* Nesne tabanlı programlama bir programlama yaklaşımı, disiplinidir.
* Bu yaklaşıma göre programcı sadece programı gerçekleştiren işlevlere odaklanmaz, bu görevleri gerçekleştirmede kullanabileceği nesnelere de odaklanır.
* Python’un kurucusu Guido Van Rossum bu dili “First-class everything” ilkesine göre oluşturdu.
* %100 nesne tabanlı bir dil olan pythonda sınıflar dahil her şey (değişken, liste, sınıf vb.) bir nesne olarak algılanır.
* Pythonda kendisine isim verilebilen bütün nesneler (tam sayılar, karakter dizileri, listeler, fonksiyonlar, sınıflar, modüller, metotlar vb.) eşit statüye sahiptir.
* Yani bütün nesneler değişkenlere atanabilir, bir liste/demet/sözlük içine yerleştirilebilir/depolanabilir, parametre olarak atanabilir, vb.
* Nesne yönelimli programlamanın temelini, sınıf (İngilizcesi: **class**) ve bu sınıfların örneklendirilmiş hali olan nesneler/objeler (İngilizcesi: **object**) oluşturur.
* **Sınıf, ortak özellikleri ve davranışları olan nesnelerin**, bu ortak özellik ve davranışlarını tanımlayan **soyut bir kavramdır.**
* Python dilini kod fabrikası gibi düşünürsek; bu fabrikada üretilen her bir şeye **nesne**, bu nesnelerin nasıl üretileceğini tarif eden tanımlar, kurallar/protokoller dizisine ise **sınıf (class)** denir.
* **Örneğin;** arsanızın üzerine ev yapmak istiyorsunuz. Öncelikle ilk iş bir mimara evin tasarımını (otopark, bahçe, kat, oda, mutfak, banyo vb.) çizdirmektir. Sonra da bir ustaya çizdirilen evin somut bir örneğini yaptırmaktır. İstenirse daha sonra bu tasarımdan benzer başka evler de(siteler de) yaptırılabilir. Bu evlerde aynı yapıya/tasarıma sahip olacaktır ama her evin boyası, badanası, mobilyası farklılıklar arz edebilir.
* Burada; mimarın taslak çizimine **sınıf(class),** ustaya/inşaat firmasına yaptırılan eve ise **nesne(object**) adı verilir.
* Anlaşılacağı üzere bir **sınıf tasarımdan/şablondan başka bir şey değildir.** Somut olarak kullanılamaz ancak bir **örneği oluşturulduğunda**(bu işleme nesne diyoruz) kullanılabilir.
* Sınıf kendisinden nesne türetilen yapay bir yapıdır. Bütün nesneler sınıfı temel alarak üretilirler. Bir sınıftan birden fazla nesne üretilebilir. Bir sınıftan nesne oluşturma işlemine **örnekleme (instantation)** adı verilir.
* Yapılandırıcı (**init**) metot da dahil olmak üzere tüm metotlar ilk parametre olarak nesnenin kendisine işaret eden **self** parametresini alırlar.
* 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.

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.

**Python’da Sınıf Tanımı**

Bir Robot sınıfı oluşturalım. En basit sınıf şu şekildedir :

**class Robot:**

**pass**

Yapısı şöyledir: İlk satır, kafa adını verdiğimiz kısımdan oluşur. **class** boşluk **sınıf ismi** ve **iki nokta** üst üste şeklindedir. İkinci kısım, beden adını verdiğimiz kısımdan oluşur. Bu kısımda yaptılırmasını istediğimiz işlemler yer alır.

**Sınıf tanımı ve sınıftan nesne üretmek için kullanılan yapılar**

* class SınıfAdı:

<işlemler 1>

<işlemler 2>

* class SınıfAdı():

<işlemler 1>

<işlemler 2>

……

Nesne\_adı=SınıfAdı([parametreler]) # Örnek kullanım

Robot sınıfından örnekler oluşturalım:

class Robot:  
 pass  
if \_\_name\_\_=="\_\_main\_\_":  
 x=Robot()  
 y=Robot()  
 y2=y  
 print(y==y2)  
 print(y==x)

True

False

ÖZNİTELİKLER

Öznitelik dediğimiz kavram, bir nesnenin sahip olduğu niteliklerdir: Örneğin özel bir yeteneği; piyano çalabilmek, basketbol oynamak ya da saç rengi; siyah saçlı, sarışın, kel ya da hızlı kitap okumak.

Robot sınıfımız henüz bir özniteliğe sahip değil. “name” ve “build\_year” niteliği ekleyelim.

class Robot:  
 pass  
x=Robot()  
y=Robot()  
x.name="Marvin"  
x.build\_year="1979"  
y.name="Caliban"  
y.build\_year="1993"  
print(x.name)  
print(y.build\_year)

{key:value, key1:value1,…}

Marvin

1993

Bu yöntem, örnek öznitelikleri oluşturmak için doğru bir yöntem değil. Konunun daha iyi anlaşılması için bu yöntemle başlanıldı.

* Örnek oluşturulurken, içte tüm öznitelikler sözlük yapısı altında saklanır.

print(x.\_\_dict\_\_)

{'name': 'Marvin', 'build\_year': '1979'}

print(y.\_\_dict\_\_)

{'name': 'Caliban', 'build\_year': '1993'}

* Örnek, sınıf parametre olarak kullanılabilir ve öznitelikler sınıf isimlerine atanabilir.

class Robot(object):  
 pass  
x=Robot()  
Robot.brand="Kuka"  
print(x.brand)

Kuka

* Örneğe bir isim atarsak eğer;

x.brand="Thales"  
print(Robot.brand)

Kuka

* Farklı bir örnek değişkenini yazdırırsak;

y=Robot()  
print(y.brand)

Kuka

* Robot sınıfının niteliğini değiştirisek;

Robot.brand="Thales"  
print(y.brand)

Thales

Eğer **\_\_dict\_\_** kullanırsak, arka planda neler olduğunu anlarız.

print(x.\_\_dict\_\_)

{'brand': 'Thales'}

print(y.\_\_dict\_\_)

{}

print(Robot.\_\_dict\_\_)

{'\_\_module\_\_': '\_\_main\_\_', '\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'Robot' objects>, '\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'Robot' objects>, '\_\_doc\_\_': None, 'brand': 'Thales'}

* Bir öznitelik sözlük içerisinde yer almıyor ise nitelik ismi tanımlanmaz ve AttributeError ile karşılaşırsınız.

print(x.energy)

AttributeError: 'Robot' object has no attribute 'energy'

* Burada **getattr** fonksiyonu ile bu duruma engel olabilirsiniz.

print(getattr(x,'energy',100))

100

* Bu şekilde fonksiyonlara da öznitelikler ekleyebiliriz.

Boşşş

**METOTLAR**

Fonksiyonlar sınıf yapısı içerisine girince metot olarak isimlendirilirler. Bir fonksiyon metottan iki açıdan ayrılmaktadır;

1.Metot sınıfa aittir ve sınıf içerisinde tanımlanır.

2.Metod’un ilk parametresi örnek (nesne)’i referans gösterir ve genelde ‘**self**’ kelimesidir.

* ‘hi’ fonksiyonu tanımlayalım. Bu fonksiyon ‘obj’ nesnesini değişken olarak alsın ve bu ‘obj’ nesnesi ‘name ’ isminde değişkene sahip olsun.

def hi(obj):  
 print("Hi I am " + obj.name + "!")  
#sınıf  
class Robot():  
 pass  
x=Robot()  
x.name="Marvin"  
hi(x)

Hi I am Marvin!

* Şimdi de ‘hi’ fonksiyonun sınıf özniteliği, ‘say\_hi’ a bağlayalım:

def hi(obj):  
 print("Hi I am " + obj.name + "!")  
class Robot:  
 say\_hi=hi  
x=Robot()  
x.name="Marvin"  
Robot.say\_hi(x)

Hi I am Marvin!

* Burada say\_hi() bir metottur. Genellikle; x.say\_hi() şeklinde çağırılır.
* Gördüğünüz gibi Robot.say\_hi(x) ile x.say\_hi() aynıdır.
* Bu kodtaki sıkıntımız, sınıf oluştrduktan sonra özniteliğe (name burada) değer atamayı unutursak kod hata verecektir.
* Nesneyi oluşturur oluşturmaz çalıştıracak(başlatacak) bir mekanizmaya ihtiyaç vardır:

**\_\_init\_\_** metodudur.

* **\_\_init\_\_** metodu örnek oluşturulur oluşturulmaz çağırılan bir metottur.
* Sınıf yapısı içerisinde herhangi bir yerde olabilir fakat genellikler ilk metottur**.**
* Genel çağırılışı şu şekildedir:

class A:  
 def \_\_init\_\_(self):  
 print("\_\_init\_\_ has been executed !")  
x=A()

\_\_init\_\_ has been executed !

* Robot sınıfımıza \_\_init\_\_ metodunu ekleyelim:

class Robot():  
 def \_\_init\_\_(self,name=None):  
 self.name=name  
 def say\_hi(self):  
 if self.name:  
 print("Hi, I am " + self.name)  
 else:  
 print("I am a robot without a name")  
x=Robot()  
x.say\_hi()  
y=Robot("Marvin")  
y.say\_hi()

I am a robot without a name

Hi, I am Marvin

DATA ABSTRACTION, DATA ENCAPSULATION, AND INFORMATION HIDING  
**Kapsülleme:** Verilerin bu veriler üzerinde çalışan yöntemlerle paketlenmesi olarak görülür. Veri üzerinde inşaa yapan metotolarla verinin sarmalanmasıdır. Genelde iki tür metot ile gerçekleşir bu işlem: Getter ve Setter metotları

* Getter metodu, öznitelikte değişiklik yapılmasına izin vermez, sadece değer döndürür.
* Setter metodu, metot aracılı ile nesnenin özniteliklerini değiştirmemize imkan sağlıyor.

Veri saklaması: Verinin hiçbir şekilde değiştirilememesidir.

* Daha önce tanımladığımız Robot sınıfına, ‘**name**’ özniteliğine erişim için ‘**getter’** ve ‘**setter**’ fonksiyonları tanımlayalım.

class Robot():  
 def \_\_init\_\_(self,name=None):  
 self.name=name  
 def say\_hi(self):  
 if self.name:  
 print("Hi, I am " + self.name)  
 else:  
 print("I am a robot without a name")  
 def set\_name(self,name):  
 self.name=name  
 def get\_name(self):  
 return self.name  
x=Robot()  
x.set\_name('Henry')  
x.say\_hi()  
y=Robot()  
y.set\_name(x.get\_name())  
print(y.get\_name())

* Burada, setter metodu ile nesne özniteliğine değer atamayı ve getter metodu ile de nesne özniteliğine erişmeyi öğrendik.
* Şimdi de Robot sınıfımıza ‘**build\_year’** isimli bir özniteliği setter ve getter metotları aracılı ile ekleyelim.
* Robot sınıfının iki şekilde ‘**name**’ ve **‘build\_year’** özniteliklerini değiştirmemize ve özniteliklere erişmemize imkan sağlaniyor.

class Robot():  
 def \_\_init\_\_(self,name=None,build\_yr=1993):  
 self.name=name  
 self.build\_yr=build\_yr  
 def say\_hi(self):  
 if self.name:  
 print("Hi, I am " + self.name)  
 else:  
 print("I am a robot without a name")  
 def set\_name(self,name):  
 self.name=name  
 def get\_name(self):  
 return self.name  
 def set\_build\_yr(self,build\_yr):  
 self.build\_yr=build\_yr  
 def get\_build\_yr(self):  
 return self.build\_yr  
x=Robot("Marvin",1993)  
y=Robot("Jesus",2020)

**\_\_STR\_\_ ve \_\_REPR\_\_ METOTLARI**

Bir nesneyi yazdırılabilir hale getirmek için kullanılabilecek iki metot bulunur.

repr(x), x.\_\_repr\_\_()‘i çağırır: x.eval genellikle bu metotun sonucunu orijinal nesneye geri dönüştürür.

str(x), x.\_\_str\_\_()‘i çağırır: nesneyi tanımlayan okunabilir bir dizeye dönüştürür. Böylece bazı teknik ayrıntılara gerek olmayabilir.

**Python’da repr() nedir?**

Birçok tür için bu işlev, iletildiğinde aynı değere sahip bir nesne verecek bir dizge döndürme girişiminde bulunur eval(). Aksi takdirde, temsil, ek bilgilerle birlikte nesnenin türünün adını içeren, açılı parantez içine alınmış bir dizedir. Bu genellikle nesnenin adını ve adresini içerir.

**Python’da str() ne işe yarar?**

Dizeler için bu fonksiyon dizenin kendisini döndürür. repr(object) ile str(object) arasındaki fark str(object) her zaman eval() için kabul edilebilir bir dizge döndürmeye çalışmaz. Daha ziyade, yazdırılabilir veya ‘okunabilir’ bir dize döndürür. Hiçbir bağımsız değişken belirtilmezse boş bir dizeyi döndürür '.

* Eğer bir nesneye **repr** ya da **str** uygulanırsa, nesnenin sınıf tanımı içerisinde **\_\_str\_\_** ve **\_\_repr\_\_** metotlarını arar.
* Aşağıdaki örnekte doğrudan nesneye repr ve str uygularsak ne olur görelim:

class A:  
 pass  
a=A()  
print(a)  
print(repr(a))  
print(str(a))

<\_\_main\_\_.A object at 0x000001E782B3CF40>

<\_\_main\_\_.A object at 0x000001E782B3CF40>

<\_\_main\_\_.A object at 0x000001E782B3CF40>

* Gördüğümüz gibi her iki metotta işe yaramaz.
* Eğer sınıf içerisinde \_\_str\_\_ metodu tanımlanırsa str metodu sınıf nesnesini ifade etmek için kullanılabilir.

class A:  
 def \_\_str\_\_(self):  
 return "42"  
a=A()  
print(str(a))

42

print(repr(a))

<\_\_main\_\_.A object at 0x0000022734BDCF40>

* Diğer taraftan bir sınıf sadece \_\_repr\_\_ metoduna sahip ve \_\_str\_\_ metoduna sahip değilse, \_\_rep\_\_ metodu \_\_str\_\_ nin de uygulanabileceği durumlarda uygulanır.

class A:  
 def \_\_repr\_\_(self):  
 return "42"  
a=A()  
print(repr(a))  
print(str(a))

42

42

* Burada aklımıza gelen soru hangi metodun tercih edilmesi gerektiğidir. Aşağıda datatime modülüne uygulanan örnekte, eval komutu kullanılacağı durumlarda, repr tarafından oluşturulan metinlerin tercih edilmesi gerektiği söylenilebilir.

import datetime  
today=datetime.datetime.now()  
str\_s=str(today)  
eval(str\_s)

Traceback (most recent call last):

File "C:\Users\nurca\Desktop\NESNE TABANLI YBS 212\Hafta\_2\Week 1 Codes.py", line 142, in <module>

eval(str(today))

File "<string>", line 1

2022-03-13 17:15:57.110477

^

SyntaxError:

* Bu sefer eval() fonksiyonu içerisinde repr() komutunu kullanalım. Bakalım hata verecek mi?

import datetime  
today=datetime.datetime.now()  
repr\_s=repr(today)  
t=eval(repr\_s)  
print(type(t))

<class 'datetime.datetime'>

* eval(repr\_s) 'datetime.datetime' nesnesine dönüştürür.

**ERİŞİM BELİRLEYİCİLER**

**ALENİ, YARI GİZLİ VE GİZLİ ÖZNİTELİKLER**

**(PUBLIC, PROTECTED, AND PRIVATE ATTRIBUTES)**

**public:** Public olarak tanımlanan üye tüm erişimlere açıktır. **(name)**

**protected:** Kendi üyesi olduğu (tanımlandığı) sınıflardan ve bu sınıftan türetilmiş sınıflardan erişilir. **(\_name)**

**private:** Yalnızca üyesi olduğu sınıftan erişilir. **(\_\_name)**

* Şimdi bu özniteliklerin davranışlarını bir örnekle kontrol edelim.

class A:  
 def \_\_init\_\_(self):  
 self.\_\_priv="I am private"  
 self.\_prot="I am protected"  
 self.pub="I am public"

* Bu sınıf **erisim1.py** dosyasına kaydettik. Aşağıdaki şekilde çağırıyoruz.

from erisim1 import A  
x=A()  
print(x.pub)

I am public

x.pub=x.pub+" and my value can be chanced."

I am public and my value can be chanced.

print(x.\_prot)

I am protected

print(x.\_priv)

Traceback (most recent call last):

File "C:\Users\nurca\Desktop\NESNE TABANLI YBS 212\Hafta\_2\Week 1 Codes.py", line 157, in <module>

print(x.\_\_priv)

AttributeError: 'A' object has no attribute '\_\_priv'

* Normalde burada ‘**I am private’** yazmasını bekliyorduk ama burada harika bir bilgi saklama mevcut.
* Kapsülleme ile gizli özniteliklere, getter ve setter fonksiyonları aracılığıyla ulaşabiliyoruz.
* Şimdi Robot sınıfımızı yeniden düzenleyelim.
* Self.name ve self.build\_year değişkenlerini self.\_\_name ve self.\_\_build\_year değişkenleri ile değiştirelim.
* class Robot():  
   def \_\_init\_\_(self,name=None,build\_yr=None,timer=0):  
   self.\_\_name=name  
   self.\_\_build\_yr=build\_yr  
   def say\_hi(self):  
   if self.\_\_name:  
   print("Hi, I am " + self.\_\_name)  
   else:  
   print("I am a robot without a name")  
   def set\_name(self,name):  
   self.\_\_name=name  
   def get\_name(self):  
   return self.\_\_name  
   def set\_build\_yr(self,build\_yr):  
   self.\_\_build\_yr=build\_yr  
   def get\_build\_yr(self):  
   return self.\_\_build\_yr  
   def set\_timer(self,timer):  
   self.timer=timer  
   def get\_timer(self):  
   return self.timer  
   def \_\_repr\_\_(self):  
   return "R("+ self.\_\_name+ "," + self.\_\_build\_yr +")"  
   def \_\_str\_\_(self):  
   return "Name: " +self.\_\_name + "build\_yr: " +self.\_\_build\_yr  
    
  x=Robot("Marvin",1990)  
  y=Robot("Caliban",1983)  
  print(x.name)

Traceback (most recent call last):

File "C:\Users\nurca\Desktop\YBS212\_HAFTA3\YBS212\_HAFTA2\_sınıf ve nesne nitelikleri.py", line 46, in <module>

print(x.name)

AttributeError: 'Robot' object has no attribute 'name'

print(x.get\_name())

Marvin

* Gördüğümüz gibi private özniteliğe **nesne.nitelik\_adı** ile ulaşamıyoruz. **nesne.getter\_method()** ile ulaşabiliyoruz.
* Kapülleme aracılığıyla getter ve setter fonksiyonları ile private özniteliklere erişim sağlayabildik.

YIKICILAR (DESTRUCTOR)

Yıkıcı metotlar hafıza yönetimi için büyük önem taşırlar. Yıkıcıların amacı kaynakları serbest bırakmaktır. \_\_del\_\_ metodu yıkıcı metottur.

* Aşağıdaki kodda \_\_init\_\_ ve \_\_del\_\_ metotlarınının kullanım örneği var.

class Robot():  
 def \_\_init\_\_(self,name):  
 print(name + " has been created.")  
 def \_\_del\_\_(self):  
 print("Robot has been destroyed.")

from test\_dosyası import Robot  
x=Robot("mali")  
del x

mali has been created.

Robot has been destroyed.

Örneğin; bir sınıf için yıkıcı (destructor) metot yazdığımızda silinen her öğrenci için öğrenci sayımızın azalması gerekir. Buna uygun bir sınıf tanımlayalım ve \_\_del\_\_ metodunu içinde barındırsın.

class Student():  
 number\_of\_students = 0  
 def \_\_init\_\_(self, name, index):  
 self.name = name  
 self.index = index  
 Student.number\_of\_students += 1  
 def \_\_del\_\_(self):  
 Student.number\_of\_students -= 1  
s1=Student("ali",23456)  
s2=Student("nurcan",12833)  
print(Student.number\_of\_students)  
del s1  
print(Student.number\_of\_students)

2

1

print(s1.name)

Traceback (most recent call last):

File "C:\Users\nurca\Desktop\YBS212\_HAFTA3\test\_dosyası.py", line 30, in <module>

print(s1.name)

NameError: name 's1' is not defined

SINIF VE ÖRNEK ÖZNİTELİĞİ (CLASS AND INSTANCE ATTRIBUTES)

* Örnek öznitelikleri, tanımlanacağı sınıfın spesifik örneği(nesnesi) tarafından oluşturulur. İki farklı nesne için örnek öznitelikleri genelde farklıdır.
* Sınıfın öznitelikleri, sınıfın kendisine ait olan özellikleridir. Sınıfın tüm örnekleri tarafından kullanılabilir ve bu yüzden tüm örnekler için aynıdır.

class A:  
 a="I am a class attribute !"  
x=A()  
y=A()  
print(y.a)

I am a class attribute !

print(A.a)

I am a class attribute !

* Sınıf özniteliklerine hem nesne hem de sınıf aracılığı ile erişilebilir.
* Eğer sınıf özelliklerini değiştirmek istiyorsan “Sınıf\_Adı.Öznitelik\_adı”(A.a) yöntemini kullanmalısınız.
* Pythonda sınıf ve nesne öznitelikleri ayrı sözlük yapılarında saklanır.

ÖRNEK. Sınıf özniteliğini kullanarak, oluşturulan nesne (örnek) sayısını toplatan, nesne silindiğinde de toplan nesne sayısından çıkaran nesne tabanlı programlama kodunu yazınız.

STATIC METHODS

Sınıf özniteliklerini ulaşılabilir öznitelikler olarak tanımladık. Bunları gizli yapmamız mümkün. Gizli olmaları durumunda bile sınıf metotlarıma erişmek mümkün.

1. Örnek metotları aracığı ile erişim sağlayabiliriz.

class Robot:  
 \_\_counter=0  
 def \_\_init\_\_(self):  
 Robot.\_\_counter+=1  
 #type(self).counter+=1  
 def Robotinstances(self):  
 return Robot.\_\_counter  
x=Robot()  
print(x.Robotinstances())  
y=Robot()  
print(x.Robotinstances())

1

2

* Bu yöntem tercih edilen bir yöntem değil ;çünkü örnek oluşturmadan robot sayısına ulaşmak mümkün değil. Robot.Robotinstances() şeklide sınıf aracılığı tanımlamada sonuç alamayız, hata verir.

2.Eğer örnek metodundaki “**self**” parametresini çıkarırsak, Robotinstances() metoduna “sınıf adı” aracılığı ile ulaşmak mümkün

class Robot:  
 \_\_counter=0  
 def \_\_init\_\_(self):  
 Robot.\_\_counter+=1  
 #type(self).counter+=1  
 def Robotinstances():  
 return Robot.\_\_counter  
x=Robot()  
print(Robot.Robotinstances())

1

print(x.Robotinstances())

print(x.Robotinstances())

TypeError: Robotinstances() takes 0 positional arguments but 1 was given

* x.Robotinstances() çağrısı, örnek metodu gibi davranır ve örnek metodunun “self” parametresine referans etmesi gerekir ve burada “self” olmadığı için HATA verir.
* Biz hem sınıf ismi hem de örnek ismi aracılığı ile “self” parametresine gerek kalmadan çağırabileceğimiz bir metot istiyoruz. Çözüm “statik metoda” dayanır.
* Yapmamız gereken statik metot yapmak istediğimiz metodun bir üst satırına @staticmethod yazısını eklemek ve referans parametresi “self” i kaldırmak.

class Robot:  
 \_\_counter=0  
 def \_\_init\_\_(self):  
 Robot.\_\_counter+=1  
 @staticmethod  
 def Robotinstances():  
 return Robot.\_\_counter  
print(Robot.Robotinstances())  
x=Robot()  
print(x.Robotinstances())  
print(Robot.Robotinstances())

0

1

1

* Burada @staticmethod bezeyicisi aracılığıyla Robotinstances() metoduna hem sınıf ismi hem de örnek ismi aracılığı ile erişim sağladık.

SINIF METOTLARI(CLASS METHODS)

* Sınıf metotları statik metotlar ile karıştırılmamalıdır. Sınıf metotları, statik metotlar gibi örneklerle sınırlı değildir. Sınıf metotları, sınıflarla sınırlıdır.
* Sınıf metodunun ilk parametresine referans eden bir sınıf nesnesidir. Genelde “**cls**” kullanılır.
* Sınıf metotlarına örnek ya da sınıf ismi aracılığı ile erişilebilir.@classmethod

class Robot:  
 \_\_counter=0  
 def \_\_init\_\_(self):  
 Robot.\_\_counter+=1  
 @classmethod  
 def Robotinstances(cls):  
 return cls, Robot.\_\_counter  
print(Robot.Robotinstances())  
x=Robot()  
print(x.Robotinstances())  
y=Robot()  
print(x.Robotinstances())  
print(Robot.Robotinstances())

(<class '\_\_main\_\_.Robot'>, 0)

(<class '\_\_main\_\_.Robot'>, 1)

(<class '\_\_main\_\_.Robot'>, 2)

(<class '\_\_main\_\_.Robot'>, 2)

ÖRNEK. Kesirli bir ifadeyi basit kesire indirgeyen **“fraction**” sınıfı oluşturalım.

* Önce iki sayının en büyük ortak bölenini bulduran statik metot oluşturalım. (Örn. gcd(15,10)=5)
* Oluşturduğumuz **gcd()** metodundan değer alıp, kesirli ifadenin iki terimini de bu değere böldüren **“reduce()”** sınıf metodu oluşturalım.

class fraction(object):  
 def \_\_init\_\_(self,n,d):  
 self.numerator, self.denominator=fraction.reduce(n,d)  
 @staticmethod  
 def gcd(a,b):  
 while b!=0:  
 a,b=b,a%b  
 return a  
 @classmethod  
 def reduce(cls,n1,n2):  
 g=cls.gcd(n1,n2)  
 return (n1//g,n2//g)  
 def \_\_str\_\_(self):  
 return str(self.numerator)+'/'+str(self.denominator)  
x=fraction(8,24)  
print(x)

SINIF METOTLARI, STATİK METOTLAR VE ÖRNEK METOTLARI

Aşağıdaki örneklerde sınıf metodunun miras alma konusundaki kullanışlılığından bahsedeceğiz. Pe sınıfı oluşturacağız ve Pet sınıfı, Cat ve Dog sınıfları tarafından miras alınacak. about() metodu tanımlayarak başlayalım.

1.Durum:

class Pet:  
 \_class\_info=”Pet animals”  
   
 def about(self):  
 print(“This class is about “ + self.\_class\_info + “!”)  
class Dog(Pet):  
 \_class\_info = “Man’s best friends”  
class Cat(Pet):  
 \_class\_info = “All kinds of cats”  
p=Pet()  
print(p.about())  
d=Dog()  
print(d.about())  
c=Cat()  
print(c.about())

This class is about Pet animals!

None

This class is about Man’s best friends!

None

This class is about All kinds of cats!

None

* Sınıfın ne hakkında olduğunu öğrenmek için Pet, Dog ve Cat sınıflarının örneklerini oluşturmamız lazım.
* Direk Pet.about() şeklinde çağıramayız. Pet.about(p) şeklinde çağırabiliriz.

2.Durum: “about” metodunu statik metot olarak tanımlayalım.

class Pet:  
 \_class\_info="Pet animals"  
 @staticmethod  
 def about():  
 print("This class is about " + Pet.\_class\_info + "!")  
class Dog(Pet):  
 \_class\_info = "Man's best friends"  
class Cat(Pet):  
 \_class\_info = "All kinds of cats"  
Pet.about()  
Dog.about()  
Cat.about()

This class is about Pet animals!

This class is about Pet animals!

This class is about Pet animals!

* Burada Pet.about(), Dog.about() ve Cat.about() metotları çalışır fakat “self” parametresi olmadığı için alt sınıf olan Dog ve Cat’i göremediğinden doğru sonucu vermez.

3.Durum: Sınıf metodu bu iki durumdan kurtulmamızı sağlar.

class Pet:  
 \_class\_info="Pet animals"  
 @classmethod  
 def about(cls):  
 print("This class is about " + cls.\_class\_info + "!")  
class Dog(Pet):  
 \_class\_info = "Man's best friends"  
class Cat(Pet):  
 \_class\_info = "All kinds of cats"  
Pet.about()  
Dog.about()  
Cat.about()

This class is about Pet animals!

This class is about Man's best friends!

This class is about All kinds of cats!

* Sınıf metodu, alt sınıflarında metodu direk kullanabilmemizi sağlıyor.

ÖRNEK.

Here we are creating 3 methods : rateperkm, noofcabs, avgnoofpassengers. First one is instance method and other two are class methods.

* rateperkm returns price of cab fare per km which is calculated by dividing total bill by no. of kms cab traveled.
* noofcabs returns number of cabs running. Think about cab agency which owns many cabs and wants to know how many cabs are busy
* avgnoofpassengers returns average number of passengers traveling in a car. To calculate average, it takes into account all the cabs running and number of passengers in each cab.

class Cab:

#Initialise variables for first iteration

numberofcabs = 0

numpassengers = 0

def \_\_init\_\_(self,driver,kms,places,pay,passengers):

self.driver = driver

self.running = kms

self.places = places

self.bill = pay

Cab.numberofcabs = Cab.numberofcabs + 1

Cab.numpassengers = Cab.numpassengers + passengers

#Returns price of cab fare per km

def rateperkm(self):

return self.bill/self.running

#Returns number of cabs running

@classmethod

def noofcabs(cls):

return cls.numberofcabs

#Returns average number of passengers travelling in a cab

@classmethod

def avgnoofpassengers(cls):

return int(cls.numpassengers/cls.numberofcabs)

firstcab = Cab("Ramesh", 80, ['Delhi', 'Noida'], 2200, 3)

secondcab = Cab("Suresh", 60, ['Gurgaon', 'Noida'], 1500, 1)

thirdcab = Cab("Dave", 20, ['Gurgaon', 'Noida'], 680, 2)

firstcab.driver

**'Ramesh'**

secondcab.driver

**'Suresh'**

thirdcab.driver

**'Dave'**

firstcab.rateperkm()

**27.5**

secondcab.rateperkm()

**25.0**

thirdcab.rateperkm()

**34.0**

Cab.noofcabs()

**3**

Cab.avgnoofpassengers()

**2**

* Yukarıdaki kod bir de statik metot ekleyebiliriz.  It is mainly useful for creating helper or utility functions like validation of driver name must be more than zero (can't be negative or zero)

class Cab:

@staticmethod

def billvalidation(pay):

return int(pay) > 0

Cab.billvalidation(0.2)

PROPERTIES VS. GETTERS AND SETTERS

Getter ve setter metotlar, veri kapsüllemeyi gerçekleştirmek için, çoğu nesne tabanlı programlama dilinde kullanılır. Bu metotlardan getter veri döndürürken setter veriyi değiştirir.

* İlk olarak, gizli veriyi (self.\_\_x) kapsüllemek için getter ve setter fonksiyonlarla Java tarzı nasıl bir sınıf oluşturabiliriz onu görelim.

**class** **P**:

**def** **\_\_init\_\_**(self, x):

self.\_\_x = x

**def** get\_x(self):

**return** self.\_\_x

**def** set\_x(self, x):

self.\_\_x = x

p1=P(42)  
p2=P(1000)  
print(p1.get\_x())  
p1.set\_x(47)  
p1.set\_x(p1.get\_x()+p2.get\_x())

print(p1.get\_x())

42

1047

* p1.set\_x(p1.get\_x()+p2.get\_x()) ifadesi pek şık değil.
* Eğer x public öznitelik olsaydı, p1.x=p1.x + p2.x şeklinde tanımlanacaktı.
* Bu örnekte biraz değişiklikler yapalım: x özniteliği 0 ile 1000 arasında değerler alsın. Eğer tanımlanan değer 0’dan küçük ise 0’a, Eğer 1000’den büyük ise 1000’e değer döndürsün.

**class** **P**: p1=P(1001)

**def** **\_\_init\_\_**(self, x):

self.set\_x(x)

**def** get\_x(self):

**return** self.\_\_x

**def** set\_x(self, x):

**if** x < 0:

self.\_\_x = 0

**elif** x > 1000:

self.\_\_x = 1000

**else**:

self.\_\_x = x

* Burada yine gizli öznitelik olduğu için değeri ve değer değiştirmeyi get\_x ve set\_x metotları ile yapabiliriz.
* Public öznitelik ve metot olmayan aşağıdaki kodu inceleyelim.

**class** **P2**:

**def** **\_\_init\_\_**(self, x):

self.x = x

al=P2(34)  
print(al.x)

34

* al.x ile public özniteliğe erişebiliyoruz. Burada tanımlama gizli olsaydı, bu şekilde ulaşamazdık. O yüzden gizli öznitelikleri getter ve setter metotları ile kullanmamızı tavsiye eder Java tabanlı diller. Böylelikle arayüzü değiştirmeden implementasyonu değiştirebiliriz.
* Python bize bu konuda bir çözüm öneriyor: “properties”.
* Aşağıdaki “property” içeren bir kodu inceleyelim:

**class** **P**:

**def** **\_\_init\_\_**(self, x):

self.x = x

@property

**def** x(self):

**return** self.\_\_x

@x.setter

**def** x(self, x):

**if** x < 0:

self.\_\_x = 0

**elif** x > 1000:

self.\_\_x = 1000

**else**:

self.\_\_x = x

p1=P(1001)  
print(p1.x)  
p1.x=20  
print(p1.x)

1000

20

* **@property** özelliği sayesinde “getter” yani değer döndüren fonksiyonun öznitelik gibi dışarıdan erişilebilmesini sağlıyoruz.
* Bir değer elde etmek için kullanılan bir yöntem "**@property"** ile dekore edilmiştir, yani bu satırı doğrudan başlığın önüne koyarız. Setter işlevi görmesi gereken yöntem "**@x.setter**" ile dekore edilmiştir. Eğer fonksiyon "f" olarak adlandırılsaydı, onu "**@f.setter**" ile süslememiz gerekirdi.
* Dikkate değer iki şey var: Sadece "self.x = x" kod satırını \_\_init\_\_ yöntemine koyduk ve değerlerin sınırlarını kontrol etmek için x özellik yöntemi kullanıldı.
* İkinci ilginç olan ise, "def x(self)" ve "def x(self,x)" parametresi aynı ve farklı sayıda "iki" metot yazmış olmamız.
* Alternatif olarak, “property”’i tanımlamak için dekoratörler olmadan farklı bir sözdizimi kullanabilirdik.

**class** **P**:

**def** **\_\_init\_\_**(self, x):

self.set\_x(x)

**def** get\_x(self):

**return** self.\_\_x

**def** set\_x(self, x):

**if** x < 0:

self.\_\_x = 0

**elif** x > 1000:

self.\_\_x = 1000

**else**:

self.\_\_x = x

x = **property**(get\_x, set\_x)

* En son sürümde hala başka bir sorun var. Artık x'in değerine erişmenin veya değerini değiştirmenin iki yolu var: Ya "p1.x = 42" kullanarak ya da "p1.set\_x(42)" ile.
* Bu istenilen bir şey değil: tek yöntem olmalı
* Getter ve setter yöntemlerini, artık P sınıfımızın kullanıcıları tarafından erişilemeyen private yöntemlere çevirerek bu sorunu kolayca çözebiliriz. **\_\_set\_x()** yaparız.

**class** **P**:

**def** **\_\_init\_\_**(self, x):

self.\_\_set\_x(x)

**def** \_\_get\_x(self):

**return** self.\_\_x

**def** \_\_set\_x(self, x):

**if** x < 0:

self.\_\_x = 0

**elif** x > 1000:

self.\_\_x = 1000

**else**:

self.\_\_x = x

x = **property**(\_\_get\_x, \_\_set\_x)

p1=P(79)  
p1.set\_x(12)

AttributeError: 'P' object has no attribute 'set\_x'

* Görüyoruz ki burada x özniteliğine set\_x() metodu ile ulaşmak mümkün değil.
* Sadece p1.x=1000 yaparak değiştirebiliriz.

p1=P(79)  
p1.x=1000  
print(p1.x)

**Örnek.** Bir **Robot()** sınıf tanımlayalım. name ve build\_year öznitelikleri public, potential\_physical ve potential\_psychic öznitelikleri private (gizli) olsun. **condition()** metodu oluşturalım ve burada potential\_physical ve poteltial\_psychic öznitelikleri toplamının değerine göre çıktı döndürsün: değer <=-1 ise “I feel misarable”, değer<=0 ise “I feel bad”, deger<=0.5 ise “Could be worse”, deger<=1 ise “Seems to be ok” aksi durumda da “Great” yazdırsın. Robot() sınfından “robot1” örneği oluşturulduğunda, “robot1.conditon” şeklinde sonuç almaya izin veren program yazınız.

**class** **Robot**:

**def** **\_\_init\_\_**(self, name, build\_year, lk = 0.5, lp = 0.5 ):

self.name = name

self.build\_year = build\_year

self.\_\_potential\_physical = lk

self.\_\_potential\_psychic = lp

@property

**def** condition(self):

s = self.\_\_potential\_physical + self.\_\_potential\_psychic

**if** s <= -1:

**return** "I feel miserable!"

**elif** s <= 0:

**return** "I feel bad!"

**elif** s <= 0.5:

**return** "Could be worse!"

**elif** s <= 1:

**return** "Seems to be okay!"

**else**:

**return** "Great!"

**if** \_\_name\_\_ == "\_\_main\_\_":

x = Robot("Marvin", 1979, 0.2, 0.4 )

y = Robot("Caliban", 1993, -0.4, 0.3)

**print**(x.condition)

**print**(y.condition)

ÖRNEK.

Restaurant() sınıfı oluştur.

Sınıf değişkeni oluştur.

Yiyecekler={“et”:25,..}

İcecekler={}

Tatlılar={}

4 tane nesne metodu oluştur:

Yiyeceksec

Iceceksec

Tatlisec

Hesap()--- Ödenekcek miktar: self.hesap

Nurcan=Restaurant()

Nurcan.Yiyeceksec(input(“Yiyecek:”))

Nurcan.İceceksec(input(“Icecek:”))

Nurcan.Tatlısec(input(“Tatlı:”))

Nurcan.Hesap()

Örnek.Bir banka hesabında, para çekmek, para yatırmak, havale,

Kalan parayı göstermek gibi işlemleri bütün kullanıcılar için

Gerçekleştirecek sınıf kodu yazınız.

BankaHesabı()

KalanPara()

ParaYatır()

ParaCek()

Havale()

Ali=BankaHesabı(5000)

Python Tanımlayıcıları (Descriptors) Nelerdir?

Tanımlayıcılar, tanımlayıcı protokolünün bir yöntemini uygulayan Python nesneleridir; bu, diğer nesnelerin öznitelikleri olarak erişildiğinde özel davranışa sahip nesneler oluşturmanıza olanak tanır. Burada tanımlayıcı protokolünün doğru tanımını görebilirsiniz: Aşağıdaki metotlardan herhangi birini tanımlamış objelere descriptor diyoruz.

\_\_get\_\_(self, obj, type=None) -> object

\_\_set\_\_(self, obj, value) -> None

\_\_delete\_\_(self, obj) -> None

\_\_set\_name\_\_(self, owner, name)

Tanımlayıcınız yalnızca **.\_\_get\_\_()** öğesini uygularsa, bunun veri olmayan bir tanımlayıcı **(non-data descriptors)** olduğu söylenir. **.\_\_set\_\_()** veya **.\_\_delete\_\_()** uygularsa, bunun bir veri tanımlayıcı **(data descriptors**) olduğu söylenir. Bu farkın sadece adla ilgili olmadığını, aynı zamanda davranışta da bir fark olduğunu unutmayın. Bunun nedeni, daha sonra göreceğiniz gibi, veri tanımlayıcılarının arama işlemi sırasında önceliğe sahip olmasıdır.

**ÖRNEK**. Sabit değer döndüren basit bir descriptor tanımlayalım.

Burada Ten() sınıfı \_\_get\_\_() metodu sabit 10 değerini döndüren bir descriptor dür.

**class** **Ten**:

**def** \_\_get\_\_(self, obj, objtype=**None**):

**return** 10

Descriptor’ü (Tanımlayıcıyı) kullanmak için, başka bir sınıfta bir sınıf değişkeni olarak saklanması gerekir:

**class** **A**:

x = 5 *# Regular class attribute*

y = Ten() *# Descriptor instance*

Etkileşimli bir oturum, normal öznitelik araması ile descriptor araması arasındaki farkı gösterir:

**>>>** a = A() *# Make an instance of class A*

**>>>** a.x *# Normal attribute lookup*

5

**>>>** a.y *# Descriptor lookup*

10

os Modülü

Bildiğiniz gibi, işletim sistemlerinin çalışma mantığı birbirinden farklıdır. Örneğin Windows ve GNU/Linux işletim sistemleri aynı işi birbirlerinden farklı şekillerde yaparlar. Mesela Windows’ta bir dizin içinde hangi klasör ve dosyaların olduğunu öğrenmek için dir komutunu kullanırız. GNU/Linux’ta ise aynı işlev için ls adlı bir komut vardır.

## os.getcwd()

os modülünün getcwd() fonksiyonu bize o anda içinde bulunduğumuz dizinin adını verir:

**>>>** os.getcwd()

C:\Users\nurca\Desktop\YBS212\_HAFTA3\YBS212\_HAFTA5

## os.listdir()

os modülünün listdir() fonksiyonu, bize bir dizin içindeki dosya ve klasörleri listeleme imkanı verir. listdir(), os modülünün en kullanışlı fonksiyonlarından biridir.

Mesela o anda içinde bulunduğumuz dizindeki dosya ve klasörleri listelemek istersek bu fonksiyonu şöyle kullanabiliriz:

**>>>** mevcut\_dizin = os.getcwd()

**>>>** os.listdir(mevcut\_dizin)

['.gitignore', '.idea', 'Hafta5\_descriptors.py', 'Lib', 'pyvenv.cfg', 'Scripts']

Mesela bu liste üzerinde bir döngü kurabilirsiniz:

**>>> for** i **in** os.listdir(os.getcwd()):

**...**  print(i)

Ya da bir dizin içindeki, belli bir uzantıya sahip dosyaları süzebilirsiniz:

**>>> for** i **in** os.listdir(os.getcwd()):

**...**  **if** i.endswith('.py'):

**...**  print(i)

Hafta5\_descriptors.py

**Dinamik aramalar**

İlginç descriptorler (tanımlayıcılar) genellikle sabitleri döndürmek yerine hesaplamaları çalıştırır:

**import** **os**

**class** **DirectorySize**:

**def** \_\_get\_\_(self, obj, objtype=**None**):

**return** len(os.listdir(obj.dirname))

**class** **Directory**:

size = DirectorySize() *# Descriptor instance*

**def** \_\_init\_\_(self, dirname):

self.dirname = dirname *# Regular instance attribute*

Etkileşimli bir oturum, aramanın dinamik olduğunu gösterir, her seferinde farklı, güncellenmiş yanıtları hesaplar:

**>>>** s = Directory('songs')

**>>>** g = Directory('games')

**>>>** s.size *# The songs directory has twenty files*

20

**>>>** g.size *# The games directory has three files*

3

**>>>** os.remove('games/chess') *# Delete a game*

**>>>** g.size *# File count is automatically updated*

2

* Bu örnek, tanımlayıcıların hesaplamaları nasıl çalıştırabileceğini göstermenin yanı sıra, \_\_get\_\_() parametrelerinin amacını da ortaya koymaktadır.
* **self** parametresi **size**, DirectorySize örneğidir**. obj** parametresi, bir Directory örneği olan **g** veya **s**'dir.
* \_\_get\_\_() yönteminin hedef dizini öğrenmesini sağlayan **obj** parametresidir. **objtype** parametresi, Directory sınıfıdır.

**Yönetilen Öznitelikler(Managing attributes)**

Örneğimize geçmeden önce pythondaki logging modülü hakkında bilgi verelim.

Python'da Nasıl Log Tutulur ?

#### **Logging nedir?**

Çalıştırdığınız sistem veya programda meydana gelen olayların takibi için tutulan kayıtlara log denir. Bu kayıtlar “**log**” dosyası diye bilinen bir dosyada tutulur. Bu dosya, sistemin analizi, sorunların çözümü ve debugging *(hata ayıklama)* için kullanılır. Bu log dosyalarının tutulma işlemine de **logging** denir.

**Python logging kütüphanesi neden kullanılır?**

logging kütüphanesinin birçok faydasından bir tanesi de bütün kodunuzu tarayıp yazdığınız bütün “print” ifadelerini silmek zorunda olmayışınızdır. Dahası, daha sonra gözden geçirmek üzere son çalışmanızın bütün sistem olaylarının bir kaydını tutmuş oluyorsunuz (düzenli uygulandığı takdirde). logging kütüphanesi sadece metin veya sayılardan daha fazlasının kaydını tutmanıza olanak tanıyor.

**Log seviyeleri nelerdir?**

Önem sırası az çoğa olacak şekilde log seviyelerin şunlardır:

* **Debug:** Program çalışmaya devam ederken ortaya çıkan beklenmedik herhangi çıktıyı düzeltmek için programcı tarafından kullanılır.
* **Info:** Programda olan olaylar hakkında bilgi verir.
* **Warning:** Beklenmedik hata vb. bir şeyler olduğunda uyarı verir.
* **Error:** Programda bir problem oluştuğu zaman, genellikle program bazı komutları çalıştıramadığı zaman kullanılır.
* **Critical:** En yüksek seviyedeki hatadır. Genellikle program daha fazla çalışamayacağı zaman kullanılır.

#### **Konfigürasyon**

Logging işlemini başlamadan önce loglarımız nasıl tutacağımızı belirlememiz gerekiyor.

logging.basicConfig(filename="logfile.txt",format="%(asctime)s %(message)s",filemode="w",level=logging.DEBUG)

1.**FILENAME:** Loglarımızın tutulacağı dosyanın ismi. Bir dizin belirtilmezse kendi bulunduğu dizinde tutar.

2.**FORMAT:** Logun tutulacağı formatı belirler. (örneğimizde 2 parametre verilmiştir.)

3.**FILEMODE:** Logun tutulacağı modu belirler. “w” veya “a” tercih edebilirsiniz. “w” daha önceden yazılan dosyanın içeriğini yazmaya baştan başlarken, “a” dosyanın son kaldığı yerden devam eder.

4.**LEVEL:** Logun minimum tutulacağı seviyeyi belirler. Örneğin “Warning” seviyesi ayarlanırsa “Info” veya “Debug” logları tutulmayacaktır.

**Not:** LEVEL opsiyonel bir parametredir daha sonra da “setLevel()” metoduyla da ayarlanabilir.

Tanımlayıcılar(descriptorler) için popüler bir kullanım, örnek verilerine erişimi yönetmektir.

* Public özniteliğe erişildiğinde tanımlayıcının **\_\_get\_\_()** ve **\_\_set\_\_()** yöntemleri tetiklenir. Aşağıdaki örnekte, **age** public özniteliktir ve **\_\_age**, private özniteliktir.
* Public özniteliğe erişildiğinde, tanımlayıcı aramayı veya güncellemeyi günlüğe kaydeder:

**import** **logging**

logging.basicConfig(filename=”logfile.txt”,level=logging.INFO)

**class** **LoggedAgeAccess**:

**def** \_\_get\_\_(self, obj, objtype=**None**):

value = obj.\_age

logging.info('Accessing **%r** giving **%r**', 'age', value)

**return** value

**def** \_\_set\_\_(self, obj, value):

logging.info('Updating **%r** to **%r**', 'age', value)

obj.\_age = value

**class** **Person**:

age = LoggedAgeAccess() *# Descriptor instance*

**def** \_\_init\_\_(self, name, age):

self.name = name *# Regular instance attribute*

self.age = age *# Calls \_\_set\_\_()*

**def** birthday(self):

self.age += 1 *# Calls both \_\_get\_\_() and \_\_set\_\_()*

# Etkileşimli bir oturum, yönetilen öznitelik olan ‘age’e ait tüm erişimlerin günlüğe kaydedildiğini, ancak normal öznitelik olan ‘name’in günlüğe kaydedilmediğini gösterir:

**>>>** mary = Person('Mary M', 30) *# The initial age update is logged*

INFO:root:Updating 'age' to 30

**>>>** dave = Person('David D', 40)

INFO:root:Updating 'age' to 40

**>>>** vars(mary) *# The actual data is in a private attribute*

{'name': 'Mary M', '\_age': 30}

**>>>** vars(dave)

{'name': 'David D', '\_age': 40}

**>>>** mary.age *# Access the data and log the lookup*

INFO:root:Accessing 'age' giving 30

30

**>>>** mary.birthday() *# Updates are logged as well*

INFO:root:Accessing 'age' giving 30

INFO:root:Updating 'age' to 31

**>>>** dave.name *# Regular attribute lookup isn't logged*

'David D'

**>>>** dave.age *# Only the managed attribute is logged*

INFO:root:Accessing 'age' giving 40

40

Bu örnekle ilgili önemli bir sorun, \_age özel adının LoggedAgeAccess sınıfında kablolu olmasıdır.

Bu, her örneğin yalnızca bir kayıtlı özniteliğe sahip olabileceği ve adının değiştirilemeyeceği anlamına gelir.

**Customized names(Özelleştirilmiş adlar)**

Bir sınıf tanımlayıcıları kullandığında, her tanımlayıcıya hangi değişken adının kullanıldığı hakkında bilgi verebilir. Bu örnekte, Person sınıfının iki tanımlayıcı örneği vardır: **name** and **age**

* Person sınıfı tanımlandığında, alan adlarının kaydedilebilmesi için **LoggedAccess**'te **\_\_set\_name\_\_()** öğesine bir geri arama yapar ve her tanımlayıcıya kendi **public\_name** ve **private\_name** verir:

**import** **logging**

logging.basicConfig(level=logging.INFO)

**class** **LoggedAccess**:

**def** \_\_set\_name\_\_(self, owner, name):

self.public\_name = name

self.private\_name = '\_' + name

**def** \_\_get\_\_(self, obj, objtype=**None**):

value = getattr(obj, self.private\_name)

logging.info('Accessing **%r** giving **%r**', self.public\_name, value)

**return** value

**def** \_\_set\_\_(self, obj, value):

logging.info('Updating **%r** to **%r**', self.public\_name, value)

setattr(obj, self.private\_name, value)

**class** **Person**:

name = LoggedAccess() *# First descriptor instance*

age = LoggedAccess() *# Second descriptor instance*

**def** \_\_init\_\_(self, name, age):

self.name = name *# Calls the first descriptor*

self.age = age *# Calls the second descriptor*

**def** birthday(self):

self.age += 1

Etkileşimli bir oturum, Alan adlarının kaydedilebilmesi için **Person** sınıfının **\_\_set\_name\_\_()** öğesini çağırdığını gösterir.

Tanımlayıcıyı tetiklemeden aramak için burada **vars()'**ı çağırıyoruz:

**>>>** vars(vars(Person)['name'])

{'public\_name': 'name', 'private\_name': '\_name'}

**>>>** vars(vars(Person)['age'])

{'public\_name': 'age', 'private\_name': '\_age'}

Yeni sınıf artık hem **‘name’** hem de ‘**age** ‘,erişimi günlüğe kaydeder:

* Sınıf(class) soyut bir kavram iken nesne(object) somut bir kavramdır.
* Örneğin; ‘Araba’ tüm arabaları tanımlayan bir üst sınıf, yani soyut bir yapı iken ‘Kamyon’ bu sınfa ait alt bir bireyi, yani somut bir nesneyi tanımlar.
* Araba: **‘Renk, Kapasite, Model, Silindir’** gibi özelliklere sahipse, ‘Kamyon’ da doğal olarak Araba sınıfının tüm özelliklerine (attributes) sahip olmalıdır.

**Örnek.** Bir araba sınıfından **‘Kamyon’, ‘Taksi’** nesneleri oluşturup, bazı özelliklerini ekrana yazdıran uygulamayı farklı şekillerde kodlayınız.

class Araba():  
 model="BMC"  
 renk="Kırmızı"  
 kapasite=2600  
 silindir=6  
kamyon=Araba() #Kamyon nesnesi  
taksi=Araba() #Taksi nesnesi  
#Taksi nesnesinin bazı özellikleri değiştirildi

KALITIM(INHERITANCE)

Genel olarak miras (inheritance), var olan bir sınıftan yeni bir sınıf oluşturmaya dayanır. Böyle yaparak bir sınıf hiyerarşisi oluşturmuş oluyoruz.

Miras, programcının var olan sınıflardan yeni sınıflar oluşturmasını sağlar ve bu da miras yolu ile oluşturulmuş sınıfın, ebeveyn sınıfının öznitelik ve metotlarını kullanmasına imkan tanır. Bu demek oluyor ki, miras, kodların tekrar kullanılabilirliğine imkan tanır.

Bir sınıfın miras alındığı sınıfa “parent (ebeveyn) sınıf” denir. Parent sınıfını miras alan sınıfa da “subclass(alt sınıf)” denir. Sınıflar arasında hiyerarşik bir ilişki vardır.

Günlük hayatımızdaki ilişkileri düşünün: Bisiklet, motor, araba, kamyon, otobüs birer “Araç”tır.

Herbir aracın kendi içinde kategorileri vardır. Örneğin araba: Spor araba, hibrit araba, sedan araba…

Bunların hepsi “araba” dır ve “Araç” sınıfının üyesidir.

Pythonda “Araç” sınıfı oluşturup “hızlandır”, “frenyap” gibi metotlar ekleyebiliriz.

Arabalar, otobüsler, kamyonlar ve bisikletler “Araç” sınıfından miras alan alt sınıflar olacaktır.

Başka bir örnek :  Bir şirketin çalışanlarını tasarlamak için sınıflar oluşturuyoruz. Bunun için Yönetici, Proje Direktörü, İşçi gibi sınıflar oluşturmamız gerekiyor. Aslında baktığımız zaman bu sınıfların hepsinin belli ortak metotları ve özellikleri bulunuyor. O zaman bu ortak özellikleri ve metodları tekrar tekrar bu sınıfların içinde tanımlamak yerine, bir tane ebeveyn sınıfı tanımlayıp alt sınıfların bu evebeyn sınfının özelliklerini ve metotlarını almalarını sağlayabiliriz.

* Bir alt sınıf yazımı aşağıdaki şekildedir:

class DerivedClassName(BaseClassName):

pass

class child\_class\_name(parent\_class\_name):

pass

* pass deyimi yerine diğer tüm sınıflarda olduğu gibi metotlar ve nitelikler olacaktır. DerivedClassName, türetilmiş sınıf tanımını içeren bir kapsamda tanımlanmalıdır.
* Kalıtım ilkesinin nasıl çalıştığını göstermek için Python eğitimimizin önceki bölümlerinden sevgili robotlarımıza veya daha iyi Robot sınıfına bağlı kalacağız. Robot'tan miras kalan bir PhysicianRobot sınıfı tanımlayacağız.

**class** **Robot**:

**def** **\_\_init\_\_**(self, name):

self.name = name

**def** say\_hi(self):

**print**("Hi, I am " + self.name)

**class** **PhysicianRobot**(Robot):

**pass**

x = Robot("Marvin")

y = PhysicianRobot("James")

**print**(x, **type**(x))

**print**(y, **type**(y))

y.say\_hi()

<\_\_main\_\_.Robot object at 0x7fd0080b3ba8> <class '\_\_main\_\_.Robot'>

<\_\_main\_\_.PhysicianRobot object at 0x7fd0080b3b70> <class '\_\_main\_\_.PhysicianRobot'>

Hi, I am James

* PhysicianRobot sınıfımızın koduna bakarsanız, bu sınıfta herhangi bir öznitelik veya yöntem tanımlamadığımızı görebilirsiniz. PhysicianRobot sınıfı Robot'un bir alt sınıfı olduğundan, bu durumda hem \_\_init\_\_ hem de say\_hi yöntemini devralır. Bu yöntemleri devralmak, onları PhysicianRobot sınıfında tanımlanmış gibi kullanabileceğimiz anlamına gelir. Bir PhysicianRobot örneği oluşturduğumuzda, \_\_init\_\_ işlevi de bir name niteliği oluşturacaktır. Say\_hi yöntemini, yukarıdaki koddan çıktıda gördüğümüz gibi PhysisicianRobot y nesnesine uygulayabiliriz.

### Overriding (İptal Etme)

Eğer biz miras aldığımız metotları **aynı isimle** kendi sınıfımızda tekrar tanımlarsak , artık metodu çağırdığımız zaman **miras aldığımız değil kendi metodumuz çalışacaktır.** Buna Nesne Tabanlı Programlamada bir metodu override etmek denmektedir.

* Yeni PhysicianRobot sınıfımıza geri dönelim. Şimdi bir PhysicianRobot örneğinin farklı bir şekilde merhaba demesi gerektiğini hayal edin. Bu durumda, say\_hi yöntemini PhysicianRobot alt sınıfının içinde yeniden tanımlamamız gerekir:

**class** **Robot**:

**def** **\_\_init\_\_**(self, name):

self.name = name

**def** say\_hi(self):

**print**("Hi, I am " + self.name)

**class** **PhysicianRobot**(Robot):

**def** say\_hi(self):

**print**("Everything will be okay! ")

**print**(self.name + " takes care of you!")

y = PhysicianRobot("James")

y.say\_hi()

### OUTPUT:

Everything will be okay!

James takes care of you!

* Metodun ebeveyn sınıfında var olan bir metodun ismi ile alt sınıfında bir metot tanımlarsak, ebeveyn sınıfındaki metodu etkisiz hale getirmiş oluruz. Buna “overriding” denir.
* Bir üst sınıfın yöntemi, alt sınıfta aynı ada sahip bir yöntem tanımlanarak geçersiz kılınır.
* “Overriding” yapılmış bir metodun orijinal metoduna erişmek istersek, doğrudan sınıf ismini kullanmalıyız.
* Bir sınıfta bir yöntem geçersiz kılınırsa, orijinal yönteme hala erişilebilir, ancak bunu yöntemi doğrudan sınıf adıyla, yani **Robot.say\_hi(y)** ile çağırarak yapmalıyız. Bunu aşağıdaki kodda gösteriyoruz:

y = PhysicianRobot("Doc James")

y.say\_hi()

**print**("... and now the 'traditional' robot way of saying hi :-)")

Robot.say\_hi(y)

Everything will be okay!

Doc James takes care of you!

... and now the 'traditional' robot way of saying hi :-)

Hi, I am Doc James

* Miras alınan bir sınıfın, üst sınıftan yöntemleri devralabileceğini ve geçersiz kılabileceğini gördük.
* Bunun yanı sıra, bir alt sınıf genellikle, üst sınıfta bulunmayan ek işlevlere sahip ek yöntemlere ihtiyaç duyar.

Şöyle bir örnek tasarlayalım: Robot süper sınıfı ve PhysicianRobot isimli alt sınıf olsun. Robot sınıfının içerisinde “health\_level” isimli bir parametre olsun ve bu parametre 0 ile 1 arasında rastgele değerler alsın. Robot sınıfının içerisinde “needs\_a\_doctor” metodu tanımlayalım. Bu metotta eğer parametre değeri 0.8’in altında ise True çıktısını değilse False çıktısını versin.

PhysicianRobot alt sınıfı Robot sınıfını miras alsın ve içerisindeki “say\_hi” metodunu “override” edelim: Çıktısı “Everything will be ok !” “self.name takes care of you!” olsun. Ayrıca “heal” isimli iyileştirme metodu tanımlayalım. Bu metot “health\_level” parametresinin değerine göre random.uniform() metodunu kullanarak iyileştirme yapsın ve yazdırsın. En sonda tanımlayacağınız örnek üzerinden 5 defa “needs\_a\_doctor” metodunu ve heal() metodunu yazdırtalım. Her defasında isim(x.name) ve x.health\_level değerlerini bir listede toplayalım.

**import** **random**

**class** **Robot**:

**def** **\_\_init\_\_**(self, name):

self.name = name

self.health\_level = random.random()

**def** say\_hi(self):

**print**("Hi, I am " + self.name)

**def** needs\_a\_doctor(self):

**if** self.health\_level < 0.8:

**return** True

**else**:

**return** False

**class** **PhysicianRobot**(Robot):

**def** say\_hi(self):

**print**("Everything will be okay! ")

**print**(self.name + " takes care of you!")

**def** heal(self, robo):

robo.health\_level = random.uniform(robo.health\_level, 1)

**print**(robo.name + " has been healed by " + self.name + "!")

doc = PhysicianRobot("Dr. Frankenstein")

rob\_list = []

**for** i **in** **range**(5):

x = Robot("Marvin" + **str**(i))

**if** x.needs\_a\_doctor():

**print**("health\_level of " + x.name + " before healing: ", x.health\_level)

doc.heal(x)

**print**("health\_level of " + x.name + " after healing: ", x.health\_level)

rob\_list.append((x.name, x.health\_level))

**print**(rob\_list)

health\_level of Marvin0 before healing: 0.5562005305000016

Marvin0 has been healed by Dr. Frankenstein!

health\_level of Marvin0 after healing: 0.7807651150204282

health\_level of Marvin1 before healing: 0.40571527448692757

Marvin1 has been healed by Dr. Frankenstein!

health\_level of Marvin1 after healing: 0.4160992532325318

health\_level of Marvin2 before healing: 0.3786957462635925

Marvin2 has been healed by Dr. Frankenstein!

health\_level of Marvin2 after healing: 0.5474124864506639

health\_level of Marvin3 before healing: 0.6384666796845331

Marvin3 has been healed by Dr. Frankenstein!

health\_level of Marvin3 after healing: 0.6986491928780778

health\_level of Marvin4 before healing: 0.5983126049766974

Marvin4 has been healed by Dr. Frankenstein!

health\_level of Marvin4 after healing: 0.6988801787833587

[('Marvin0', 0.7807651150204282), ('Marvin1', 0.4160992532325318), ('Marvin2', 0.5474124864506639), ('Marvin3', 0.6986491928780778), ('Marvin4', 0.6988801787833587)]

* Bir yöntemi geçersiz kıldığımızda, bazen üst sınıfın yöntemini bazı yeni şeylerde yeniden kullanmak isteriz.
* Bunu göstermek için PhysicianRobot'un yeni bir versiyonunu yazacağız. say\_hi, Robot sınıfı versiyonundaki metni ve de “and I am a physician!" metnini döndürmelidir.

**class** **PhysicianRobot**(Robot):

**def** say\_hi(self):

Robot.say\_hi(self)

**print**("and I am a physician!")

doc = PhysicianRobot("Dr. Frankenstein")

doc.say\_hi()

### OUTPUT:

Hi, I am Dr. Frankenstein

and I am a physician!

* Gereksiz kod yazmak istemiyoruz ve bu nedenle Robot.say\_hi(self) adını verdik. Süper işlevi de kullanabiliriz:

**class** **PhysicianRobot**(Robot):

**def** say\_hi(self):

**super**().say\_hi()

**print**("and I am a physician!")

doc = PhysicianRobot("Dr. Frankenstein")

doc.say\_hi()

### OUTPUT:

Hi, I am Dr. Frankenstein

and I am a physician!

MULTIPLE INHERITANCE ( ÇOKLU MİRAS)

“SubclassName” alt sınıfının BaseClass1, BaseClass2, BaseClass3 vb. üst sınıflardan miras aldığı bir sınıf tanımı şöyle görünür:

class SubclassName(BaseClass1, BaseClass2, BaseClass3, ...):

pass

ÖRNEK.

Örneğin artık Çalışan sınıfının **init** metodunu kullanmak yerine Yönetici sınıfında **init** metodunu override edebiliriz. Böylelikle Yönetici sınıfına ekstra özellikler(**attribute**) ekleyebiliriz.

**class** Çalışan():

**def** \_\_init\_\_(self,isim,maaş,departman):

print("Çalışan sınıfının init fonksiyonu")

self**.**isim **=** isim

self**.**maaş **=** maaş

self**.**departman **=** departman

**def** bilgilerigoster(self):

print("Çalışan sınıfının bilgileri.....")

print("İsim : {} \nMaaş: {} \nDepartman: {}\n"**.**format(self**.**isim,self**.**maaş,self**.**departman))

**def** departman\_degistir(self,yeni\_departman):

print("Departman değişiyor....")

self**.**departman **=** yeni\_departman

**class** Yönetici(Çalışan):

**def** \_\_init\_\_(self,isim,maaş,departman,kişi\_sayısı): *# Sorumlu olduğu kişi sayısı*

print("Yönetici sınıfının init fonksiyonu")

self**.**isim **=** isim

self**.**maaş **=** maaş

self**.**departman **=** departman

self**.**kişi\_sayısı **=** kişi\_sayısı *# Yeni eklenen özellik*

**def** zam\_yap(self,zam\_miktarı):

print("Maaşa zam yapılıyor....")

self**.**maaş **+=** zam\_miktarı

a **=** Yönetici("Mustafa Murat Coşkun",3000,"Bilişim",10) *# Yönetici sınıfının init fonksiyonu. Override edildi.*

Yönetici sınıfının init fonksiyonu

İsterseniz şimdi de **bilgilerigoster** metodunu override edelim.

**class** Yönetici(Çalışan):

**def** \_\_init\_\_(self,isim,maaş,departman,kişi\_sayısı): *# Sorumlu olduğu kişi sayısı*

print("Yönetici sınıfının init fonksiyonu")

self**.**isim **=** isim

self**.**maaş **=** maaş

self**.**departman **=** departman

self**.**kişi\_sayısı **=** kişi\_sayısı *# Yeni eklenen özellik*

**def** bilgilerigoster(self):

print("Yönetici sınıfının bilgileri.....")

print("İsim : {} \nMaaş: {} \nDepartman: {}\nSorumlu kişi sayısı: {}"**.**format(self**.**isim,self**.**maaş,self**.**departman,self**.**kişi\_sayısı))

**def** zam\_yap(self,zam\_miktarı):

print("Maaşa zam yapılıyor....")

self**.**maaş **+=** zam\_miktarı

b **=** Yönetici("Serhat Say",2500,"Pazarlama",5)

Yönetici sınıfının init fonksiyonu

In [93]:

b**.**bilgilerigoster() *# Kendi yazdığımız fonksiyon çağrıldı. Overriding*

Yönetici sınıfının bilgileri.....

İsim : Serhat Say

Maaş: 2500

Departman: Pazarlama

Sorumlu kişi sayısı: 5

ÇOKLU MİRAS

Eğitimimizin önceki bölümünde, kalıtımı veya daha spesifik "tek kalıtımı" ele aldık. Gördüğümüz gibi, bir sınıf bu durumda bir sınıftan miras alır. Çoklu kalıtım ise, bir sınıfın birden fazla üst sınıftan öznitelikleri ve yöntemleri devralabileceği bir özelliktir.

SubClassName alt sınıfının BaseClass1, BaseClass2, BaseClass3 vb. üst sınıflardan miras aldığı bir sınıf tanımı şöyle görünür:

**class SubclassName(BaseClass1, BaseClass2,BaseClass3, ...):**

**pass**

* Çoklu kalıtım ilkelerini bir örnekle tanıtmak istiyoruz. Bu amaçla bağımsız sınıflara uygulayacağız: bir "Clock" ve bir "Calendar" sınıfı.
* Bundan sonra, adından da anlaşılacağı gibi, "Clock" ve "Calendar" kombinasyonu olan bir "CalendarClock" sınıfını tanıtacağız. CalendarClock, hem "Clock” hem de "Calendar"den miras alır.
* Clock sınıfı, bir saatin tik taklarını simüle eder. Bu sınıfın bir örneği, self.hours, self.minutes ve self.seconds özniteliklerinde saklanan zamanı içerir.
* Prensip olarak, \_\_init\_\_ yöntemini ve set yöntemini şöyle yazabilirdik:

def \_\_init\_\_(self,hours=0, minutes=0, seconds=0):

self.\_hours = hours

self.\_\_minutes = minutes

self.\_\_seconds = seconds

def set(self,hours, minutes, seconds=0):

self.\_hours = hours

self.\_\_minutes = minutes

self.\_\_seconds = seconds

* Hem \_\_init\_\_ hem de set metot içerisinde aynı tanımlamaları yaptık. Gereksiz koddan kaçınmak için \_\_init\_\_ metodunun içerisinde set metodunu çağıralım.
* Ayrıca set metodunu biraz düzenleyelim. “hours” 0 ile 24 arasında bir değer, “minutes” 0 ile 60 arasında bir değer ve “seconds” 0 ile 60 arasında bir değer olsun ve aksi durumlarda her bir değişken için “type error” yazdırsın.

class Clock():  
 def \_\_init\_\_(self,hours,minutes,seconds):  
 self.set\_clock(hours,minutes,seconds)  
 def set\_clock(self,hours,minutes,seconds):  
 if type(hours)==int and 0<=hours<24:  
 self.hours=hours  
 else:  
 raise TypeError("Hours must be integers between 0 and 23 !")  
 if type(minutes)==int and 0<=minutes<60:  
 self.minutes=minutes  
 else:  
 raise TypeError("Minutes must be integers between 0 and 59 !")  
 if type(seconds)==int and 0<=seconds<60:  
 self.seconds=seconds  
 else:  
 raise TypeError("Seconds must be between 0 and 60 !")  
 def \_\_str\_\_(self):  
 return "{}:{}:{}".format(self.hours,self.minutes,self.seconds)  
 def tick(self):  
 if self.seconds==59:  
 self.seconds=0  
 if self.minutes==59:  
 self.minutes=0  
 if self.hours==23:  
 self.hours=0  
 else:  
 self.hours+=1  
 else:  
 self.minutes+=1  
 else:  
 self.seconds+=1  
"""Bu yöntem, saatin "tıklamasına" izin verir,   
bu, dahili zamanın   
bir saniye ileri alınacağı anlamına gelir."""  
Saat1=Clock(22,59,59)  
print(Saat1)  
Saat1.tick()  
print(Saat1)  
Saat2=Clock(23,59,56)  
print(Saat2)  
Saat2.tick()  
print(Saat2)

22:59:59

23:0:0

23:59:56

23:59:57

* Şimdi sıra Calendar sınıfını oluşturmada. Öncelikle artık yıl ( Leap year ) kavramında bahsedelim.
* Artık yıl, takvim yılının mevsimlerle ve dünyanın güneş çevresinde dönme süresi ile uyumlu olması için, fazladan bir gün içeren yıllara denir.
* Gregoryen takviminde fazladan bir gün 29 Şubat olarak seçilmiştir.
* Bir yılın artık yıl olabilmesi için şu şartları sağlaması gerekli :

Bir yıl 400 ile bölünüyorsa artık yıldır.

Bir yıl 400 ile bölünmüyor fakat 100 ile bölünüyorsa artık yıl değildir.

Bir yıl 4 ile bölünebiliyor fakat 100 ile bölünmüyorsa artık yıldır.

Diğer tüm yıllar ortak yıldır.

class Calendar():  
 months=[31,28,31,30,31,30,31,31,30,31,30,31]  
 date\_style="British"  
 """  
 Yılın artık yıl olup olmadığını kontrol ettiren  
 fonksiyondur.  
 Artık yıl ise True verir değil ise False verir.  
 """  
 @staticmethod  
 def leapyear(year):  
 if year %4==0:  
 return True  
 else:  
 return False  
 def \_\_init\_\_(self,d,m,y): #integer olmalı day,month,year  
 self.set\_Calendar(d,m,y)  
 def set\_Calendar(self,d,m,y):  
 #d,m,y murt be integer and year must be a four digit year number  
 if type(d)==int and type(m)==int and type(y)==int:  
 self.days=d  
 self.months=m  
 self.years=y  
 else:  
 raise TypeError("d,m,y must be integers!")  
 def \_\_str\_\_(self):  
 if Calendar.date\_style=="British":  
 return "{}/{}/{}".format(self.days,self.months,self.years)  
 else: # Assuming Americal style  
 return "{}/{}/{}".format(self.months,self.days,self.years)  
  
 def advance(self):  
 last\_days=Calendar.months[self.months-1]  
 if self.months==2 and Calendar.leapyear(self.year):  
 last\_days+=1  
 if self.days==last\_days:  
 self.days=1  
 if self.months==12:  
 self.months=1  
 self.years+=1  
 else:  
 self.months+=1  
 else:  
 self.days+=1  
  
Cal1=Calendar(12,3,2020)  
print(Cal1)  
Cal1.advance()  
print(Cal1)  
Cal2=Calendar(31,4,2034) #Nisan 30 da biter o yüzden hatalı sonuç alırız  
Cal2.advance()  
print(Cal2)  
Cal3=Calendar(30,4,2034)  
print(Cal3)  
Cal3.advance()  
print(Cal3) #1/5/2034 tarihini verir.

12/3/2020

13/3/2020

32/4/2034

30/4/2034

1/5/2034

* Sonunda, çoklu kalıtım örneğimizi tanıtacağız. Artık, hem Clock hem de Calendar'dan miras alacak olan, orijinal olarak amaçlanan CalendarClock sınıfını uygulayabiliyoruz.
* Clock'un "tick" yönteminin geçersiz kılınması (overriding) gerekecek.
* Ancak, CalendarClock'un yeni onay yöntemi, Clock: Clock.tick(self) öğesinin onay yöntemini çağırmalıdır.

from Clock import Clock  
from Calendar import Calendar  
class CalendarClock(Clock,Calendar):  
 def \_\_init\_\_(self,hours,minutes,seconds,days,months,years):  
 Clock.\_\_init\_\_(self,hours,minutes,seconds)  
 Calendar.\_\_init\_\_(self,days,months,years)  
 def tick(self):  
 #advance the clock by one second  
 previous\_hour=self.hours #previous\_hour'a başlangıç saatini atar  
 Clock.tick(self) #Saati bir saniye ilerletir  
 if self.hours<previous\_hour:  
 self.advance()  
 """previous\_hour=10:59---tick()  
 self.hour=11:00---böyle devam eder  
 previous\_hour=23:59  
 self.hour=0:0---bu geçişte takvim bir gün atar  
 self.advance()  
 """  
 def \_\_str\_\_(self):  
 return Calendar.\_\_str\_\_(self) +"," + Clock.\_\_str\_\_(self)  
if \_\_name\_\_ == "\_\_main\_\_":  
 x=CalendarClock(23,59,59,31,12,2022)  
 print("Tıklanma öncesi",x,end=" ")  
 x.tick()  
 print("Tıklanma sonrası",x)

Tıklanma öncesi 31/12/2022,23:59:59 Tıklanma sonrası 1/1/2023,0:0:0

ÇOKLU MİRAS: ÖRNEK

Python programlama dilinin bu önemli nesne yönelimli ilkesi için daha kapsamlı bir örnek sunacağız. Robot sınıfımızın bir varyasyonunu üst sınıf olarak kullanacağız. Python benzeri özelliklerle nesne yöneliminin diğer bazı önemli yönlerini de özetleyeceğiz. Ayrıca üzerine yazma, aşırı yükleme ve geçersiz kılma arasındaki farkları da çözeceğiz.

Bir Robot sınıfı tanımlayacağız taban sınıfı olarak ve ‘FightingRobot’ ve ‘NursingRobot’ alt sınıfları tanımlayacağız. Daha sonra bu iki sınıfı miras alan ‘FightingNurseRobot’ isimli çoklu miras sınıfı oluşturacağız.

“Robot” sınıfı ile başlayalım:

* **‘\_\_illegal\_names’** isimli özel sınıf niteliği ekliyoruz. Bu nitelik isim olarak kullanılması yasak birkaç tane ismi içeriyor.
* **crucial\_health\_level** isimli sınıf niteliği tanımlıyoruz ve 0.6 değerini atıyoruz.
* **“name”** metodu: yeni bir örnek oluşturulduğundaki isim “**illegal\_names**” listesinde var ise, bu ismi “**Marvin**” olarak değiştiriyoruz yoksa “name” degerini atıyoruz.
* Bir **\_\_add\_\_** yöntemi sağlayarak robotlarımızın yayılma yeteneğine sahip olduğundan emin oluruz. Ortaya çıkan robotun adı otomatik olarak oluşturulacaktır. Bir 'bebek' robotunun adı, bir tire ile ayrılmış her iki ebeveynin adlarının birleştirilmesinden oluşacaktır. Bir ebeveyn adının kısa çizgi içeren bir adı varsa, yalnızca kısa çizgiden önceki ilk kısmı kullanırız.
* Robotlar, “**health\_level**” özelliği için 0 ile 1 arasında rastgele bir değerle 'canlanacak'.
* Bir **Robot**un iyileşmeye ihtiyacı olup olmadığını belirlemek için, değer **Robot.\_\_crucial\_health\_level**'in altındaysa **True**, aksi halde **False** değerini döndüren **need\_a\_nurse** yöntemini sağlarız.
* Hem sağlık secviyesini **health\_level** ve adını **name** yazdıran bir **say\_hi()** metodu oluşturalım.

import random  
class Robot():  
 \_\_illegal\_names = {"Henry", "Oscar"}  
 \_\_crucial\_health\_level = 0.6  
 def \_\_init\_\_(self, name):  
 self.name = name # ---> property setter  
 self.health\_level = random.random()  
 @property  
 def name(self):  
 return self.\_\_name  
 @name.setter  
 def name(self, name):  
 if name in Robot.\_\_illegal\_names:  
 self.\_\_name = "Marvin"  
 else:  
 self.\_\_name = name  
 def \_\_str\_\_(self):  
 return self.name + ", Robot"  
 def \_\_add\_\_(self, other):  
 first = self.name.split("-")[0]  
 second = other.name.split("-")[0]  
 return Robot(first + "-" + second)  
 def needs\_a\_nurse(self):  
 if self.health\_level < Robot.\_\_crucial\_health\_level:  
 return True  
 else:  
 return False

def say\_hi(self):  
 print("Hi, I am " + self.name)  
 print("My health level is: " + str(self.health\_level))  
  
print(Robot("Marvin") + Robot("Enigma-Alan"))  
print(Robot("Marvin")+Robot("Enigma-Alan")+Robot("Charles-Henry"))  
first\_generation = (Robot("Marvin"),  
 Robot("Enigma-Alan"),  
 Robot("Charles-Henry"))  
  
gen1 = first\_generation # used as an abbreviation  
babies = [gen1[0] + gen1[1], gen1[1] + gen1[2]]  
babies.append(babies[0] + babies[1])  
for baby in babies:  
 baby.say\_hi()

Alt Sınıf “NursingRobot”:

* NursingRobot sınıfını oluşturarak başlayacağız.
* **\_\_init\_\_** yöntemini yeni bir **healing\_power** niteliği ile genişletiyoruz.
* **healing\_power** kavramını anlamak önemli. İlk olarak healing\_power değerinin elle girilip girilmediğini kontrol ediyoruz, girildiyse, değer olarak atanır. Girilmedi ise rastgele 0.8 ile 1 arası değer atanır.
* Genel olarak, yalnızca sağlık seviyesi 1'in altındaysa bir Robotu iyileştirmek mantıklıdır.
* “**heal**” metodunda ise iyileştirmek istediğimiz örneğin “**health\_level**” ı, eski “**health\_level**”

dan büyük ise, “self.name + " not strong enough to heal " + robo.name

” yazdırsın,

* Değil ise de “**healing\_level”** değişkenine yeni örneğin **health\_level**’ı ile **healing\_power** arasında rastgele değer atasın.

from Robot import Robot  
import random  
class NursingRobot(Robot):  
 def \_\_init\_\_(self, name="Hubert", healing\_power=None):  
 super().\_\_init\_\_(name)  
 if healing\_power is None:  
 self.healing\_power = random.uniform(0.8, 1)  
 else:  
 self.healing\_power = healing\_power  
 def say\_hi(self):  
 print("Well, well, everything will be fine ... " + self.name + " takes care of you!")  
 def say\_hi\_to\_doc(self):  
 Robot.say\_hi(self)  
 def heal(self, robo):  
 if robo.health\_level > self.healing\_power:  
 print(self.name + " not strong enough to heal " + robo.name)  
 else:  
 robo.health\_level = random.uniform(robo.health\_level, self.healing\_power)  
 print(robo.name + " has been healed by " + self.name + "!")

Marvin-Enigma, Robot

Marvin-Charles, Robot

Hi, I am Marvin-Enigma

My health level is: 0.12545339244304565

Hi, I am Enigma-Charles

My health level is: 0.9102313543244981

Hi, I am Marvin-Enigma

My health level is: 0.9249454476611294

Alt Sınıf FightingRobot:

Ne yazık ki, sanal robot dünyamız onların insan emsallerinden daha iyi değil. Başka bir deyişle, devam eden bazı savaşlar da olacak. FightingRobot adında bir sınıf oluşturmak için Robot'u bir kez daha alt sınıflıyoruz.

* **maximum\_damage** isimli sınıf özniteliği ekliyoruz.
* Robot sınıfı özniteliklerine sahip olacak aynı zamanda **fighting\_power** isimli nesne özniteliği eklenecek.
* Eğer nesne tanımlanırken **fighting\_power** değeri girilirse ona eşit olacak, girilmez ise rastgele maximum\_damage ile 1 arasında bir değer atanacak.
* **attack** metodu ekleyeceğiz. Atağı, tanımlanan bir nesne gerçekleştirecek. Yani attack metodu parametre olarak nesne alacak. Yeni nesnenin **health\_level** değeri **fighting\_power** ile çarpımına eşittir.

En sonda iki alt sınıfı miras alan bir üstsınıf oluşturalım

#An Example of Multiple Inheritance  
"""  
Aşağıdaki FightingNurseRobot sınıfının altında yatan fikir,  
hem iyileştirebilen  
hem de savaşabilen robotlara sahip olmaktır.  
"""  
from Robot import Robot  
from NursingRobot import NursingRobot  
from FightingRobot import FightingRobot  
class FightingNurseRobot(NursingRobot, FightingRobot):  
 def \_\_init\_\_(self, name, mode="nursing"):  
 super().\_\_init\_\_(name)  
 self.mode = mode # alternatively "fighting"  
 def say\_hi(self):  
 if self.mode == "fighting":  
 FightingRobot.say\_hi(self)  
 elif self.mode == "nursing":  
 NursingRobot.say\_hi(self)  
 else:  
 Robot.say\_hi(self)  
x=FightingNurseRobot("Nura","nursing")  
x.say\_hi()

MAGIC METOTLAR VE OPERATÖR OVERLOADİNG

* Magic metotlar, eğitimimizin önceki bölümlerinde zaten gördüğümüz, sabit adları olan özel yöntemlerdir.
* Bunlar, bu beceriksiz sözdizimine sahip yöntemlerdir, yani başında ve sonunda çift alt çizgi.
* Onlar hakkında konuşmak da zor. \_\_init\_\_ gibi bir yöntem adını nasıl telaffuz eder veya söylersiniz? "Alt çizgi alt çizgi init alt çizgi alt çizgi" kulağa korkunç geliyor ve neredeyse bir tekerleme gibi. "Çift alt çizgi init çift alt çizgi" çok daha iyidir, ancak ideal yol "dunder init dunder".
* Peki \_\_init\_\_ yönteminin sihri nedir? Cevap, doğrudan çağırmak zorunda değilsiniz.
* Çağrı, perde arkasında gerçekleştirilir. "x = A()" ifadesiyle bir A sınıfının x örneğini oluşturduğunuzda, Python \_\_new\_\_ ve \_\_init\_\_ için gerekli çağrıları yapacaktır.
* Eğitimimizin bu bölümünün sonuna doğru \_\_call\_\_ yöntemini tanıtacağız. Birçok yeni başlayan ve hatta Python'un ileri düzey programcıları tarafından gözden kaçırılır.
* \_\_call\_\_ yöntemi, Python programcılarının, örneklerin işlevler gibi davrandığı sınıflar yazmasına olanak tanır.
* Hem işlevlere hem de bu tür sınıfların örneklerine çağrılabilirler denir.
* Bu eğitimde operatör aşırı yükleme kavramıyla birçok kez karşılaştık. Sayısal değerler eklemek, dizileri birleştirmek veya listeleri birleştirmek için artı işaretini kullandık:

