Goals

- What are **class variables**

Class Variables

- Variables shared amongst all instances of a class
- Variable is the same for each instance of the class

The Beta Method

Let's add a new method to our class which calculates the quantity β, which tells us how close to the speed of light a particle is traveling. It is defined as the particles speed, v, over the speed of light, c:

$\beta = \frac{v}{c}$

In this case it would be nice to have the same value of c shared across all particle. We can impliment the method ***beta*** and see how to use this without setting a class variable first

In [1]:
```
class Particle:
    def __init__(self, name, mass, mass_unit, charge, vel):
        self.name = name
        self.mass = mass
        self.mass_unit = mass_unit
        self.vel = vel #added velocity attribute
        self.mass_list = '{} {}'.format(mass, mass_unit)

    def mass_square(self):
        return '{} {}^2'.format(self.mass**2, self.mass_unit)

    def get_beta(self):
        return self.vel/3.0e8

par_2 = Particle('Proton', 0.938, 'GeV', 1,1.2e6)
print(par_2.get_beta())
```

```
0.004
```

The above works fine, however there are a couple things that can be done better.

1) What is we wanted to see the value used to the speed of light?

- It would be nice have a method return that value.

2) What if we wanted to update our speed of light value?

- If we wanted to make the value more percise
- As our code get more complex (and longer) who knows where in the code we use the speed of light. Going though 1000's of line of code and changing hard coded values is a pain and prone to errors!

We can do better by implimenting the speed of light as a class variable

In [2]:
```
class Particle:
    #define class variables at top of the class
    c = 3.0e8 #speed of light

    def __init__(self, name, mass, mass_unit, charge, vel):
        self.name = name
        self.mass = mass
        self.mass_unit = mass_unit
        self.vel = vel #added velocity attribute
        self.mass_list = '{} {}'.format(mass, mass_unit)

    def mass_square(self):
        return '{} {}^2'.format(self.mass**2, self.mass_unit)

    def get_beta(self):
        return self.vel/c # try just using c. What happens? A 'name' err
or!

par_2 = Particle('Proton', 0.938, 'GeV', 1,1.2e6)
print(par_2.get_beta())
```

```
---------------------------------------------------------------------------
----
NameError                                 Traceback (most recent call l
ast)
<ipython-input-2-daf98e07ecdb> in <module>
     17
     18 par_2 = Particle('Proton', 0.938, 'GeV', 1,1.2e6)
---> 19 print(par_2.get_beta())

<ipython-input-2-daf98e07ecdb> in get_beta(self)
     14
     15     def get_beta(self):
---> 16         return self.vel/c # try just using c. What happens? A
 'name' error!
     17
     18 par_2 = Particle('Proton', 0.938, 'GeV', 1,1.2e6)

NameError: name 'c' is not defined
```

Class variables **must** be 1) accessed through the class

> Particle.c

2) or through an instance of the class

> self.c

In [3]:
```
class Particle:
    #define class variables at top of the class
    c = 3.0e8 #speed of light

    def __init__(self, name, mass, mass_unit, charge, vel):
        self.name = name
        self.mass = mass
        self.mass_unit = mass_unit
        self.vel = vel #added velocity attribute
        self.mass_list = '{} {}'.format(mass, mass_unit)

    def mass_square(self):
        return '{} {}^2'.format(self.mass**2, self.mass_unit)

    def get_beta(self):
        return self.vel/self.c # Also use Particle.c

par_2 = Particle('Proton', 0.938, 'GeV', 1,1.2e6)
print(par_2.get_beta())
```

```
0.003157894736842105
```

Why can we assess class variable through the instance? Two ways to access the class variable:

In [4]:
```
print(par_2.c)
print(Particle.c)
```

```
380000000.0
380000000.0
```

Trying to access an attribute on an instance: 1) First check if the instance contains that attribute 2) If it doesn't then check the class or any class that it inherits from does

Our instances (par_2.c) does not actually have the c attribute, but is accessing the class's (Particle) attribute (the class variable c)

