Lec12-objects-RyanSponzilli

October 3, 2024

1 ASTR 310 Lecture 12 - objects

1.0.1 1. Create a class

Create a class called `Star` to store data and methods associated with individual stars. It should contain the following attributes:

- name as a string
- Right Ascension as a tuple (hours, minutes, seconds)
- Declination as a tuple (degrees, arcminutes, arcseconds)
- distance as a float (pc)
- absolute magnitude as a float

The constructor method should take these as arguments and use them to set the values of the attributes associated with the object.

Give the class two methods:

- `appmag` returns the apparent magnitude using $m - M = 5(\log d - 1)$
- `__str__` returns the string representation of the star, like the following. (Use embedded newlines.)

```
Star:    Betelgeuse
RA:      05h 55m 10.3s
Dec:     +07d 24' 25.4"
Mag:     -5.85
Dist:    220.0 pc
```

Test your class by creating a new `Star` object, printing its string representation, and evaluating its apparent magnitude. [5 pts]

[76]:
```
class Star:

    def __init__(self, name: str, ra: tuple[int, int, float], dec: tuple[int,␣
↪int, float], distance: float, absMag: float):
        self.name = name
        self.ra = ra
        self.dec = dec
        self.distance = distance
        self.absMag = absMag
    def appmag(self):
        from math import log10
        return (5 * (log10(self.distance) - 1)) + self.absMag

    def __str__(self):

        return f"Star:    {self.name}\nRA:      {str(self.ra[0]).zfill(2)}h␣
↪{str(self.ra[1]).zfill(2)}m {str(self.ra[2]).zfill(2)}s\nDec:     {str(self.
↪dec[0]).zfill(2)}d {str(self.dec[1]).zfill(2)}\' {str(self.dec[2]).
↪zfill(2)}\nMag:     {self.appmag()}\nDist:    {self.distance}\n"
```

[77]:
```
star1 = Star("Betelgeuse", (5, 55, 10.3), (7, 24, 25.4), 220.0, -5.85)
print(star1)
```

```
Star:    Betelgeuse
RA:      05h 55m 10.3s
Dec:     07d 24' 25.4
Mag:     0.8621134041110317
Dist:    220.0
```

2. Modify the class Download the file `riseset.py` from the Canvas page for today's exercise. It contains a function called `riseandset()` that computes the rise and set times for a star with a given right ascension and declination as observed at a given latitude and longitude for a given year, month, and date:

```
riseandset(ra, dec, lam, phi, y, m, d)
```

The local rise and set times are returned as two tuples (H, M, S). The file also contains three helper routines.

Create a new class `Star2` which is similar to the previous `Star` class, with the following enhancement. Make the `riseandset()` routine into a method of the `Star2` class. (Import the routine, don't copy and paste the code into this notebook.) Modify the routine so that it takes the RA and Declination from the class's attributes, but continues to take the latitude, longitude, year, month, and date as arguments. Let the other routines be global functions. Try the following tests.

```
Betelgeuse = Star2("Betelgeuse", (5,55,10.3), (7,24,25.4), 220., -5.85)

print(Betelgeuse.riseandset(40.112, -88.221, 2024, 1, 30))
```

[5 pts]

[78]:
```
class Star2:
    def __init__(self, name: str, ra: tuple[int, int, float], dec: tuple[int,␣
↪int, float], distance: float, absMag: float):
        self.name = name
        self.ra = ra
        self.dec = dec
        self.distance = distance
        self.absMag = absMag


    def appmag(self):
        from math import log10
        return (5 * (log10(self.distance) - 1)) + self.absMag

    def __str__(self):

        return f"Star:    {self.name}\nRA:      {str(self.ra[0]).zfill(2)}h␣
↪{str(self.ra[1]).zfill(2)}m {str(self.ra[2]).zfill(2)}s\nDec:     {str(self.
↪dec[0]).zfill(2)}d {str(self.dec[1]).zfill(2)}\' {str(self.dec[2]).
↪zfill(2)}\nMag:     {self.appmag}\nDist:    {self.distance}\n"

    def riseandset(self, lat: float, long: float, year: int, month: int, day:␣
↪int):
        import riseset
        return riseset.riseandset(self.ra, self.dec, lat, long, year, month,␣
↪day)
```

[79]:
```
Betelgeuse = Star2("Betelgeuse", (5,55,10.3), (7,24,25.4), 220., -5.85)

print(Betelgeuse.riseandset(40.112, -88.221, 2024, 1, 30))
```

```
((14, 39, 42), (3, 25, 48))
```

