Goals

- Class Methods
- Static Methods

Reminder:

Regular methods and classes take the instance automatically as the first argument (***self***).

Class Method

Takes the class automatically as the first argument. You create a class method with a **decarator**

> @classmethod

A **decarator** modifies the functonality of an existing function, method or class. We can use a decatrator to change the functionality of a method to accept the class as the first argument, and not the instance.

You can learn mor about decarators from this Corey Shafer video (https://www.youtube.com/watch?v=FsAPt_9Bf3U).

In [3]:
```
import numpy as np
```

Working with the class and not the instance

In [7]:

```
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

    @classmethod  # use decarator to distinquish following method as a c
lass method
    def set_c_value(cls, val): #cls is used as convention. class can not
be used (it is python key work)
        cls.c = val #note we are now working with the class and not the
 instance
```

In [9]:

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

print(Particle.c)
print(par_1.c)
print(par_2.c)
```

```
380000000.0
380000000.0
380000000.0
```

All print statements above are the same due the class variable being set.

We can change the value c to $c = 2.99 \times 10^8 m/s$ by using our ***set_c* class method**

The class method ***set_c_value*** works with the class and modifies the class variable

In [13]:

```
#Use the class to access the class method
Particle.set_c_value(2.99e8) #automatically accepts class as first argum
ent, so we don't need to specify it
print(Particle.c)
print(par_1.c)
print(par_2.c)

#You can also use the instance to access the class method
#Not common practice in Python
par_1.set_c_value(3)
print(Particle.c)
print(par_1.c)
print(par_2.c)
```

```
299000000.0
299000000.0
299000000.0
3
3
3
```

Using Class Methods as Constructors

Class methods can be used to create various constructors.

Example:

We would like to pass our particle data in as a csv string

```
'Muon,0.1057,GeV,-1,9.2e7'
```

We can prepaer the data so that it is in the appropriate form for the class

In [22]:

```
par_str_1 = 'Muon,0.1057,GeV,-1,9.2e7'

#to parse the sting we can use split method and save splits into seperat
e variables
name, mass, mass_unit, charge, vel = par_str_1.split(',')

#pass those split variables to Particle class to make a new particle
par_3 = Particle(name, mass, mass_unit, charge, vel)

#Works, yeah!
print(par_3.name)
print(par_3.vel)
```

```
Muon
9.2e7
```

That is fine if it is a one off thing. But, what if we get a lot of data in this form? This is annoying we would need to do it for all of the entries.

We can modify our class to also except this form using a class method to creat an **alternative constructor**.

In [26]:
```
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

    @classmethod  # use decarator to distinquish following method as a class method
    def set_c_value(cls, val): #cls is used as convention. class can not be used (it is python key work)
        cls.c = val #note we are now working with the class and not the instance

    @classmethod
    def from_string(cls, par_string):  #starting with from is convention for alternative constructor
        name, mass, mass_unit, charge, vel = par_string.split(',')
        return cls(name, mass, mass_unit, charge, vel) #creates new particle
        #The above is just like below which is how we usually create new particles. But we replace Particle class name
        #with cls (remeber that is the class)
        #Particle(name, mass, mass_unit, charge, vel)

        #we now need to return the new particle so we can receive the particle object when the method is called
```

In [27]:

```
par_str_1 = 'Muon,0.1057,GeV,-1,9.2e7'
par_3 = Particle.from_string(par_str_1)
print(par_3.name)
```

```
Muon
```

from_string Summary:

1) Pass in particle string\ 2) Class method (***from_string***) receives the string and parses it via split\ 3) Various parts are saved as arguments taken by the init method\ 4) Class method creates a new Particle object and returns it

Static Methods

- **Regular methods** automatically pass the instance of the class as the first argument (e.g. ***self***)
- **Class methods** automatically pass the class as the first argument (e.g. ***cls***)
- **Statice methods** do not pass anything automatically
    - Behave like regular functions

When do you use a static method?

When you have a function in your class that does not access the class or instance of the class.

In [50]:

```python
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

    @classmethod  # use decarator to distinquish following method as a class method
    def set_c_value(cls, val): #cls is used as convention. class can not be used (it is python key work)
        cls.c = val #note we are now working with the class and not the instance

    @classmethod
    def from_string(cls, par_string):  #starting with from is convention for alternative constructor
        name, mass, mass_unit, charge, vel = par_string.split(',')
        mass = float(mass)
        vel = float(vel)
        charge = int(charge)

        return cls(name, mass, mass_unit, charge, vel) #creates new particle
        #The above is just like below which is how we usually create new particles. But we replace Particle class name
        #with cls (remeber that is the class)
        #Particle(name, mass, mass_unit, charge, vel)

        #we now need to return the new particle so we can receive the particle object when the method is called

    @staticmethod
    def static_beta(vel, c):
        return vel/c
```

In [52]:

```
par_str_1 = 'Muon,0.1057,GeV,-1,9.2e7'

print(Particle.c)
par_1 = Particle('Electron', 0.511, 'MeV', -1,1.2e7)
par_2 = Particle('Proton', 0.938, 'GeV', 1,1.2e6)
par_3 = Particle.from_string(par_str_1)


print(par_3.get_beta())
print(par_3.static_beta(9.2e7,3e8))
```

```
300000000.0
0.30666666666666664
0.30666666666666664
```

In [ ]: