AST415

Astronomide Sayısal Çözümleme - I

9. Python'da Sınıf (Class) Yapısına Giriş

Bu derste neler öğreneceksiniz?

✔ Sınf Yapısına Neden İhtiyaç Duyulur?

- ✗ Problemin Kötü Bir Çözümü: Global Değişken Kullanmak
- ✗ Bir Fonksiyonu Sınıf Yapısı İçerisinde Temsil Etmek
- ✗ Örnek: Jeans Kütlesi Hesabı
- ✗ Başlatıcı Fonksiyon Olmadan Sınıf Oluşturmak (Opsiyonel)
- ✗ Örnek: Banka Hesapları (Opsiyonel)
- ✗ Örnek: Telefon Rehberi
- ✗ Örnek: Geometrik Şekiller

✔Özel Metotlar

- ✗ __call__ Metodu
- ✗ Örnek: Nümerik Türev
- ✗ __add__ Metodu
- ✗ __mul__ Metodu
- ✗ Örnek: Polinom Türevi
- ✗ Aritmetik İşlemler ve Diğer Özel Metotlar
- ✗ __repr__ Metodu
- ✗ Örnek: Vektörel İşlemler (Opsiyonel)

✔Statik Metotlar ve Öznitelikler

- ✗ Örnek: Aralık Aritmetiği

Sınıf (Class) Yapısına Neden İhtiyaç Duyulur?

- ✓ En basit tanımıyla sınıf, bir veri setini (değişkenler), bu veri seti üzerinde işlemler gerçekleştiren fonksiyonlarla birlikte paketlemeye verilen isimdir.

$$y(t) = v_0 t - \frac{1}{2} g t^2$$

$$g(x; A, a) = Ae^{-ax}$$

- ✓ Dikey atış ve harmoink hareket problemlerini dersin başından beri görüyoruz. Bu problemlerden dikey atış probleminde y (düşey konum) sadece t bağımsız değişkeninin bir fonksiyonudur. Ancak V_0 (ilk hız) da programcılık bağlamında baktığınızda bir değişkendir. g (yer çekimi ivmesi) ise problemin yapısına bağlı olarak (sadece Dünya dikkate alındığında) bir sabit şekinde düşünülebilir. Bu durumda fonksiyonu y = y(t; V_0) düşünebiliriz. Benzer şekilde harmonik hareketi de g = g(x; A, a) şeklinde düşünebiilriz. Bu iki problemi çözmek üzere yazabileceğimiz iki Python fonksiyonu aşağıda verilmiştir.

```
def y(t, v0):
    g = 9.81
    return v0*t - 0.5*g*t**2

def g(x, A, a):
    from math import exp
    return A*exp(-a*x)
```

! Problem: Matematiksel fonksiyonlar üzerine uygulayabileceğiniz pek çok başka kod, tek değişkenli bir fonksiyonun bir programlama diliyle yazılmış halinin de sadece bir argümanı olduğunu varsayar. Örnek olarak bir f(x) fonksiyonunun x noktasındaki türevini hesaplayan turev fonksiyonunu düşünelim. Bu fonksiyon bizim yukarıda verdiğimiz iki fonksiyon için de çalışmaz!

```
def turev(f, x, h=1E-10):
    return (f(x+h) - f(x)) / h
```

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

Problemin Kötü Bir Çözümü: Global Değişken Kullanmak

- ✓ Fonksiyonlarımızı aşağıdaki şekilde değiştirir ve bu fonksiyonları çağırmadan önce V_0 ve A, a değişkenlerini global değişken olarak tanımlarsak amacımıza ulaşmış, türev fonksiyonunu da üzerinde kullanabileceğimiz daha genel bir yapı oluşturmuş oluruz.

```
def y(t):
    g = 9.81
    return v0*t - 0.5*g*t**2

def g(x):
    from math import exp
    return A*exp(-a*x)
```

```
>>> v0 = 1
>>> dy = turev(y, 1)
>>> A = 1; a = 0.1
>>> dg = turev(g, 1.5)
```

- ✓ Ancak global değişken kullanımı genellikle bir “kötü programcılık” pratiği olarak değerlendirilir. Bunun bir nedeni örneğin y fonksiyonunu farklı V_0 değerleri için her çalıştırışımızda v0 değişkenini yeniden tanılmamamız gerekliliği, diğer bir nedeni ise v0, A, a değişkenlerinin programımızın başka bir noktasında değiştirilme ihtimalidir. Uzun ve global değişkenlerin sık kullanıldığı bir programda bu durumu kontrol etmek hiç kolay olmayabilir.

- ✓ Sınıf yapısı tüm bu problemleri çözer ve “iyi programcılık” sınıf yapısını doğru ve yerli yerinde kullanmaktan geçer!

Bir Fonksiyonu Sınıf Yapısı İçerisinde Temsil Etmek

- ✓ Sınıf yapısı fonksiyonları ve değişkenleri bir arada tek bir birim halinde tutar. Değişkenler sınıfın içerisindeki tüm fonksiyonlar tarafında “görülür”. Bir başka deyişle bir sınıfın içinde tanımlı bir değişken o sınıfın içerisindeki tüm fonksiyonların görebildiği bir global değişken gibi davranır.

- ✓ Dikey atış problemine iyi bir programcılık çözümü, zamanı bağımsız değişken kabul eden bir fonksiyon (y(t)) ve V_0 ile g'ye ulaşımın sağlandığı bir sınıf kullanılarak getirilebilir.

- ✓ y(t) fonksiyonuna ek olarak sınıf yapısında genellikle bulunan ve adı her zaman `__init__` olan ve tüm sınıf yapısında geçerli değişikenleri başlatan bir fonksiyona daha ihtiyaç duyulur. Python programcılığında sınıf isimleri genellikle büyük harfle başlayacak şekilde verilir.

- ✓ Sonuç olarak elimizde __init__ fonksiyonu ile düşey konumu hesaplayacak bir fonksiyon (dusey_konum) ile iki de sınıf değişkeni (V_0 ve g) bulunmaktadır. Aşağıda bu sınıfın nasıl tanımlandığını görüyorsunuz.

```
class Y:
    def __init__(self,v0):
        self.v0 = v0
        self.g = 9.81

    def dusey_konum(self,t):
        return self.v0*t - 0.5*self.g*t**2
```

- ✓ Bu noktada şimdiye kadar hiç karşılaşmadığımız bir parametreyle (`self`) parametresi ile karşılaşıyoruz. Bu parametrenin ne işe yaradığını öncelikle yukarıda verilen sınıf yapısının düşey konumu nasıl hesapladığnı anlayarak görmeye çalışalım.

- ✓ Oluşturduğumuz `Y` sınıfı Y adında yeni bir veri türü yaratır. Bu veri türü üzerinden yeni nesneler tanımlanabilir (kullanıcı taraından tanımlanan bir sınıfın nesnelerine olgu (ing. `instance`) adı verilir. Liste (list), demet (tuple), metin (string), noktalı sayı (float), tam sayı (integer) gibi nesneler özünde bu isimlerle yaratılmış birer Python sınıfıdır.

- ✓ Aşağıdaki ifade y değişkenine bağlı bir olgu (instance) yaratır.

```
y = Y(3)
```

- ✓ Python bu ifadedeki Y(3) 'ü hemen Y sınıfındaki __init__ fonksiyonunu çağırmak üzere kullanır. Bu çağrı yapılırken kullanılan değer(ler) (burada sadece 3 nümerik değeri), __init__ fonksiyonunda self parametresinin hemen arkasından gelen değişken(ler)e transfer edilir (örneğimizde v0 bu şekilde 3 değerini alır). Sınıf yapısındaki fonksiyonlarda self parametresine hiçbir değer gönderilmez. Gönderilen değer ya da değerler bu parametreden sonra gelen değişkenlere atanır.

- ✓ y olgusuyla (instance) t = 0.1 saniye ve V_0 = 3 m/s için düşey konum aşağıdaki ifadeyle elde edilebilir.

```
v = y.dusey_konum(0.1)
```

- ✓ Gördüğünüz gibi dusey_konum fonksiyonundaki self parametresine de değer gönderilmemekte, 0.1 değeri fonksiyon tanımında ondan hemen sonra gelen t değişkenine atanmaktadır.

- ✓ y oluşumunun parametrelerine (fonksiyon ve değişkenlerine), bu oluşumun adının arkasına istenen parametreyi koyarak ulaşmak mümkündür.

```
>>> print y.v0
3
```

- ✓ Bir sınıf nesnesine olgu (instance), sınıftaki fonksiyonlara metotlar (method), değişkenlere (veri) ise öznitelikler (attribute) denir.

- ✓ Örneğimizdeki Y sınıfında iki metot (__init__ ve dusey_konum), iki de öznitelik (attribute) (v0,g) bulunmaktadır. Tüm Python fonksiyonlarında olduğu gibi isimlendirme kurallarına uymak koşuluyla metot ve öznitelik isimleri de özgürce seçilebilir. Ancak başlatıcı (ing. constructor) fonksiyonun adı __init__ olarak verilmek zorundadır. Aksi takdirde yeni olgular (instances) oluştururken bu fonksiyon (metot) otomatik olarak çağrılamaz ve istenen öznitelikler (attributes) oluşmaz.

- ✓ Herhangi bir metot herhangi bir işi yapmak için oluşturulurken, __init__ metodu öznitelikleri (sınıf değişkenleri, attribute) yaratmak için oluşturulur.

- ✓ **`self`** Değişkeni: Oluşturulan olguyu (instance, örneğimizde y) __init__ fonksiyonu içerisinde tutan değişkendir.

  `y = Y(3)` yazdığımız vakit, Python bu ifadeyi geri planda `Y.__init__(y,3)` ifadesine dönüştürür. Yani `self.v0` yazdığımız vakit de y.v0 özniteliğini “başlatımış” oluruz.

  Aynı şekilde `konum = y.dusey_konum(0.1)` yazdığımız vakit Python bu ifadeyi `konum = y.dusey_konum(y,0.1)` 'e dönüştürür.

  Dolayısı ile dusey_konum fonksiyonu içerisindeki `self.v0*t – 0.5*self.g*t**2` ifade `y.v0*t – 0.5*y.g*t**2` ile aynı işlevi görür. Özet olarak self, yaratılan olgunun (instance) yerini tutar.

- ✓ Self değişkeni ile ilgili kurallar aşağıdaki gibidir:

  - ✓ Her sınıf metodunun ilk argümanı self değişkeni olmak zorundadır!
  - ✓ self, sınıfın (herhangi) bir olgusunun (instance) yerini tutar!
  - ✓ Sınıfın içindeki diğer metot ve özniteliklere ulaşmak için, ulaşılmak istenen metot ya da özniteliğin adı self değişkenin arkasına yazılır (self.metotadi ya da self.degiskenadi gibi)
  - ✓ Sınıfın fonksiyonları (metotlar) çağrılırken self bir argüman olarak verilmez. Fonksiyona (varsa) gönderilen değer self argümanından bir sonrakine atanır!

- ✓ Bir sınıfa istediğimiz kadar metot ya da öznitelik ekleyebiliriz. Dikey atış problemini çözmek üzere geliştirdiğimiz sınıfımıza (Y) bir de `formul` fonksiyonu ekleyelim.

```
def formul(self):
        return 'v0*t - 0.5*g*t**2; v0=%g' % self.v0
```

- ✓ Programımızın en altına aşağıdaki satırları ekleyerek çalıştırıp, çıktısını görebiliriz. Unutulmaması gereken, ana programın `class` ifadesi ile aynı düzeyde (aynı miktarda bloklanmış şekilde) yazılması gerekliliğidir.

```
y = Y(5)
t = 0.2
v = y.dusey_konum(t)
print 'y(t=%g; v0=%g) = %g' % (t, y.v0, v)
print y.formul()
```

```
y(t=0.2; v0=5) = 0.8038
v0*t - 0.5*g*t**2; v0=5
```

- ✓ Şimdi farklı V_0 değerleri için farkı y örnekleri oluşturarak, bu örneklerin her birinin sonucunu herhangi bir fonksiyona gönderebiliriz. Örnek olarak 3. slaytta verdiğimiz `turev` fonksiyonuna bu değeri gönderebiliriz. Böylece bu fonksiyona görünürde sadece bir değer (t) gönderdiğimz halde y örnekleri aracılığıyla v0 ve g'ye de ulaşımımız var ve istersek bu değerleri de değiştirebiliriz! Böylece problemimiz çözülmüş oldu!

```
def turev(f, x, h=1E-10):
    return (f(x+h) - f(x)) / h
```

```
y1 = Y(1)
y2 = Y(1.5)
y3 = Y(-3)
dy1dt = turev(y1.dusey_konum,0.1)
dy2dt = turev(y2.dusey_konum,0.1)
dy3dt = turev(y3.dusey_konum,0.2)
```

✓ Açıklama satırlariyla birlikte kodumuzun son hali aşağıdaki (sinif_dikeyatis.py) gibidir.

```
class Y:
    """
    Dikey atilan bir cismin t anindaki dusey konumunu hesaplayan sinif
    Metotlar (Methods):
    __init__(v0): baslangici hizi v0 'i belirler
    dusey_konum(t): cismin t'nin fonksiyonu olarak dusey konumunu hesaplar
    formul(): formulu ekrana yazdirir
    Oznitelikler (Attributes):
    v0: cismin ilk hizi (t=0 anindaki hiz)
    g: yercekimi ivmesi (sabit)
    Kullanim:
    >>> y = Y(3)
    >>> konum1 = y.value(0.1)
    >>> konum22 = y.value(0.3)
    >>> print y.formul()
    v0*t - 0.5*g*t**2; v0=3
    """
    def __init__(self,v0):
        self.v0 = v0
        self.g = 9.81

    def dusey_konum(self,t):
        return self.v0*t - 0.5*self.g*t**2

    def formul(self):
        return 'v0*t - 0.5*g*t**2; v0=%g' % self.v0

def turev(f, x, h=1E-10):
     return (f(x+h) - f(x)) / h

y = Y(5)
t = 0.2
v = y.dusey_konum(t)
print 'y(t=%g; v0=%g) = %g' % (t, y.v0, v)
print y.formul()

y1 = Y(1)
y2 = Y(1.5)
y3 = Y(-3)
dy1dt = turev(y1.dusey_konum,0.1)
dy2dt = turev(y2.dusey_konum,0.1)
dy3dt = turev(y3.dusey_konum,0.2)
print dy1dt, dy2dt, dy3dt
```

Örnek: Jeans Kütlesi Hesabı

- ✓ Bir gaz bulutunun kendi çekim etkisi altında çökerek yıldız oluşturabileceği limit kütleye Jeans Kütlesi adı verilir. Jeans kütlesi gaz bulutunun sıcaklığı (T), ortalama molekül ağırlığı (μ) ve ortalama yoğunluğa (ρ) bağlıdır. Yıldız oluşumun gerçekleştiği gaz bulutlarının ortalama molekül ağırlığı ile yoğunluğunu ölçmek kolay olmadığı için genellikle bu değerler teorik bazı değerlere eşit kabul edilerek işlem yapılır. Bu nedenle Jeans Kütlesi, sıcaklığın temel bağımsız parametre olarak kabul edilebileceği M(T; μ, ρ)) bir fonksiyonla ifade edilebilir. k (Boltzman sabiti), m_H (Hidrojen atomunun kütlesi) ve G (evrensel çekim sabiti) ise fonksiyonun sabitleridir. Π ise kolaylıkla math modülünden çekilebilecek matematiksel bir sabittir.

$$M_{\text{Jeans}} \sim \left(\frac{5kT}{G\mu m_{\text{H}}}\right)^{\frac{3}{2}} \left(\frac{3}{4\pi\rho}\right)^{\frac{1}{2}}$$

- ✓ Bu hesap için yazacağımız sınıf doğal olarak bir başlatıcı fonksiyon (__init__) ve bir de hesabı yapan fonksiyon (hesap) icermelidir. Sınıfımız adını JeansKutlesi olarak belirlemiş olalım.

```
class JeansKutlesi:
    def __init__(self,mu,rho):
        # Boltzman sabiti
        self.k = 1.3806488e-23 # J/K
        # Hidrojen atomunun kutlesi
        u = 1.660538921e-27 # kg (atomik birim kutle)
        self.mH = 1.00784*u # kg
        # Evrensel cekim sabiti
        self.G = 6.67408e-11 # m3 / (kg s^2)
        # Ortalama Molekul Agirligi ve Ortalama Yogunluk
        self.mu,self.rho = mu,rho

    def hesap(self,T):
        from math import pi
        mu,rho,k,mH,G = self.mu,self.rho,self.k,self.mH,self.G
        M = ((5*k*T)/(G*mu*mH))**(3./2.)*(3./(4*pi*rho))**(1./2.)
        return M
```

✓ Açıklama satırlariyla birlikte kodumuzun son hali aşağıdaki (sinif_jeanskutlesi.py) gibidir.

```
class JeansKutlesi:
    """
    Bir gaz bulutunun kendi cekim etkisi altinda cokmesi icin sahip olmasi
    gereken limit kutleye Jean's Limiti ya da Jean's kutlesi adi verilir.
    Bu sinif bu limit kutleyi hesaplamaktadir.
    Metotlar (Methods):
    __init__(mu,rho): Gaz bulutunun yogunlugu (rho) ve kimyasal yapisini
    (ortalama molekul agirligi, mu) belirleyen baslatici metot.
    hesap(T): Verilen bir gaz bulutu icin sicakliga bagli olarak Jean's kutlesi
    hesabini yapan metot
    Oznitelikler (Attributes):
    mu: Ortalama molekul agirligi (kg)
    rho: Ortalama yogunluk (kg / m^3)
    k: Boltzmann sabiti
    mH: Hidrojen atomunun kutlesi (kg)
    G: Evrensel cekim sabiti (m^3 / (kgs^2))
    T: sicaklik (K)
    Kullanim:
    >>> m = JeansKutlesi(100)
    >>> Mj = m.hesap(mu=2.,rho=3.3e-18)
    >>> print Mj
    """

    def __init__(self,mu,rho):
        # Boltzman sabiti
        self.k = 1.3806488e-23 # J/K
        # Hidrojen atomunun kutlesi
        u = 1.660538921e-27 # kg (atomik birim kutle)
        self.mH = 1.00784*u # kg
        # Evrensel cekim sabiti
        self.G = 6.67408e-11 # m3 / (kg s^2)
        # Ortalama Molekul Agirligi ve Ortalama Yogunluk
        self.mu,self.rho = mu,rho

    def hesap(self,T):
        from math import pi
        mu,rho,k,mH,G = self.mu,self.rho,self.k,self.mH,self.G
        M = ((5*k*T)/(G*mu*mH))**(3./2.)*(3./(4*pi*rho))**(1./2.)
        return M

# Ortalama molekul kutlesini 2, yogunlugu 3.3e-18 kg/m^3 kabul edelim
m = JeansKutlesi(mu=2.0,rho=3.3e-18)
# Farkli sicakliklarda boyle bir gaz bulutunun kendi cekim etkisi altinda
# cokebilmesi icin hangi kutleye sahip olmasi gerektigine bakalim
import numpy as np
T = np.array([10,50,100,250,500,1000])
Mj = m.hesap(T)
# Kutleyi gunes kutlesi cinsinden ifade edelim
Mgunes = 1.989e30 #kg
Mj = Mj / Mgunes
# Simdi sicakliga karsilik kutleyi cizdirelim
from matplotlib import pyplot as plt
plt.plot(T,Mj,"ro")
plt.xlim((-100,1100))
plt.ylim((-1000,25000))
plt.show()
```

Başlatıcı Fonksiyon Olmadan Sınıf Oluşturmak (Opsiyonel!)

- ✓ Sınıf oluştururken başlatıcı fonksiyon (`__init__`) kullanmak iyi bir yoldur ancak bir zorunluluk değildir. Dikey atış problemini çözmek üzere yarattığımız Y sınıfından `__init__` başlatıcı fonksiyonunu çıkarıp, `dusey_konum` fonksiyonuna v0 değişkenini opsiyonel (değer gönderilmesi zorunlu olmayan) bir değişken yapalım. Bu değişkene varsayılan olarak `None` değerini atayalım ki kullanıcı bu değeri sağlamazsa bu bir hataya neden olmadan çözüm arayabilelim. Y sınıfının alternatif versiyonu olan Y2 sınıfı aşağıdaki gibidir.

```
class Y2:
    def dusey_konum(self,t,v0=None):
        if v0 is not None:
            self.v0 = v0
        g = 9.81
        return self.v0*t - 0.5*self.g*t**2
```

- ✓ Bu durumda Y2 sınıfının tek bir metodu (dusey_konum) ve tek bir özniteliği (v0) olur. g ise yerel bir değişken durumundadır. Bu durumda v0 özniteliği bir başlatıcı fonksiyon tarafından “başlatılmadığı” için nasıl değer alacağı gibi bir sorunla karşılaşılır. Python'ın bu probleme çözümü “boş bir başlatıcı” sağlamasıdır.

```
y = Y2()    # Baslatici fonksiyon olmaksizin olgu bu sekilde olusturulur
```

- ✓ Ancak bu durumda `print y.v0` ifadesi hata döndürür (`AttributeError: Y2 instance has no attribute 'v0'`) çünkü y'nin bir başlatıcısı yoktur. Bu nedenle v0 başlatılamaz ve bir ilk değer de alamaz. Ancak aynı çıktı ifadesi `(print y.v0), v = y.dusey_konum(0.1, 5))` ifadesi sonrası verilecek olursa bu kez çıktı olarak ekranda 5 görürüz ve bir hata mesajı almayız! Çünkü v0 dusey_konum metodu içerisinde “başlatılmıştır”.

- ✓ Kodu aşağıdaki şekilde (v0 sağlamadan) başlatmamız durumunda v0 değer alamayacağı için, dusey_konum metodu içerisinde istediğimiz hesap da yapılamaz ve `AttributeError: Y2 instance has no attribute 'v0'` hatası alırız.

```
y = Y2()
v = y.dusey_konum(0.1)
```

- ✓ Bu hata mesajının yerine kullanıcının daha işine yarayacak bir hata mesajı sağlamak için Python'un `hasattr` fonksiyonundan faydalanabiliriz. `hassatr(self, 'v0')` ifadesi self olgusu v0 adında bir özniteliğe (attribute) sahipse `True` döndürür.

```
class Y2:
    def dusey_konum(self,t,v0=None):
        if v0 is not None:
            self.v0 = v0
        if not hasattr(self, 'v0'):
            print "dusey_konum fonksiyonunu dusey_konum(t) "\
                "seklinde cagirmadan once v0 ilk hizini belirlemek "\
                "uzere dusey_konum(t,v0) seklinde cagirmalisiniz!"
            return None
        g = 9.81
        return self.v0*t - 0.5*self.g*t**2
```

- ✓ Alternatif bir diğer uyarlama try-except bloğu ve TypeError hatasından faydalanmakla yapılabilir.

```
class Y2:
    def dusey_konum(self,t,v0=None):
        if v0 is not None:
            self.v0 = v0
        g = 9.81
        try:
            sonuc = self.v0*t - 0.5*g*t**2
        except AttributeError:
            msj = "dusey_konum fonksiyonunu dusey_konum(t) "
                "seklinde cagirmadan once v0 ilk hizini belirlemek "
                "uzere dusey_konum(t,v0) seklinde cagirmalisiniz!"
        raise TypeError(msj)
        return sonuc
```

Sınıflara Dayalı Programcılık Örnekleri: Banka Hesapları (Opsiyonel)

- ✓ Banka hesapları konusu sınıf kavramına iyi bir örnek teşkil eder. Bir banka hesabının verileri tipik olarak hesap sahibinin adı, hesap numarası ve anlık para miktarı gibi bilgilerdir. Bir hesapla yapabileceğiniz üç şey para çekmek, para yatırmak ve hesabın anlık görüntüsünü almak olabilir. (Günümüz bankacılığında bunların çok ötesinde işler yapılabilmekle örneğimiz için bu işlemler yeterlidir) Bu işlemleri metotlar ile sağlamamız (modellememiz) gerekir.

```
class Hesap:
    def __init__(self, ad, hesapno, baslangic_miktari):
        self.ad = ad
        self.hesapno = hesapno
        self.pmiktari = baslangic_miktari
    def pyatirma(self, miktar):
        self.pmiktari += miktar
    def pcekme(self, miktar):
        self.pmiktari -= miktar
    def hesapcikti(self):
        s = "%s, %s, Hesap Durumu: %s" % \
        (self.ad, self.hesapno, self.pmiktari)
        print s
```

- ✓ Şimdi bu sınıfı nasıl kullanacağımızı görelim.

```
>>> from sinif_bankahesabi import Hesap
>>> a1 = Hesap('Sir James Jeans', '19371554951', 20000)
>>> a2 = Hesap('Lord Rayleigh', '19371564761', 20000)
>>> a1.pyatirma(1000)
>>> a1.pcekme(4000)
>>> a2.pcekme(10500)
>>> a1.pcekme(3500)
>>> print "a1'in hesap durumu:", a1.pmiktari
a1'in hesap durumu: 13500
>>> a1.hesapcikti()
Sir James Jeans, 19371554951, Hesap Durumu: 13500
>>> a2.hesapcikti()
Lord Rayleigh, 19371564761, Hesap Durumu: 9500
```

- ✓ Görüldüğü üzere burada sınıfı yaratan kişi hesabın adı, numarası ve anlık para miktarının doğrudan kullanıcı tarafından değiştirilmesini istemiyor. İstenen kullanıcının sadece `pyatirma`, `pcekme` ve `hesapcikti` metotlarını çağırması ve (eğer isterse) hesap durumunu inceleyebilmesi; ancak diğer bilgileri doğrudan değiştirememesi. Bunun için Python'da özel bir tür olmamakla birlikte kullanıcı tarafından değiştirilmesi istenmeyen öznitelik isimlerinin başına "_" (altçizgi) karakteri konur. Bu karakterle başlayan öznitelikleri "dokunulamaz", metotlar "çağırılamaz". Bu isimler "koruma altındaki" (protected) isimler olarak adlandırılır ve metotların içerisinde kullanılabilir, ancak dışında kullanılamazlar.

- ✓ Örneğimizde hesap sahibinin adı, numarası ve anlık para miktarının (para çekme ve yatırma dışında) doğrudan değiştirilmesini istemeyeceğimiz için bu öznitelikleri "_" karakteriyle başlatarak korunmalarını sağlayabiliriz. Ancak hesaptaki paranın durumunu görebilmek için para miktarına ulaşmamız gerekir, bunun için de pmiktari özniteliğine ek olarak bir de pmiktari_oku metodu tanımlamak sorunumuzu çözecektir. Bu durumda kodumuz:

```
class Hesap2:
    def __init__(self, ad, hesapno, baslangic_miktari):
        self._ad = ad
        self._hesapno = hesapno
        self._pmiktari = baslangic_miktari
    def pyatirma(self, miktar):
        self._pmiktari += miktar
    def pcekme(self, miktar):
        self._pmiktari -= miktar
    def pmiktari_oku(self):
        return self._pmiktari
    def hesapcikti(self):
        s = "%s, %s, Hesap Durumu: %s" % \
        (self.ad, self.hesapno, self.pmiktari)
        print s
```

```
>>> from sinif_bankahesabi import Hesap2
>>> a1 = Hesap2('Sir James Jeans', '19371554951', 20000)
>>> a2 = Hesap2('Lord Rayleigh', '19371564761', 20000)
>>> a1.pyatirma(1000)
>>> a1.pcekme(4000)
>>> a1.pcekme(3500)
>>> a1.hesapcikti()
Sir James Jeans, 19371554951, Hesap Durumu: 13500
>>> print a1._pmiktari    # yanlis hesap durumu ogrenme yontemi ama calisir
13500
>>> print a1.pmiktari_oku() # dogru hesap durumu ogrenme yontemi
13500
>>> a1._hesapno = '19371554955' # bu “ciddi bir suctur!” ve programiniz bunu engellemeli!
```

Sınıflara Dayalı Programcılık Örnekleri: Telefon Rehberi

- ✓ Modern bir telefon rehberi (örneğin cep telefonunuzdaki) ad, telefon numarası, e-mail adresi, adres gibi bilgileri tutar. Kişisel bu bilgilere programınızla ulaşmanın iyi bir yolu `Kisi` isimli bir sınıf yaratmaktır. Böyle bir sınıfın öznitelikleri (attributes) ad, cep telefonu numarası (gsm), ofis numarası, ev numarası ve e-mail adresi olabilir. Başlatıcı fonksiyon bu öznitleliklerden bazılarını başlatır. Bazı ek öznitelikler ise sınıfın diğer metotları çağrılarak başlatılabilir. Metotlardan biri veriyi ekrana göstermek için kullanılmalıdır. Diğer metotlar başka telefon numaraları ya da e-mail adresi eklemek için kullanılabiir (sinif_telefonrehberi.py).

```
class Kisi:
    def __init__(self, ad,
                 cep_telefonu=None, is_telefonu=None,
                 ev_telefonu=None, eposta=None):
        self.ad = ad
        self.cep = cep_telefonu
        self.ofis = is_telefonu
        self.ev = ev_telefonu
        self.email = eposta
    def cep_numarasi_ekle(self, numara):
        self.cep = numara
    def is_numarasi_ekle(self, numara):
        self.ofis = numara
    def ev_numarasi_ekle(self, numara):
        self.ev = numara
    def eposta_ekle(self, adres):
        self.email = adres
```

```
>>> from sinif_telefonrehberi import Kisi
>>> k1 = Kisi('Stephen Hawking',is_telefonu='+1-555 1234567',eposta='hawking.cambridge.edu')
>>> k2 = Kisi('Neil deGrasse Tyson',is_telefonu='+1-555-1234567')
>>> k2.eposta_ekle('tyson@mit.edu')
>>> telefonrehberi = [k1,k2]
```

- ✓ Kisi sınıfının bir olgusunu (instance) güzel bir şekilde ekrana gösterebilmek için de bir metot olması iyi bir fikirdir.

```
def rehbercikti(self):
    s = self.ad + '\n'
    if self.cep is not None:
        s += 'Cep Telefonu: %s\n' % self.cep
    if self.ofis is not None:
        s += 'Is Telefonu: %s\n' % self.ofis
    if self.ev is not None:
        s += 'Ev Telefonu: %s\n' % self.ev
    if self.email is not None:
        s += 'E-posta Adresi: %s\n' % self.email
    print s
```

- ✓ Bu metotla telefon rehberinin herhangi bir andaki görüntüsünü kolaylıkla elde edebiliriz.

```
>>> for kisi in telefonrehberi:
        kisi.rehbercikti()

Stephen Hawking
Is Telefonu: +1-555-1234567
E-posta Adresi: hawking.cambridge.edu

Neil deGrasse Tyson
Is Telefonu: +1-555-1234567
E-posta Adresi: tyson@mit.edu
```

Sınıflara Dayalı Programcılık Örnekleri: Geometrik Şekiller: Çember

- ✓ Geometrik şekiler (örneğin bir çember) sınıf kavramını anlayabilmek açısından iyi bir örnektir. Bir çember merkez noktasının koordinatları (x_0,y_0) ve yarıçapı (R) ile tekil olarak tanımlanabilir. Bu üç sayıyı bir sınıfın öznitelikleri olarak değerlendirebilliriz. Bu sayılar başlatıcı metotta (__init__) başlatılabilir. Sınıfın diğer metotları ise alan ve cevre olabilir (sinif_cember.py).

```
class Cember:
    def __init__(self, x0, y0, R):
        self.x0, self.y0, self.R = x0, y0, R
    def alan(self):
        from math import pi
        return pi*self.R**2
    def cevre(self):
        from math import pi
        return 2*pi*self.R
```

```
>>> from sinif_cember import Cember
>>> c = Cember(2,-1,5)
>>> print '%g yaricapina sahip merkez koordinatlari (%g,%g) olan bir cemberin alani %g dir' %\
          (c.R, c.x0, c.y0, c.alan())

5 yaricapina sahip merkez koordinatlari (2,-1) olan bir cemberin alani 78.5398 dir
```

- ✓ Bu kavram pek çok geometrik şeklin (dikdörtgen, üçgen, elips, dikdörtgenler prizması olarak düşünülebilecek bir kutu, küre ...) alan ve çevresini hesap etmek üzere uygulanabilir (bkz. Ödev 10).

- ✓ Programcılıkta bir problemin genellikle pek çok çözümü bulunur. Örneğimizde çemberin merkez koordinatları ve yarıçapı bir listenin üyeleri olarak düşünülebilir ve metotlar buna uygun olarak da düzenlenebilir.

```
class Cember2:
    def __init__(self, x0, y0, R):
        self.cember = [x0, y0, R]
    def alan(self):
        from math import pi
        return pi*self.cember[2]**2
    def cevre(self):
        from math import pi
        return 2*pi*self.cember[2]
```

```
>>> from sinif_cember import Cember2
>>> c = Cember2(2,-1,5)
>>> print '%g yaricapina sahip merkez koordinatlari (%g,%g) olan bir cemberin alani %g dir' %\
          (c.cember[2], c.cember[0], c.cember[1], c.alan())

5 yaricapina sahip merkez koordinatlari (2,-1) olan bir cemberin alani 78.5398 dir
```

- ✓ Ya da çember, koordinatları ve yarıçapını anahtar olarak alan sözlük (dictionary) türünde bir öznitelik olarak da tanımlanabilir.

```
class Cember3:
    def __init__(self, x0, y0, R):
        self.cember = {'merkez':(2,-1),'yaricap':5}
    def alan(self):
        from math import pi
        return pi*self.cember['yaricap']**2
    def cevre(self):
        from math import pi
        return 2*pi*self.cember['yaricap']
```

Özel Metotlar
1. __call__ özel metodu

- ✓ Daha önce gördüğünüz başlatıcı metot __init__ gibi "__" ile başlayıp biten, başka bazı "özel metotlar" da bulunmaktadır. Bu metotlar olgular arasında aritmetik işlemler, karşılaştırmalar (>, <, ==, != gibi) yapmak, sıradan bir fonksiyon çağırır gibi olguları çağırmak ve bir olgunun Boolean (True ya da False) değerini belrilemek gibi işlevleri görürler.

- ✓ Bir olguyu (instance) tıpkı bir fonksyon gibi çağırabilmek için (örneğin dikey atışta düşey konum hesaplayan dusey_konum(t) fonksiyonunu çağırmak için y.dusey_konum(t) yerine y(t) yazıp aynı hesaplamayı yaptırmak istersek) kullanmamız gereken özel metot `__call__` metodudur.

```
class Y:
    ...
    ...
    def __call__(self,t):
        return self.v0*t - 0.5*g*t**2
```

- ✓ İyi bir programcılık prensibi matematiksel bir fonksiyon işlevi içeren tüm sınıfların `__call__` metoduna sahip olmaları ve işlemlerin bu metot içerisinde yapılmasıdır. Bu şekilde `__call__` metodu içeren tüm olgular, "çağrılabilir nesneler" olarak tanımlanır (tıpkı fonksiyonlar gibi!).