1.0.2 3. Another class

Create a class called `Particle` to represent different kinds of particles in an N-body simulation. This class should have the following data:

- \$ x, y, z, vx, vy, vz\$ – 3D position and velocity values
- \$ ax, ay, az \$ – 3D acceleration values
- \$ m \$ – mass

It should have the following methods:

- `__init__`: initialize a Particle with \$ m, x, y, z, vx, vy, vz\$ passed in
- `move`: advance the particle over a timestep dt using $vx = vx + ax\,(dt)$ and \$ x = x + vx(dt) + 0.5, ax, (dt)^2\$ etc.
- `compute_accels`: compute total gravitational acceleration on the particle due to a list of particles. See the gravity review in the slides.

Create a subclass, `StarParticle`. `StarParticle` should have, in addition to the base class:

- t – age
- `age_stars`: a method that increments the particle's age by dt.

[6 pts]

[80]:
```
class Particle:

    def __init__(self, m: float, x: float, y: float, z: float, vx: float, vy:␣
↪float, vz: float):
        self.m = m
        self.x = x
        self.y = y
        self.z = z
        self.vx = vx
        self.vy = vy
        self.vz = vz
        self.ax = 0
        self.ay = 0
        self.az = 0

    def move(self, dt: float):
        self.vx = self.vx + self.ax*dt
        self.vy = self.vy + self.ay*dt
        self.vz = self.vz + self.az*dt
        self.x = self.x + self.vx*dt + 0.5*self.ax*dt**2
        self.y = self.y + self.vy*dt + 0.5*self.ay*dt**2
        self.z = self.z + self.vz*dt + 0.5*self.az*dt**2

    def compute_accels(self, particles: list):
        from math import sqrt
        atx = 0
        aty = 0
        atz = 0
        for p in particles:
            r = (p.x - self.x, p.y - self.y, p.z - self.z)
            G = 6.67430e-11
            r_abs = sqrt(r[0]**2 + r[1]**2 + r[2]**2)
            atx += G*p.m / r_abs**3 * r[0]
            aty += G*p.m / r_abs**3 * r[1]
            atz += G*p.m / r_abs**3 * r[2]

        self.ax = atx
        self.ay = aty
        self.az = atz

    def __str__(self):
        return f"{self.x} {self.y} {self.z} {self.vx} {self.vy} {self.vz} {self.
↪ax} {self.ay} {self.az}"
```

[81]:
```
class StarParticle(Particle):
    def __init__(self, m: float, x: float, y: float, z: float, vx: float, vy:␣
↪float, vz: float, t: float):
        super().__init__(m, x, y, z, vx, vy, vz)
        self.t = t

    def age_stars(self, dt):
        self.t += dt
```

Test your class and methods to verify that they're producing the output you expect. [4 pts]

[82]:
```
p1 = Particle(2e10, 0, 0, 0, 0, 0, 0)
p2 = Particle(3e10, 1, 1, 1, 0, 0, 0)
p3 = Particle(1e10, -1, 2, 1, 0, 0, 1)

p1.compute_accels([p2, p3])
p2.compute_accels([p1, p3])
p3.compute_accels([p1, p2])
print(p1)
print(p2)
print(p3)
p1.move(1)
p2.move(1)
p3.move(1)
print(p1)
print(p2)
print(p3)
```

```
0 0 0 0 0 0 0.3399280307478493 0.47616660899999735 0.4307537495826147
1 1 1 0 0 0 -0.37628743479463606 -0.19719717276791396 -0.2568939267768213
-1 2 1 0 0 1 0.44900624288820956 -0.36074169969625286 -0.09082571883476538
0.509892046121774 0.714249913499996 0.646130624373922 0.3399280307478493
0.47616660899999735 0.4307537495826147 0.3399280307478493 0.47616660899999735
0.4307537495826147
0.43556884780804583 0.704204240848129 0.614659109834768 -0.37628743479463606
-0.19719717276791396 -0.2568939267768213 -0.37628743479463606
-0.19719717276791396 -0.2568939267768213
-0.32649063566768566 1.4588874504556206 1.863761421747852 0.44900624288820956
-0.36074169969625286 0.9091742811652346 0.44900624288820956 -0.36074169969625286
-0.09082571883476538
```

[83]:
```
sp = StarParticle(2e25, 0, 0, 0, 1, 1, 1, 2e7)
print(sp.t)
sp.age_stars(1e6)
print(sp.t)
```

```
20000000.0
21000000.0
```