Goals ¶

- Special methods

Special Methods

Special methods, also known as ***magic*** methods allow us to emulate built in features of Python. These methods will allow us to impliment **operater overloading**

In [2]:
```
#depending on the contex '+' has different behaviors
print(1+2) # add two numbers
print('a' + 'b') # cocat. string
```

```
3
ab
```

In [3]:
```
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 [4]:
```
par_1 = Particle('Electron', 0.511, 'MeV', -1, 1.2e7)
par_2 = Particle('Proton', 0.938, 'GeV', 1,1.2e6)
```

In [7]:
```
print(par_1) # would like this to give more user friendly info... we can
use special methods
```

```
<__main__.Particle object at 0x1104f06d0>
```

Special methods are usually surrounded by '__', double under score, aslo refered to as dunder

> **special_method**

Our class init method is a dunder method and is a special method

Two other special methods:

- **repr** : representation of the object. Mostly seen by developers and used for debugging
- **str** : a readable representation of the object. Meant to be seen by the users.

repr, we would want to be something that can be used to recreate our object

In [17]:
```
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.charge = charge
        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

    def __repr__(self):
        #return a string which would create a Particle object
        return 'Particle({},{},{},{},{})'.format(self.name, self.mass, s
elf.mass_unit, self.charge, self.vel)

    #def __str__(self):
    #    pass
```

In [19]:
```
par_1 = Particle('Electron', 0.511, 'MeV', -1, 1.2e7)
par_2 = Particle('Proton', 0.938, 'GeV', 1,1.2e6)

print(par_1) #repr returns the string needed to create the object
```

```
Particle(Electron,0.511,MeV,-1,12000000.0)
```

Let's define the str method

In [20]:
```
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.charge = charge
        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

    def __repr__(self):
        #return a string which would create a Particle object
        return 'Particle({},{},{},{},{})'.format(self.name, self.mass, s
elf.mass_unit, self.charge, self.vel)

    def __str__(self):
        return '{}: {} {}'.format(self.name, self.mass, self.mass_unit)
```

In [22]:
```
par_1 = Particle('Electron', 0.511, 'MeV', -1, 1.2e7)
par_2 = Particle('Proton', 0.938, 'GeV', 1,1.2e6)

print(par_1) #str returns the string specified
#str get printed by default of repr.
#Both can still be accessed
```

```
Electron: 0.511 MeV
```

We can access repr or str methods by specifying them

In [24]:
```
print(repr(par_1))
print(str(par_1))
```

```
Particle(Electron,0.511,MeV,-1,12000000.0)
Electron: 0.511 MeV
```

Above are actually running the repr and str special methods

In [26]:
```
# press tab after par_1.__ for list of methods
print(par_1.__repr__())
print(par_1.__str__())
```

```
Particle(Electron,0.511,MeV,-1,12000000.0)
Electron: 0.511 MeV
```

Some more special methods

In [27]:
```
print(1+2) #uses __add__ special method of int object
```

```
3
```

In [28]:
```
print(int.__add__(1,2))
```

```
3
```

In [30]:
```
print(str.__add__('a','b')) #str object has its own __add__ special meth
od
```

```
ab
```

We can define our own dunder add method for our Particle class to get the total mass.

In [44]:
```
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.charge = charge
        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

    def __repr__(self):
        #return a string which would create a Particle object
        return 'Particle({},{},{},{},{})'.format(self.name, self.mass, s
elf.mass_unit, self.charge, self.vel)

    def __str__(self):
        return '{}: {} {}'.format(self.name, self.mass, self.mass_unit)

    def __add__(self, other):
        #convert to GeV
        if self.mass_unit == 'MeV':
            self.mass = self.mass / 1000.0
        else:
            self.mass = self.mass / 1.0

        if other.mass_unit == 'MeV':
            other.mass = self.mass / 1000.0
        else:
            other.mass = other.mass / 1.0

        return self.mass + other.mass #return in GeV
```

In [45]:
```
par_1 = Particle('Electron', 0.511, 'MeV', -1, 1.2e7)
par_2 = Particle('Proton', 0.938, 'GeV', 1,1.2e6)

print(par_1 + par_2)
#If we try this without our add method we get an error about unsupported
'+'
```

```
0.938511
```

The ***len*** method is also a special method

In [46]:
```
print(len('test'))
```

```
4
```

In [49]:
```
print('test'.__len__()) # len is just a dunder method applied to a strin
g object.
```

```
4
```

In [56]:
```
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.charge = charge
        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

    def __repr__(self):
        #return a string which would create a Particle object
        return 'Particle({},{},{},{},{})'.format(self.name, self.mass, s
elf.mass_unit, self.charge, self.vel)

    def __str__(self):
        return '{}: {} {}'.format(self.name, self.mass, self.mass_unit)

    def __add__(self, other):
        #convert to GeV
        if self.mass_unit == 'MeV':
            self.mass = self.mass / 1000.0
        else:
            self.mass = self.mass / 1.0

        if other.mass_unit == 'MeV':
            other.mass = self.mass / 1000.0
        else:
            other.mass = other.mass / 1.0

        return self.mass + other.mass #return in GeV

    def __len__(self):
        return len(self.name)
```

In [58]:
```
par_1 = Particle('Electron', 0.511, 'MeV', -1, 1.2e7)
par_2 = Particle('Proton', 0.938, 'GeV', 1,1.2e6)

print(len(par_1))
print(len(par_2))
```

```
8
6
```

In [ ]:
```

```