- ✓ Bu şekilde programlanmış Y sınıfının bir olgusu, daha önce tanımladığımız (Slayt 8) `turev` fonksiyonuna bir fonksiyon argümanı olarak gönderilebilir.

```
>>> from sinif_dikeyatis import Y
>>> y = Y(v0 = 5)
>>> dydt = turev(y,0.1)
```

Örnek: Nümerik Türev

- ✓ Problem: Python diline entegre edilmiş bir f(x) matematiksel fonksiyonu için bir Python fonksiyonu gibi davranan ve f(x)'in türevini alan (f'(x)) nesnesi tanımlamak istiyor olalım. Diyelim ki bu nesnein türü `Turev` olsun. Bu durumda kodumuz aşağıdaki gibi çalışmalıdır.

```
>>> def f(x):
        return x**3

>>> dfdx = Turev(f)
>>> x = 2
>>> print dfdx(x)
12.000000992884452
```

- ✓ Yani adını Turev koyduğumuz sınıfın dfdx olgusu tıpkı bir fonksiyon gibi x^3 fonksiyonunun türevini alıp ($3x^2$) herhangi bir x için değerini döndürebilmelidir. Türev fonksiyonu için basit bir yaklaşım kulllanabiliriz. Daha iyi bir hassasiyet için başka bir yaklaşım (nümerik algoritma) kullanmak gerekebilir.

```
class Turev:
    def __init__(self, f, h=1E-9):
        self.f = f
        self.h = float(h)
    def __call__(self, x):
        f, h = self.f, self.h # daha kisa yazabimek icin donusum
        return (f(x+h) - f(x))/h
```

- ✓ Örnek olarak sinüs fonksiyonunun x = π noktasındaki türevini alıp, gerçek değeri ile (sin(x=π)' = cos(x=π) = -1) ile karşılaştıralım.

```
>>> from math import cos,sin,pi
>>> from sinif_turev import Turev
>>> df = Turev(sin)
>>> x = pi
>>> df(x)
-1.000000082740371
>>> cos(x)
-1.0
```

- ✓ Bir başka örnek olarak x^3 fonksiyonunu tanımlayalım ve x = 1 noktasındaki türevini alıp, gerçek değeri (f'(x= 1) = $3x^2$ = 3) ile karşılaştıralım.

```
>>> def g(t)
        return t**3

>>> dg = Turev(g)
>>> t = 1
>>> dg(t)   # sonucu x**3 un turevi 3x**2 nin x=1 noktasindeki degeri 3 ile karsilastiralim
3.000000248221113
```

Örnek: Nümerik İntegrasyon

- ✓ Problem: Python diline entegre edilmiş bir f(x) matematiksel fonksiyonu için bir Python fonksiyonu gibi davranan ve f(x)'in integralini alan bir nesne tanımlamak istiyor olalım. Diyelim ki bu nesnein türü `Integral` olsun ve integrasyonu da yamuk yöntemiyle alıyor olalım. Bu durumda kodumuz şu şekilde çalışmalıdır.

$$F(x;a) = \int_a^x f(t)dt$$

$$\int_a^x f(t)dt = h\left(\frac{1}{2}f(a) + \sum_{i=1}^{n-1} f(a+ih) + \frac{1}{2}f(x)\right)$$

```
>>> from sinif_integral import Integral
>>> def f(x):
        return exp(-x**2)*sin(10*x)

>>> a = 0, n = 200     # n kullanilan yamuk sayisi
>>> F = Integral(f,a,n)
```

- ✓ Öncelikle yamuk yöntemini Python'a adapte edelim.

```
def yamuk_yontemi(f, a, x, n):
    h = (x-a)/float(n)
    I = 0.5*f(a)
    for i in range(1, n):
        I += f(a + i*h)
    I += 0.5*f(x)
    I *= h
    return I
```

- ✓ Bu durumda Integral fonksiyonunun tamamı aşağıdaki şekilde olur (sinif_integral.py).

```
def yamuk_yontemi(f, a, x, n):
        h = (x-a)/float(n)
        I = 0.5*f(a)
        for i in range(1, n):
            I += f(a + i*h)
        I += 0.5*f(x)
        I *= h
        return I
class Integral:
    def __init__(self, f, a, n=100):
        self.f, self.a, self.n = f, a, n
    def __call__(self, x):
        return yamuk_yontemi(self.f, self.a, x, self.n)
```

- ✓ Aslında yamuk_yonteminin tamamını `__call__` fonksiyonunun içine de yazabilirdik ama ayrı bir fonksiyon olarak yazmayı tercih ettik. Bunda hiçbir sakınca olmadığı gibi `__call__` fonksiyonunun yapısını da böylece basit tutmuş oluyoruz. Bu sınıfın kullanıldığı örnek bir çalışma aşağıdaki gibidir.

```
>>> from sinif_integral import Integral
>>> from sinif_integral import yamuk_yontemi
>>> from math import sin,pi
>>> G = Integral(sin,0,200)
>>> print G(2*pi)
6.46022049487e-17                # Bir baska deyisle 0
>>> print yamuk_yontemi(sin,0,2*pi,200)      # alternatif calistirma
6.46022049487e-17
```

2. Bir Olguyu Metne Dönüştürmek: __str__ özel metodu

- ✓ Bir başka özel metot `__str__` özel metodudur. Bu metot bir sınıfa ait olgu ekranda gösterilmek (print) istendiğinde çağrılır. Eğer olgunun bir `__str__` metodu varsa ve bu metot bir metin (string) döndürüyorsa döndürülen metin, aksi takdirde sınıfın adı yazdırılır. Örneğin dikey atış probleminin çözümü için yazdığımız Y sınıfının bir olgusunu ekrana yazdırmaya çalışalım.

```
>>> from sinif_dikeyatis import Y
>>> y = Y(v0=5)
>>> print y
<sinif_dikeyatis.Y instance at 0xa72142c>
```

- ✓ `print` ifadesi Y sınıfınn bir olgusu olan y'nin bulunduğu hafıza adresini ekrana yazdırmaktadır. Eğer bunun yerine hesaplanan düşey konumu ekrana yazdırmak istiyorsak bu kez bir `__str__` metoduna ihtiyaç duyarız.

```
class Y:
    ...
    ...
    def __str__(self):
        return 'v0*t - 0.5*g*t**2; v0 = %g' % self.v0
```

- ✓ Gördüğünüz gibi __str__ fonksiyonu bizim Y sınıfımızdaki formul metodunun yerini, __call__ ise dusey_konum metodunun yerini tutuyor.

- ✓ Bu özel metotları daha önce yazdıklarımızla değiştirecek olursak aşağıdaki kodu elde ederiz (sinif_dikeyatis3.py).

```
class Y:
    def __init__(self, v0):
        self.v0 = v0
        self.g = 9.81
    def __call__(self, t):
        return self.v0*t - 0.5*self.g*t**2
    def __str__(self):
        return 'v0*t - 0.5*g*t**2; v0=%g' % self.v0
```

- ✓ Kodumuzun örnek bir çalışması aşağıdaki gibidir.

```
>>> from sinif_dikeyatis3 import Y
>>> y = Y(1.5)
>>> y(0.2)
0.1038
>>> print y
v0*t - 0.5*g*t**2; v0=1.5
```

Özel Metot Örnekleri: Telefon Rehberi

- ✓ 17. slaytta gördüğümüz Telefon Rehberi uygulamasının `Kisi` isimli sınıfını tekrar ele alalım. Bu sınıfta bulunan ve telefon rehberinin herhangi bir andaki görüntüsünü almamızı sağlayan `rehbercikti` fonkksiyonu yerine __str__ özel metodunu kullanmak daha "Pythonic" bir yol olarak değerlendirilmelidir. Bunu yapmak oldukça kolaydır. `rehbercikti` fonksiyonun adını __str__ ile değiştirip `print s` ifadesi yerine de `return s` ifadesi kullanmamız yeter.

- ✓ `Kisi` sınıfının bir olgusunu sözlük değişkeni içerisinde saklamak bir telefon rehberi oluşturmak için iyi bir yoldur. Ancak bunun yerine `TelefonRehberi` adında bir sınıf oluşturup, sözlük değişkenini onun içinde kullanmayı tercih edeceğiz.

```
class TelefonRehberi:
    def __init__(self):
        self.kisiler = {} # Kisi sinifinin olgularindan olusan sozluk
    def ekle(self, ad,
             cep_telefonu=None, is_telefonu=None,
             ev_telefonu=None, eposta=None):
        p = Kisi(ad, cep_telefonu, is_telefonu, ev_telefonu, eposta)
        self.kisiler[ad] = p
```

- ✓ `__str__` metodu ise oluşturulan telefon rehberinin bir t anındaki görüntüsünü ekrana yazdırmalıdır.

```
def __str__(self):
    S = ''
    for p in sorted(self.kisiler):
        s += str(self.kisiler[p])
    return s
```

- ✓ Kisi sınıfının bir olgusunu oluşturmak için __call__ özel metodunu bilgilerini çağıracağımız kişinin adını argüman olarak vererek çağırmalıyız. Bu metodun sağladığı tek şey yazım kolaylığıdır. Rehber1 isimli bir telefon rehberindeki (TelefonRehberi olgusundaki) Ozgur Basturk'ün telefon numarasını `Rehber1['Ozgur Basturk']` şeklinde çağırmak `Rehber1.kisiler['Ozgur Basturk']` diye çağırmaktan daha kolay ve açıktır.

```
def __call__(self, ad):
        return self.kisiler[ad]
```

- ✓ Şimdi programımızı bir çalıştıralım ve nasıl işlediğine bakalım.

```
>>> from sinif_telefonrehberi2 import *
>>> Rehber1 = TelefonRehberi()
>>> Rehber1.ekle('Claudio Ranieri', is_telefonu='+44-116-1110000', eposta='boss@lcfc.co.uk')
>>> Rehber1.ekle('Jamie Vardy', is_telefonu='+44-116-5551234', eposta='vardy@lcfc.co.uk')
>>> Rehber1.ekle('Riyad Mahrez', is_telefonu='+44-116-7771234', eposta='mahrez@lcfc.co.uk')
>>> print Rehber1
Claudio Ranieri
Is Telefonu: +44-116-1110000
E-posta Adresi: boss@lcfc.co.uk
Jamie Vardy
Is Telefonu: +44-116-5551234
E-posta Adresi: vardy@lcfc.co.uk
Riyad Mahrez
Is Telefonu: +44-116-7771234
E-posta Adresi: mahrez@lcfc.co.uk
```

! Önemli Not: Bu programı, test örneğimizi de içerecek şekilde ders9_ornekler.tar.gz dosyasının içinde sinif_telefonrehberi2.py ismiyle bulabilirsiniz. Bu kodu öncelikle elinizle çalıştırmaya çalışarak anlamaya gayret ediniz. Elinizle çalıştırdığınızla elde ettiğiniz çıktıyla kabukta çalıştırdığınızda elde ettiğiniz çıktının aynı olup olmadığını varsa fark(lar)ın neden(ler)ini anlamaya çalışınız. Bu örneği iyi anlamış olmanız programcılıkta sınıf konseptini kullanmak konusunda size ciddi mesafe aldıracaktır.

3. İki Olguyu “Toplamak”: __add__ özel metodu

- ✓ Eğer bir C sınıfında tanımlı bir `__add__` özel metodu bulunuyor ise bu sınıfın iki ayrı olgusu (instance) a ve b toplanabilir ve toplam (yeni bir olgu olarak) bu metodla döndürülür.

```
class c:
    ...
    ...
    def __add__(self,diger):
        ...
```

- ✓ **Örnek Problem (Polinomlar):** Polinomlar üzerinde işlem yapacak bir program yazmak üzere `Polinom` adında bir sınıf tanımlayalım. Bu sınıfın başlatıcı fonksiyonuna polinomun katsayılarını bir liste halinde geçirebiliriz. `Polinom([1,0,-1,2])` bu durumda $1 + 0.x - 1.x^2 + 2.x^3 = 1 - x^2 + 2x^3$ polinomunun karşılığı olur. İki polinom toplanabileceği için sınıfımızın bir __add__ metodu olması doğaldır. Verilen bir x değeri için polinomun değerini döndürecek bir __call__ metodu da olmalıdır (sinif_polinomlar.py).

```
class Polinom:
    def __init__(self, katsayilar):
        self.katsayi = katsayilar
    def __call__(self, x):
        s = 0
        for i in range(len(self.katsayi)):
            s += self.katsayi[i]*x**i
        return s
    def __add__(self, diger):
        # uzun olan listeyle baslayip digerini onun uzerine ekle
        if len(self.katsayi) > len(diger.katsayi):
            toplam_katsayi = self.katsayi[:] # katsayilari kopyala
            for i in range(len(diger.katsayi)):
                toplam_katsayi[i] += diger.katsayi[i]
        else:
            toplam_katsayi = diger_katsayi[:] # katsayilari kopyala
            for i in range(len(self.katsayi)):
                toplam_katsayi[i] += self.katsayi[i]
        return Polinom(toplam_katsayi)
```

Polinom Sınıfına Eklentiler: Polinomlarla Diğer İşlemler
__mul__ özel metodu

- ✓ Polinom sınıfına bir de çarpma işlemi ekleyelim. Bu işlem biraz daha komplike bir matematiğe sahiptir.

$$\left(\sum_{i=0}^{M} c_i x^i\right)\left(\sum_{j=0}^{N} d_j x^j\right) = \sum_{i=0}^{M}\sum_{j=0}^{N} c_i d_j x^{i+j}$$

- ✓ Sonuçta iki toplam işlemi içiiçe görünüyor dolayısıyla Python'a entegrasyon da içiçe iki döngü kullanmayı gerektirecektir. Ancak öncelikle yapılması gereken, sonucu saklamak üzere M + N + 1 (sabit terim için 1 ekliyoruz) uzunluklu bir liste oluşturmaktır.

```
def __mul__(self, diger):
        c = self.katsayi
        d = diger.katsayi
        M = len(c) - 1
        N = len(d) - 1
        sonuc_katsayi = zeros(M+N+1)
        for i in range(0, M+1):
            for j in range(0, N+1):
                sonuc_katsayi[i+j] += c[i]*d[j]
        return Polinom(sonuc_katsayi)
```

Polinomlarla Diğer İşlemler
Polinom Türevi

- ✓ Polinom sınıfına ayrıca verilen polinomun aşağıdaki formüle göre türevini alan bir metot daha ekleyebiliriz.

$$\frac{d}{dx}\sum_{i=0}^{n} c_i x^i = \sum_{i=1}^{n} i c_i x^{i-1}$$

- ✓ Bu formüle göre türev almanın iki farklı yolu bulunmaktadır: 1) verilen polinomun katsayı ve üstlerinde değişiklik yaparak türev almak 2) türev polinomu için yeni bir polinom olgusu oluşturmak ve orjinal olguya dokunmadan türev işlemini gerçekleştiren metottan bu olguyu döndürmek. Her iki yöntem için de birer metot aşağıda bulunmaktadır. Birinci metot (p.turev1()) hiçbir şey döndürmeyip p olgusunun katsayıları değiştirilmektedir. İkinci metot (p.turev2()) ise türevin katsayılarına sahip yeni bir Polinom nesnesi döndürmektedir.

```
def turev1(self):
    """Yeni bir nesne dondurmeden gelen polinomun turevini alan metot"""
    for i in range(1, len(self.katsayi)):
        self.katsayi[i-1] = i*self.katsayi[i]
        del self.katsayi[-1]

def turev2(self):
    """Gelen polinomun bir kopyasini alip turevini donduren metot"""
    dpdx = Polinom(self.katsayi[:]) # gelen polinomu kopyala
    dpdx.turev1()
    return dpdx
```

- ✓ Programımızın nasıl çalıştığını göstermek için aşağıdaki iki polinomdan faydalanalım.

$$p_1(x) = 1 - x, \quad p_2(x) = x - 6x^4 - x^5$$

```
>>> from sinif_polinomlar import *
>>> p1 = Polinom([1,-1])
>>> p2 = Polinom([0,1,0,0,-6,-1])
>>> p3 = p1 + p2
>>> print p3.katsayi
[1, 0, 0, 0, -6, -1]
>>> p4 = p1*p2
>>> print p4.katsayi
[ 0.  1. -1.  0. -6.  5.  1.]
>>> p5 = p2.turev2()
>>> print p5.katsayi
[1, 0, 0, -24, -5]
>>> x = 0.5 # Bu nokta icin p1(x) + p2(x) ile p3(x) = (p1+p2)(x) ayni degeri verir mi bakalim
>>> p1_arti_p2_deger = p1(x) + p2(x)
>>> p3_deger = p3(x)
>>> print p1_arti_p2_deger - p3_deger
0.0
```

Polinomlarla Diğer İşlemler
Polinomların Ekrana Yazdırılması

- ✓ Polinom sınıfına ayrıca verilen polinomun ekrana güzel bir şekilde yazdırılacağı bir __str__ metodu da ekleyebiliriz. Basit bir kod aşağıdaki gibi olabilir.

```
class Polinom:
    ...
    def __str__(self):
        s = ''
        for i in range(len(self.katsayi)):
            S += ' +%g*x^%d' % (self.katsayi[i],i)
        return s
```

- ✓ Ancak bu basit yaklaşım matematiksel “şıklık” açısından bakıldığında tatmin edici bir çıktı vermekten birkaç sebeple uzak olacaktır. Örneğin [1,0,0,-1,-6] katsayılar listesi +1*x^0 + 0*x^1 + 0*x^2 + -1*x^3 + -6*x^4 polinomunu ekrana yazdırır. Bu çıktı birkaç nedenle oldukça “çirkindir”. Bundan kaçınmak için 1) 0 katsayılı terimlerin yazdırılmaması gerekir 2) '+' ile '-' yan yana getirilmemelidir. 3) Katsayı 1 olduğunda katsayı yazdırılmamalıdır. 4) Üs bir olduğunda yazdırılmamalıdır. 5) Üs 0 olduğunda x'li terim yazdırılmamalıdır. 6) En başa + işareti gelmemeli, - işareti geldiğinde sayıyla arasında boşluk kalmamalıdır.. Bu şekilde matematiksel olarak daha şık bir yazıma ulaşmak mümkün olur. Örneğimizde x – x^3 – 6x^4 daha “şık” bir gösterimdir.

```
class Polinom:
    ...
    def __str__(self):
        s = ''
        for i in range(0, len(self.katsayi)):
            if self.katsayi[i] != 0:
                s += ' + %g*x^%d' % (self.katsayi[i], i)
        # ciktiyi guzellestir:
        s = s.replace('+ -', '- ')
        s = s.replace('x^0', '1')
        s = s.replace(' 1*', ' ')
        s = s.replace('x^1 ', 'x ')
        s = s.replace('x^1', 'x')
        if s[0:3] == ' + ': # basa gelen + isaretini kaldir
            s = s[3:]
        if s[0:3] == ' - ': # basa gelen - isaretini kaydir
            s = '-' + s[3:]
        return s
```

- Programımızı test edelim.

```
>>> from sinif_polinomlar import *
>>> p1 = Polinom([1,-1])
>>> print p1
1 - x
>>> p2 = Polinom([0, 1, 0, 0, -6, -1])
>>> print p2
x - 6*x^4 - x^5
>>> p2.turev1()
>>> print p2
1 - 24*x^3 - 5*x^4
```

Aritmetik İşlemler ve Diğer Özel Metotlar

Bir sınıfın iki olgusu olan a ve b olguları için standart aritmetik işlemler aşağıdaki özel metotlarla tanımlanır.

- a + b : a.__add__(b)
- a – b : a.__sub__(b)
- a * b : a.__mul__(b)
- a / b : a.__div__(b)
- a ** b : a__pow__(b)

Diğer kullanışlı özel metotlar:

- a olgusunun uzunluğu: len(a) : a.__len__()
- a olgusunun mutlak değeri: abs(a) : a.__abs__()
- a == b : a.__eq__(b)
- a > b : a.__gt__(b)
- a >= b : a.__ge__(b)
- a < b : a.__lt__(b)
- a <= b : a.__le__(b)
- a != b : a.__ne__(b)
- -a : a.__neg__()

Metne Dönüştürme Üzerine Birkaç Not
__repr__ özel metodu

- ✓ Aşağıdaki interaktif Python kabuğunda yazılmış sınıf yapısını inceleyelim.

```
>>> class BenimSinifim:
        def __init__(self):
            self.veri = 2
        def __str__(self):
            return 'In __str__: %s' % (self.veri)

>>> bs = BenimSinifim()
>>> print bs
In __str__: 2
>>> bs
<__main__.BenimSinifim instance at 0xb75125ac>
```

- ✓ İnteraktif bir kabukta sadece olgunun adı yazıldığında, her ne kadar bir `__str__` özel metodu olsa da Python bu kez `__repr__` özel metodunu arar. Bu metot __str__'ye çok benzer ancak olgunun içeriğinin ekrana yazımını sağlayan __str__ 'den farklı olarak olgunun içeriğinin tamamını temsil eder. Pek çok Python nesnesi için (int, float, complex, list, tuple, dict) __repr__ ve __str__ aynı çıktıyı verir.

- ✓ Teknik olarak `str(a)` ifadesi `a.__str__()` metodunu çağırırken, interaktif kabukta sadece a, `a.__repr__()` metodunu çağırır.

```
>>> print a          # print str(a) demektir
>>> a                # repr(a) anlamina gelir
```

- ✓ Bu durumdan kaçınmak için BenimSinifim sınıfına aşağıdaki metodu eklemek yeterli olacaktır.

```
>>> def __repr__(self):
        return self.__str__() # ya da return str(self)
```

- ✓ Daha önce gördüğünüz gibi `eval(e)` fonksiyonu e metnini bir Python ifadesi olarak çalıştırır. __repr__ özel metodunun asıl niteliği eval fonksiyonu uygulandığında Python ifadesi olarak çalıştırılabilecek (aynı olguyu tekrar oluşturabilecek) bir metin döndürmesidir.

- ✓ Örneğin, bu derste dikey atış problemini çözmek üzere tanımladığımız Y sınıfında v0 = 10 için __repr__ metodu 'Y(10)' metnini döndürmelidir ki `eval('Y(10)')` ifadesi `Y(10)` ifadesi ile aynı işi görsün!

- ✓ Aşağıda bu derste tanımladığımız sınıf yapıları için __repr__ özel metodunun nasıl kullanıldığının örnekleri verilmiştir.

```
class Y:
    ...
    def __repr__(self):
        return 'Y(v0=%s)' % self.v0

class Polinom:
    ...
    def __repr__(self):
        return 'Polinom(katsayilar=%s)' % self.katsayi

class BenimSinifim:
    ...
    def __repr__(self):
        return 'BenimSinifim()'
```

- ✓ Bu tanımlarla `eval(repr(x))` ifadesi x nesnesini tekrar yaratabilir. Örneğin x'i bir dosyaya yazıp, dosyadan x'i nesne bilgisiyle birlikte geri çekebiliriz.

```
# dosya bir dosya nesnesi olmak uzere
dosya.write(repr(x))
dosya.close()
...
veri = dosya.readline()
x2 = eval(veri)      # x2, x nesnesinin tekrar yaratilmis halidir x == x2: True
```

Sınıflarla Programlama
Vektörlerle İşlemler (Opsiyonel)

- ✓ İki boyutlu bir düzlemde vektörler (a,b) reel sayı çiftiyle tanımlanırlar.
- ✓ Vektörler üzerine tanımlanan bazı işlemler aşağıdaki gibidir:

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

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

$$(a,b)\cdot(c,d)=ac+bd$$

$$||(a,b)||=\sqrt{(a,b)\cdot(a,b)}$$

Ayrıca, (a,b) = (c,d) → a = b ve c =d

- ✓ Vektor adında bir sınıf oluşturup, yukarıdaki işlemleri de özel metotlardan yararlanarak tanımlayabiliriz. Vektör koordinatlarını göstermek üzere iki adet özniteliğe (attribute) (x,y) ve bir de çıktı veren metoda ihtiyacımız olacak.

```
from math import abs,sqrt
class Vektor:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __add__(self, other):
        return Vektor(self.x + other.x, self.y + other.y)
    def __sub__(self, other):
        return Vektor(self.x - other.x, self.y - other.y)
    def __mul__(self, other):
        return self.x*other.x + self.y*other.y
    def __abs__(self):
        return sqrt(self.x**2 + self.y**2)
    def __eq__(self, other):
        fark = 1e-16 # iki reel sayiyi == operatoruyle karsilastirmak risklidir!
        return abs(self.x - other.x) <= fark and abs(self.y - other.y) <= fark
    def __str__(self):
        return '(%g, %g)' % (self.x, self.y)
    def __ne__(self, other):
        return not self.__eq__(other)
```

- ✓ Şimdi yarattığımız nesnelerle biraz “oynayalım” (sinif_vektorler.py).

```
>>> from sinif_vektorler import *
>>> u = Vektor(0,1)
>>> v = Vektor(1,0)
>>> w = Vektor(1,1)
>>> a = u + v
>>> print a
(1, 1)
>>> a == w
True
>>> a = u - v
>>> print a
(-1, 1)
>>> a = u*v
>>> print a
0
>>> print abs(u)
1.0
```

Statik Metot ve Öznitelikler

- ✓ Şu ana kadar her bir olgunun (instance) kendi özniteliklerine (attribute) sahip olduğunu gördük. Bazen aynı sınıfın farklı olguları arasında paylaşılan özniteliklere de ihtiyaç duyulur. Örneğin bir sınıftan kaç tane olgu üretildiğini tutan bir öznitelik (sınıf değişkenlerine öznitelik –attribute- diyoruz) faydalı olur. Bunun için özniteliği sınıfın metotlarıyla aynı hizadan (indentation level) başlatmak ve self öneki (prefix) yerine sınıfın adını önek olarak kullanmak yeterlidir. Bu şekilde aynı sınıfın tüm olguları tarafından erişlebilen özniteliklere statik öznitelikler adı verilir.

```
>>> class UzaydaNokta:
        sayac = 0
        def __init__(self,x,y,z):
            self.nokta = (x,y,z)
            UzaydaNokta.sayac += 1

>>> p1 = UzaydaNokta(0,0,0)
>>> UzaydaNokta.sayac
1
>>> for i in range(400):
        p = UzaydaNokta(i*0.5,i,i+1)

>>> UzaydaNokta.sayac
401
```

- ✓ Şu ana kadar gördüğümüz tüm sınıf metotları da bir olgu tarafından “çağrılıyor” ve self değişkeniyle “besleniyorlardı”. Herhangi bir olguya bağlı olmaksızın çalışan metotlar yaratmak da mümkündür. Bu durumda metot, bir sınıf yapısı içinde yer alması ve bu nedenle o sınıfın adının önek (prefix) olarak verilmesi gerekliliği dışında tipik bir Python fonksiyonu gibi davranır. Bu tür metotlar statik metotlar adı verilir.

```
>>> class A:
        @staticmethod
        def mesajyaz(mesaj):
            print mesaj

>>> A.mesajyaz('Merhaba Dunyali Biz Tostuz!')
Merhaba Dunyali Biz Tostuz!
>>> a = A()           # istenirse bir olgu da bu fonksiyonu kullanabilir!
>>> a.mesajyaz('Tost degil salak Dost!')
Tost degil salak Dost!
```

Dersi Özetleyen Örnek: Aralık Aritmetiği

- ✓ Astronomide pek çok formülün girdisi, ölçüm hataları gibi nedenlerle bir belirsizliğe sahiptir. Böyle durumlarda bir girdi parametresini tek bir değerle belirlemek yerine bu parametrenin değerinin içinde bulunduğu garanti edilen bir aralıkla belirlemek daha iyi bir yoldur. Aralığın büyüklüğü parametre üzerindeki belirsizliğin büyüklüğüne bağlıdır.

- ✓ Diyelim ki bir formülün tüm girdi parametreleri belirsizliklere sahip ve bu nedenle aralıklarla tanımlanıyorlar. Bu durumda formülün çıktısı nasıl bir belirsizliğe sahip olur ya da hangi aralık dahilinde yer alır.

- ✓ Örneğin, bir yerde (örnek olarak Mars yüzeyinde) yüzey çekim ivmesini hesaplamak için bir cismi serbest düşmeye bırakıyoruz ve yere ulaştığı (y = 0) zamanı (T) ölçerek buradan yüzey çekim ivmesine geçimek istyoruz.

$$y(t) = y_0 - \frac{1}{2}gt^2 \qquad g = 2y_0T^{-2}$$

- ✓ Böyle bir deneyde başlangıç konumunu (y_0) ve toplam süreyi (T) belirlerken belirsizliklere muhatap oluruz, zira ölçümlerimiz aletlerimizin hassasiyeti ile sınırlıdır.

- ✓ Diyelim ki y ∈ [0.99, 1.01] ve T ∈ [0.43, 0.47] olsun. Bir başka deyişle, ölçümlerimiz üzerinde konumda %2, zamanda %10 gibi bir belirsizlik bulunoyor olsun. Bu durumda g (yerçekimi ivmesi) üzerindeki hata ne olur?

- ✓ **Problem:** p ∈ [a,b] ve q ∈ [c,d] olsun. Öyle ise p + q, s = a + c, t = b + d olmak üzere [s,t] aralığının içinde yer alıır. Diğer aralık aritmetiği kuralları şu şekilde listelenebilir:

  1) (p + q) ∈ [a+c,b+d]

  2) (p - q) ∈ [a-d,b+c]

  3) pq = [min(ac,ad,bc,bd),max(ac,ad,bc,bd)]

  4) p/q = [min(a/c, a/d, b/c, b/d),max(a/c, a/d, b/c, b/d)]; Not: [c,d] aralığı 0 içermemelidir.

- ✓ Böyle bir problemin en iyi çözümüne yeni bir veri türü tanımlamakla ulaşılabilir. Bu veri türü üzerinde yukarıdaki kurallar dahilinde + / - / * / / işlemleri tanımlanmalıdır. Dolayısı ile çözüm, listelenen bu kuralların özel metotlar yoluyla uygulandığı yeni bir tür olarak bir aralık aritmetiği sınıfı yaratmaktan geçer. Bu sınıfın doğal öznitelikleri aralığın üst ve alt limitleridir. Matematiksel işlemlerin yanı sıra bir de çıktı üreten metot olması genel uygulamadır.

```
class AralikAritmetigi:
    def __init__(self, altlim, ustlim):
        self.al = float(altlim)
        self.ul = float(ustlim)
    def __add__(self, other):
        a, b, c, d = self.al, self.ul, other.al, other.ul
        return AralikAritmetigi(a + c, b + d)
    def __sub__(self, other):
        a, b, c, d = self.al, self.ul, other.al, other.ul
        return AralikAritmetigi(a - d, b - c)
def __mul__(self, other):
    a, b, c, d = self.al, self.ul, other.al, other.ul
    return AralikAritmetigi(min(a*c, a*d, b*c, b*d),max(a*c, a*d, b*c, b*d))
def __div__(self, other):
    a, b, c, d = self.al, self.ul, other.al, other.ul
    # [c,d] 0 (sifir) iceremez!
    if c*d <= 0:
        raise ValueError\
        ('Aralik %s seklinde verilemez, cunku 0 icermektedir')
    return AralikAritmetigi(min(a/c, a/d, b/c, b/d),max(a/c, a/d, b/c, b/d))
def __str__(self):
        return '[%g, %g]' % (self.al, self.ul)
```

- Kodumuzun nasıl çalıştığına bakalım.

```
>>> from sinif_aralikaritmetigi import *
>>> arlk = AralikAritmetigi
>>> a = arlk(-3,-2)
>>> b = arlk(4,5)
>>> islem = 'a+b', 'a-b', 'a*b', 'a/b'
>>> for I in islem:
        print '%s = ' % I, eval(I)

a+b =  [1, 3]
a-b =  [-8, -6]
a*b =  [-15, -8]
a/b =  [-0.75, -0.4]
```

- Küçük bir kod parçasıyla ne kadar “muhteşem” bir iş yapmış olduğunu düşünüyor olabilirsiniz. Ancak kodumuzun aslında pek çok eksiği var. Örneğin, parametrelerden biri üzerinde belirsizlik yoksa, yani bir aralık değil de reel sayı geliyorsa kodumuz nasıl davranacak?

```
>>> a = arlk(4,5)
>>> q = 2
>>> b = a*q
File "sinif_aralikartimetigi.py", line 12, in __mul__
a, b, c, d = self.al, self.ul, other.al, other.us
AttributeError: ’float’ object has no attribute ’ul’
```

- Gördüğünüz gibi işler pek de yolunda gitmedi! a parametresi için bir sorun yok ancak q parametresi bir aralık değil, dolayısıyla bir alt limiti (Python önce bu hatayla karşılaştığı için size hata mesajında bunu söylüyor) ve bir üst limiti yok! Bunun için ilgili metotlara şartlı bir ifade koyup gelen sayının reel olup olmadığını denetlyebilir ve kdounuzun buna göre davranmasını sağlayabilirsiniz. Ancak aralık yerine sayı şeklinde gelen parametreler için ayrı işlemler (metotlar) tanımlamak ve gerektiğinde bu metotları çağırmak daha iyi bir fikir.

- ✓ Reel sayıyla çarpma için __rmul__ özel metodundan faydalanabiliriz. Aşağıdaki Python kodu bu metot için olmakla birlikte diğer işlemlere (toplama, çıkarma, çarpma) kolaylıkla uyarlanabilir.

```
def __rmul__(self, other):
        if isinstance(other, (int, float)):
            other = AralikAritmetigi(other, other)
        return other*self
```

- ✓ Bir diğer problemimiz de üs alma. Örneğin dikey atış probleminde yerçekimi ivmesini bulmak için ($g = 2y_0T^{-2}$) üs almaya (üstelik de negatif!) ihtiyacımız olacak! Bunun için aşağıdaki kodu da sınıfımıza eklemeliyiz.

```
def __pow__(self, us):
        if isinstance(-, int):
            p = 1
            if us > 0:
                for i in range(us):
                    p = p*self
            elif us < 0:
                for i in range(-us):
                    p = p*self
                    p = 1/p
            else: # us == 0
                p = AralikAritmetigi(1, 1)
        return p
        else:
            raise TypeError('us tam sayi olmalidir')
```

- ✓ Kodumuza bir de herhangi bir aralığın orta noktasını kullanarak onu bir reel sayıya dönüştürebilen bir metot eklemek faydalı olacaktır.

```
def __float__(self):
        return 0.5*(self.al + self.ul)
```

- ✓ Olguları doğru yazım kurallarıyla (syntax) tekrar oluşturabilmek için gerekli __repr__ metodu da neredeyse her sınıfta yerini alır.

```
def __repr__(self):
        return '%s(%g, %g)' % (self.__class__.__name__, self.al, self.ul)
```

- ✓ Artık kodumuzu bütünsel olarak test edebliriz (sinif_aralikaritmetigi.py)

Test 1: Dikey atış problemi

```
>>> from sinif_aralikaritmetigi import *
>>> arlk = AralikAritmetigi
>>> g = 9.81
>>> y_0 = arlk(0.99,1.01)
>>> Tm = 0.45
>>> print T
[0.4275, 0.4725]
>>> g = 2*y_0*T**(-2)
>>> g
AralikAritmetigi(8.86873, 11.053)
>>> print g
[8.86873, 11.053]
>>> # Simdi hesabimizi reel sayilarla yapalim
>>> T = float(T)
>>> y = 1
>>> g = 2*y*T**(-2)
>>> print '%.2f' % g
9.88
```

Test 2: Küre Hacmi

```
>>> from sinif_aralikaritmetigi import *
>>> from math import pi
>>> arlk = AralikAritmetigi
>>> Rm = 6
>>> R = arlk(0.9*R,1.1*R)  # %10 hata
>>> V = (4./3)*pi*R**3
>>> V
AralikAritmetigi(659.584, 1204.26)
>>> print V
[659.584, 1204.26]
>>> print float(V)
931.922044761
>>> # Simdi hesabimizi reel sayilarla yapalim
>>> R = float(R)
>>> V = (4./3)*pi*R**3
>>> print V
904.778684234
```