For a clearer idea of what is going on, let's print the namespace of one of our instances

In [5]:
```
print(par_2.__dict__) # no c attribute in the list
```

```
{'name': 'Proton', 'mass': 0.938, 'mass_unit': 'GeV', 'vel': 1200000.0,
'mass_list': '0.938 GeV'}
```

In [6]:
```
print(Particle.__dict__)
#class does have c attribute
#that is the value our instances see when we access the c attribute from
our instances
```

```
{'__module__': '__main__', 'c': 380000000.0, '__init__': <function Part
icle.__init__ at 0x1105d4f80>, 'mass_square': <function Particle.mass_s
quare at 0x1105e0050>, 'get_beta': <function Particle.get_beta at 0x110
5e00e0>, '__dict__': <attribute '__dict__' of 'Particle' objects>, '__w
eakref__': <attribute '__weakref__' of 'Particle' objects>, '__doc__':
None}
```

We can easily modify our attribute by changing the variable in the class definition or on the fly!

In [7]:
```
par_1 = Particle('Electron', 0.511, 'MeV', -1,1.2e7)
Particle.c = 2.99792458e8
print(Particle.c)
print(par_1.c)
print(par_2.c)
#change made to all instances
```

```
299792458.0
299792458.0
299792458.0
```

What if we used the instance to change c, rather than using the class?

In [8]:
```
par_1.c = 3.0e8
print('Particle:',Particle.c)
print('par_1',par_1.c)
print('par_2',par_2.c)
# attribute only changed for that particular instance!
```

```
Particle: 299792458.0
par_1 300000000.0
par_2 299792458.0
```

Why? Created c attribute in par_1

In [9]:
```
print(par_1.__dict__)
#we have a c attribute here!
```

```
{'name': 'Electron', 'mass': 0.511, 'mass_unit': 'MeV', 'vel': 1200000
0.0, 'mass_list': '0.511 MeV', 'c': 300000000.0}
```

In [10]:
```
print(par_2.__dict__)
#par_2 instance falls back to the class def, and does not have the attri
bute in its name space
```

```
{'name': 'Proton', 'mass': 0.938, 'mass_unit': 'GeV', 'vel': 1200000.0,
'mass_list': '0.938 GeV'}
```

So what does this mean for our get_beta method?

1)

```
return self.val/Particle.c
```

2)

```
return self.vel/self.c
```

Using 1) or 2) we can get different results.

In [11]:
```
print(par_1.get_beta())
print(par_2.get_beta())
```

```
0.04
0.004002769142377825
```

- 2) uses the instances attribute c to calculate β
  - Ability to change the class variable for an individual instance
  - Allows a sub class to override the class variable (e.g. c) if we want to. (more on this later)
- 1) uses the class's attribute c to compute β

Another Class Variable Example

Introduce a variable where we do not want to use *self*. Lets create a class variable that tracks the number of particles we have defined.

- This quantity will incriment by one for each particle we create.
- Place in init method -- runs first for each instance created

In [12]:
```
class Particle:
    #define class variables at top of the class
    num_part = 0
    c = 3.0e8 #speed of light

    def __init__(self, name, mass, mass_unit, charge, vel):
        self.name = name
        self.mass = mass
        self.mass_unit = mass_unit
        self.vel = vel #added velocity attribute
        self.mass_list = '{} {}'.format(mass, mass_unit)

        Particle.num_part += 1 #incriment num_part by 1
        #Particle used rather than self, because no particular instance
 should have a different total number of particles

    def mass_square(self):
        return '{} {}^2'.format(self.mass**2, self.mass_unit)

    def get_beta(self):
        return self.vel/self.c # Also use Particle.c
```

In [13]:
```
print(Particle.num_part)

par_1 = Particle('Electron', 0.511, 'MeV', -1,1.2e7)
par_2 = Particle('Proton', 0.938, 'GeV', 1,1.2e6)

print(Particle.num_part)
```

```
0
2
```

In [ ]: