x-1}, \quad \text{and} \quad H(x) \ = \ \frac{x^3+3x-1}{3x^2-3x+2}$$

In all case $\lim_{n\to\infty} x_n = \phi$. But different methods will go faster.

We can now see what happens in full detail.

In [ ]:
```
def phi(k):
    if k == 0: return rational(1)
    d = phi(k-1)
    return (d+1)/d

def Newton_phi(k):
    if k == 0: return rational(1)
    x = Newton_phi(k-1)
    n = x**2 + 1
    d = x*2  - 1
    return n/d

def Halley_phi(k):
    if k == 0: return rational(1)
    x = Halley_phi(k-1)
    n =  x**3 + x*3 - 1
    d = (x**2)*3 - (x*3) + 2
    return n/d
```

Newton_phi(k) = phi(2**k-1) ?

In [ ]:
```
for k in range(10):
    print(Newton_phi(k)==phi(2**k-1))
```

Halley_phi(k) = phi(3**k-1) ?

In [ ]:
```
for k in range(7):
    print(Halley_phi(k)==phi(3**k-1))
```

We can look at the error. But remember, we can't compare how well the iteration does compared to the "exact" answer. Because the "exact" answer is a fiction.

We can do one of two things:

```
1) Compute a very high number of iterations and compare how well small numbers of i
terations do in comparrison.

2) See how well the approximate solution approximately solves the original equatio
n.
```

The problem with option `1` is: how many iterations is enough to compare against? This is the idea of a Cauchy sequence. It's a good method, but has a few too many moving parts.

We'll do option `2`.

So we can check the size of

$$(2\,x_n - 1)^2 - 5 \ = \ ?$$

This should vanhish is $x_n = \phi$, which it never will.

Let's see:

In [ ]:
```
x = Halley_phi(6)
print('x =',x)

print('')

e = abs( (x*2 - 1)**2 - 5 )
print("%e = %s" %(e.decimal,e))
```

6 iterations with Halley's method gives ϕ to more than 300 digits!

Also remember how I said the continued fraction sequence for ϕ was all 1 ?

We can check that:

In [ ]:
```
x=phi(20)

y = rational(1)/(x-1)

print("x = %-11s = %f" %(x,x.decimal))
print("y = %-11s = %f" %(y,y.decimal))

print(x.continued_fraction())
print(y.continued_fraction())

print(y==phi(19))
```

Remember what I said about the Fibinacci numbers:

In [ ]:
```
for i in range(11):
    print(phi(i))
```

And for $\sqrt{2}$?

In [ ]:
```
x=root_two(20)

y = rational(1)/(x-1) - 1

print("x = %-11s = %f" %(x,x.decimal))
print("y = %-11s = %f" %(y,y.decimal))

print(x.continued_fraction())
print(y.continued_fraction())

print(y==root_two(19))
```

This gives the sequence from the tutorial

In [ ]:
```
for i in range(11):
    print(root_two(i))
```

`right` operations

You might have noticed that I was careful to only add integers to rational numbers when the integer is on the *right*.

As we've seen many times before, this is becuase the `methods` are binded to the object on the *left*.

Integers don't know how to add themselves to rationals.

In [ ]:
```
r = rational(7,5)
print(r)
```

This works just fine (as we've seen)

In [ ]:
```
s = r + 3
print(s)
```

In [ ]:
```
t = 3 + r
```

This failed for exactly the reason that Python reported. How do we fix it?

We can't (and don't want to) get into the `integer` class. What is Python trying to do.

```
1) replace 3 + r --> 3.__add__(r)
2) call int._add__(self,other) where self = 3, and other = r
3) fail because the add in the integer class knows nothing about our rational clas
s.
```

But every time you think you at a dead end, don't worry. The Python inventors probably thought about the same thing. Definitely in this case. It is natural to want to add new things to intergers.

This is what "right add" is for:

```
def __radd__(self, other): return self + other
```

Now the sequence above goes this way:

```
1) replace 3 + r --> 3.__add__(r)
2) call int._add__(self,other) where self = 3, and other = r
3) fail because the add in the integer class knows nothing about our rational clas
s.

4) check to see if other has a "__radd__" method.
5) if _radd__ does exist, then use that instead.
```

In [ ]:
```
class rational():
    """A class for arithmetic with rational numbers.

       Parameters
       ----------
       num, den: int


       Attributes
       ----------
       reduced: True/False
       decimal: float


       Methods
```

```
    -------
    gcd()               : returns integer
    continued_fraction(): returns list of integers
    is_reduced()        : returns True/False
    reduce()            : reduces by gcd()

    All other methods are standard magic methods:
        int, float, abs, max, min
        + , - , * , ~ , /, //, %,  ** , == , != , > , >= , < , <=

    """

def __init__(self,num=0,den=1):
    self.num     = int(num)
    self.den     = int(den)
    self.reduced = self.is_reduced()
    self.decimal = float(self)

def gcd(self):
    n, d = self.num, self.den
    while (d > 0): n, d = d, n%d
    return n

def continued_fraction(self):
    n, d = self.num, self.den
    L = []
    while (d>0):
        L += [n//d]
        n , d = d, n%d
    return L

def is_reduced(self): return self.gcd() == 1

def reduce(self):
    n = self.gcd()
    self.num, self.den = self.num//n, self.den//n
    self.reduced = True

def __str__(self):
    if self.den == 1: return str(self.num)
    return '%i/%i' % (self.num, self.den)

def __repr__(self): return str(self)

def __float__(self): return self.num/self.den

def __int__(self): return self.num//self.den

def __add__(self,other):
    if isinstance(other,float): return float(self) + other
    if isinstance(other,int)  : other = rational(other)
    a, b = self.num,  self.den
    c, d = other.num, other.den
    q = rational(a*d+b*c,b*d)
    q.reduce()
    return q
```

```
    def __neg__(self): return rational(-self.num,self.den)

    def __sub__(self,other): return self + (-other)

    def __pos__(self): return self

    def __abs__(self): return rational(abs(self.num),abs(self.den))

    def __max__(self,other):
        if self > other: return self
        return other

    def __max__(self,other):
        if self < other: return self
        return other

    def __mul__(self,other):
        if isinstance(other,int)  : other = rational(other)
        if isinstance(other,float): return float(self) * other
        a, b = self.num,  self.den
        c, d = other.num, other.den
        q = rational(a*c,b*d)
        q.reduce()
        return q

    def __invert__(self): return rational(self.den,self.num)

    def __truediv__(self,other):
        if isinstance(other,float): return float(self)/other
        if isinstance(other,int)  : other = rational(other)
        return self*(~other)

    def __floordiv__(self,other): return int(self/other)

    def __mod__(self,other): return self - (self//other)*other

    def __pow__(self,n):
        if not isinstance(n,int): raise ValueError('Undefined: exponent not in
t.')
        if n == 0: return rational(1)
        if n  > 0: return self*(self**(n-1))
        return self*((~self)**(1-n))

    def __eq__(self,other):
        if isinstance(other,int): other = rational(other)
        return self.num*other.den == self.den*other.num

    def __ne__(self,other):
        return not (self == other)

    def __lt__(self,other):
        if isinstance(other,int): other = rational(other)
        self.reduce()
        other.reduce()
        return self.num*other.den < self.den*other.num

    def __le__(self, other):
```

```
        if isinstance(other,int): other = rational(other)
        self.reduce()
        other.reduce()
        return self.num*other.den <= self.den*other.num

    def __gt__(self,other):
        if isinstance(other,int): other = rational(other)
        self.reduce()
        other.reduce()
        return self.num*other.den > self.den*other.num

    def __ge__(self,other):
        if isinstance(other,int): other = rational(other)
        self.reduce()
        other.reduce()
        return self.num*other.den >= self.den*other.num

    def __radd__(self, other): return self + other

    def __rsub__(self,other): return -self + other

    def __rmax__(self,other): return max(self,other)

    def __rmin__(self,other): return min(self,other)

    def __rmul__(self, other): return self * other

    def __rtruediv__(self,other):
        if isinstance(other,float): return other/float(self)
        return ~(self/other)

    def __rfloordiv__(self,other):
        if isinstance(other,float): return other//float(self)
        return int(~(self/other))

    def __rmod__(self,other): return - self*(other//self) + other

    def __req__(self,other): return self == other

    def __rne__(self,other): return self != other

    def __rgt__(self,other): return self <= other

    def __rlt__(self,other): return self >= other

    def __gle__(self,other): return self <  other

    def __rle__(self,other): return self >  other
```

Notice how we have to rearrange things if the operations don't commute:

```
other.__operator__(self) != self.__operator__(other)

self always needs to go on the left.
```

But if you get it right, then everything works greats

In [ ]:
```
r = rational(5,2)
s = rational(17,3)
```

In [ ]:
```
2 + r
```

In [ ]:
```
max(r,s)
```

In [ ]:
```
3 % r
```

In [ ]:
```
3 - r
```

In [ ]:
```
3*r
```

In [ ]:
```
13/r
```

In [ ]:
```
1.3 + r
```

In [ ]:
```
3.3*r
```

In [ ]:
```
3.2/r
```

In [ ]:
```
4*r//5
```

We can even use `rational` with `np.array` out of the box.

In [ ]:
```
import numpy as np

a = rational(3,2)
b = rational(5,7)
c = rational(1,5)
d = rational(4,5)

A = np.array([[a,b],[c,d]])


x = np.array([rational(1,3),rational(5,13)])

y = A.dot(x)
```

In [ ]:
```
print(A)
print('-----------')
print(x)
print('-----------')
print(y)
```

In [ ]:
```
print(x+y)
print(x*y)
```

Almost "right out of the box".

In [ ]:
```
a*A
```

We havn't told anyone how to multiply `rational` and `array`

But we can do this:

In [ ]:
```
a.num*A/a.den
```

This works becuase `numpy array` knows how to multiply and divide with integers. Basically, it knows how to pass the operations through the brackets to the contents of the matrix. Once the integers get there, they know how to cooperate with `rational` and everything works after that.

And we could teach everyone to behave more easily if we want. But this is pretty good for now.

The "Amazing matrix":

See: https://arxiv.org/pdf/0806.3583.pdf (https://arxiv.org/pdf/0806.3583.pdf)

In [ ]:
```
def Amazing(b):

    if isinstance(b,int): b = rational(b)

    r = 1/(6*b**2)
    n = r.num
    d = r.den
    P = [[ b**2 + 3*b + 2, 4*b**2 - 4, b**2 - 3*b + 2],
         [ b**2       - 1, 4*b**2 + 2, b**2       - 1],
         [ b**2 - 3*b + 2, 4*b**2 - 4, b**2 + 3*b + 2]]
    return n*np.array(P)/d
```

In [ ]:
```
a = rational(3,17)
b = rational(5,2)
c = a*b

Pa = Amazing(a)
Pb = Amazing(b)
Pc = Amazing(c)

print("Pa:")
print(Pa)

print('')
print("Pb:")
print(Pb)

print('')
print("Pc:")
print(Pc)
```

In [ ]:
```
Pa.dot(Pb)
```

There are many properties that make this matrix "Amazing":

In [ ]:
```
v = np.array([[1,4,1]])

print(v.dot(Pa))
print(v.dot(Pb))
print(v.dot(Pc))
```

In [ ]:
```
u = np.array([1,1,1])

print(Pa.dot(u))
print(Pb.dot(u))
print(Pc.dot(u))
```

There's always more to do:

At this point, it would be possible to tweak the `rational` class a little to include some other nice things (eg `%`, and `+=`). But the point is that you should have a very good understanding about how the basic Python framework functions.

You can read more about some of the other things you could include at:

https://docs.python.org/2/library/operator.html (https://docs.python.org/2/library/operator.html)

Next time, we'll see how `matplotlib` does plotting with objects.