4 + 5

#### OUTPUT:

3.8 + 9

#### OUTPUT:

12.8

"Peter" + " " + "Pan"

#### OUTPUT

'Peter Pan'

[3,6,8] + [7,11,13]

#### OUTPUT:

[3, 6, 8, 7, 11, 13]

Lenght(3, “cm”) + 5 ===🡺 \_\_add\_\_(self, other)

* Kendi sınıfınız için "+" operatörünü ve diğer tüm operatörleri aşırı yüklemek bile mümkündür.
* Bunu yapmak için, temel mekanizmayı anlamanız gerekir. Her operatör işareti için özel (veya "sihirli") bir yöntem vardır.
* "+" işaretinin sihirli yöntemi \_\_add\_\_ yöntemidir. "-" için \_\_sub\_\_ vb. Biraz daha aşağıda tüm sihirli yöntemlerin tam bir listesi var.
* Mekanizma şu şekilde çalışır: "x + y" ifademiz varsa ve x, K sınıfının bir örneğiyse, Python, K'nin sınıf tanımını kontrol edecektir. K'nin \_\_add\_\_ yöntemi varsa, x.\_\_add\_\_(y) ile çağrılır, aksi takdirde bir hata mesajı alırız:

Traceback (most recent call last):

File "", line 1, in

TypeError: unsupported operand type(s) for +: 'K' and 'K'

## Overview of Magic Methods

### Binary Operators

|  |  |
| --- | --- |
| **Operator** | **Method** |
| + | object.\_\_add\_\_(self, other) |
| - | object.\_\_sub\_\_(self, other) |
| \* | object.\_\_mul\_\_(self, other) |
| // | object.\_\_floordiv\_\_(self, other) |
| / | object.\_\_truediv\_\_(self, other) |
| % | object.\_\_mod\_\_(self, other) |
| \*\* | object.\_\_pow\_\_(self, other[, modulo]) |
| << | object.\_\_lshift\_\_(self, other) |
| >> | object.\_\_rshift\_\_(self, other) |
| & | object.\_\_and\_\_(self, other) |
| ^ | object.\_\_xor\_\_(self, other) |
| | | object.\_\_or\_\_(self, other) |

### Extended Assignments

|  |  |
| --- | --- |
| **Operator** | **Method** |
| += | object.\_\_iadd\_\_(self, other) |
| -= | object.\_\_isub\_\_(self, other) |
| \*= | object.\_\_imul\_\_(self, other) |
| /= | object.\_\_idiv\_\_(self, other) |
| //= | object.\_\_ifloordiv\_\_(self, other) |
| %= | object.\_\_imod\_\_(self, other) |
| \*\*= | object.\_\_ipow\_\_(self, other[, modulo]) |
| <<= | object.\_\_ilshift\_\_(self, other) |
| >>= | object.\_\_irshift\_\_(self, other) |
| &= | object.\_\_iand\_\_(self, other) |
| ^= | object.\_\_ixor\_\_(self, other) |
| |= | object.\_\_ior\_\_(self, other) |

### Unary Operators

|  |  |
| --- | --- |
| **Operator** | **Method** |
| - | object.\_\_neg\_\_(self) |
| + | object.\_\_pos\_\_(self) |
| abs() | object.\_\_abs\_\_(self) |
| ~ | object.\_\_invert\_\_(self) |
| complex() | object.\_\_complex\_\_(self) |
| int() | object.\_\_int\_\_(self) |
| long() | object.\_\_long\_\_(self) |
| float() | object.\_\_float\_\_(self) |
| oct() | object.\_\_oct\_\_(self) |
| hex() | object.\_\_hex\_\_(self |

### Comparison Operators

|  |  |
| --- | --- |
| **Operator** | **Method** |
| < | object.\_\_lt\_\_(self, other) |
| <= | object.\_\_le\_\_(self, other) |
| == | object.\_\_eq\_\_(self, other) |
| != | object.\_\_ne\_\_(self, other) |
| >= | object.\_\_ge\_\_(self, other) |
| > | object.\_\_gt\_\_(self, other) |
|  |  |

ÖRNEK. (Lenght)

Lenght(Uzunluk) sınıfını ve kendi sınıfınız için "+" operatörünü nasıl aşırı yükleyebileceğinizi göstereceğiz. Bunu yapmak için \_\_add\_\_ yöntemini aşırı yüklememiz gerekiyor. Sınıfımız \_\_str\_\_ ve \_\_repr\_\_ metotlarını da içermektedir. Örneğin öznitelikleri “value” ve “unit” olacaktır. Başlangıçta “unit” değişkenine “metre” birimini atayalım. Bu sınıf, aşağıdaki gibi karışık birimlerle ifadeleri hesaplamamızı sağlayacaktır:

2.56 m + 3 yd + 7.8 in + 7.03 cm

**from** **unit\_conversions** **import** Length

L = Length

**print**(L(2.56,"m") + L(3,"yd") + L(7.8,"in") + L(7.03,"cm"))

* metric = {"mm" : 0.001, "cm" : 0.01, "m" : 1, "km" : 1000,"in" : 0.0254, "ft" : 0.3048, "yd" : 0.9144,"mi" : 1609.344 } şeklinde sözlük verilsin ve bu sınıf özniteliği olsun.
* convertToMeters() metodu tanımlayalım, girilen birimi “metre” ye dönüştürsün.
* \_\_add\_\_ metodu tanımlayalım, iki “Lenght” sınıfının toplanma işlemini gerçekleştirsin. (\_\_add\_\_(self,other) şeklinde)
* \_\_str\_\_ ve\_\_repr\_\_ metotları tanımlayalım, biri convertToMeters() çıktısını yazdırsın, biri de Lenght(self.value, self.unit) şeklinde çıktı versin.

class Length:  
 metric = {"mm" : 0.001, "cm" : 0.01, "m" : 1, "km" : 1000,  
 "in" : 0.0254, "ft" : 0.3048, "yd" : 0.9144,  
 "mi" : 1609.344 }  
 def \_\_init\_\_(self, value, unit = "m" ):  
 self.value = value  
 self.unit = unit  
 def ConvertToMetres(self):  
 return self.value \* Length.metric[self.unit]  
 def \_\_add\_\_(self, other):  
 l = self.ConvertToMetres()+ other.ConvertToMetres()  
 return Length(l / Length.metric[self.unit], self.unit )  
 def \_\_str\_\_(self):  
 return str(self.ConvertToMetres())  
 def \_\_repr\_\_(self):  
 return "Length(" + str(self.value) + ", '" + self.unit + "')"  
Leg1=Length(50,"cm")  
print(Leg1.ConvertToMetres())  
Leg2=Length(100,"cm")  
print(Leg1+Leg2)  
if \_\_name\_\_ == "\_\_main\_\_":  
 x = Length(4)  
 print(x)  
 y = eval(repr(x))  
 z = Length(4.5, "yd") + Length(1)  
 print(repr(z))  
 print(z)

0.5

1.5

4

Length(5.593613298337708, 'yd')

5.1148

* Genişletilmiş atamayı uygulamak için \_\_iadd\_\_ yöntemini kullanıyoruz:

def \_\_iadd\_\_(self, other):  
 l = self.ConvertToMetres() + other.ConvertToMetres()  
 self.value = l / Length.metric[self.unit]  
 return self

Şimdi aşağıdaki ödevleri yazabiliyoruz:

x += Length(1)

x += Length(4, "yd")

* Yukarıdaki örnekte "x += Length(1)" yazarak 1 metre ekledik. Bunun yerine basitçe "x += 1" yazmanın daha uygun olacağı konusunda kesinlikle bizimle hemfikir olacaksınız.
* "Length(5,"yd") + 4.8" gibi ifadeleri de benzer şekilde ele almak istiyoruz. Bu nedenle, birisi bir tür int veya float kullanırsa, sınıfımız bunu otomatik olarak "metre" olarak alır ve onu bir “Lenght” nesnesine dönüştürür.

Bu görev için \_\_add\_\_ ve \_\_iadd\_\_ yöntemimizi uyarlamak kolaydır. Tek yapmamız gereken "diğer" parametresinin türünü kontrol etmektir:

tanım \_\_add\_\_(kendi, diğer):

def \_\_add\_\_(self, other):  
 if type(other) == int or type(other) == float:  
 l = self.ConvertToMetres() + other  
 else:  
 l = self.ConvertToMetres() + other.ConvertToMetres()  
 return Length(l / Length.metric[self.unit], self.unit)

leg1=Length(56,"cm") + 1  
print(leg1)  
leg1+=2  
print(leg1)

1.56

3.56

* Birisi tam sayıları toplamaya çalışırsa ve bir süre sağ taraftan kayarsa, aynısını sol taraftan da almak isteyeceğine dair güvenli bir bahis! Eğer aşağıdaki kod satırını çalıştırırsak ne olacak ?

x = 5 + Length(3, "yd")

Aşağıdaki hatayı alacağız:

AttributeError: 'int' object has no attribute 'ConvertToMetres'

* Python bu sorun için de bir çözüm sunar. Bu \_\_radd\_\_ yöntemi.

def \_\_radd\_\_(self, other):  
 if type(other) == int or type(other) == float:  
 l = self.ConvertToMetres() + other  
 else:  
 l = self.ConvertToMetres() + other.ConvertToMetres()  
 return Length(l / Length.metric[self.unit], self.unit)

* \_\_radd\_\_ içerisinde \_\_add\_\_ metodunun kullanılması önerilir:

def \_\_radd\_\_(self, other):  
 return Length.\_\_add\_\_(self, other)

STANDART SINIFLARIN TABAN SINIF OLARAK KULLANILMASI

* ınt, float, dict veya list gibi standart sınıfları da temel sınıflar olarak kullanmak mümkündür.
* Bir push yöntemi ekleyerek liste sınıfını genişletiyoruz:

class Plist(list):  
 def \_\_init\_\_(self, l):  
 list.\_\_init\_\_(self, l)  
 def push(self, item):  
 self.append(item)  
if \_\_name\_\_ == "\_\_main\_\_":  
 x = Plist([3,4])  
 x.push(40)  
 print(x)

* Bu, daha önce tanıtılan tüm ikili ve genişletilmiş atama operatörlerinin "tersine çevrilmiş" sürümde de mevcut olduğu anlamına gelir: \_\_radd\_\_ , \_\_rsub\_\_ , \_\_rmul\_\_ etc.

ÖRNEK 1. (Kur değişikliği hesaplama)

Önceden tanımlanmış Uzunluk sınıfına benzer şekilde Ccy adında bi sınıf yazın. Ccy, çeşitli para birimlerinde değerler içermelidir, örn. "EUR", "GBP" veya "USD". Bir örnek, tutarı ve para birimini içermelidir. Bir alıştırma olarak tasarlayacağınız sınıf, en iyi aşağıdaki örnek oturumla açıklanabilir:

**Callables and Callable objects**

Öncelikle Python'da bir fonksiyon yazdığımızda aynı zamanda **'fonksiyonu çağırdığımızı'** da söylüyoruz. İkincisi, Python'da fonksiyonlar olarak adlandırılan ancak tam anlamıyla fonksiyon olmayan nesneler vardır. Farklı şekilde tanımlanan 'lambda fonksiyonları' vardır. Örneklerin 'normal' fonksiyonlar gibi çağrılabilir olduğu sınıfları tanımlamak da mümkündür. Bu, \_\_call\_\_ yöntemi isimli bir sihirli yöntem eklenerek elde edilecektir.

* \_\_call\_\_ metoduna gelmeden önce ‘callable'ın ne olduğunu bilmeliyiz.
* Genel olarak, bir "**callable**", bir fonksiyon gibi çağrılabilen ve bir fonksiyon gibi davranan bir nesnedir.
* Tüm fonksiyonlar ‘çağrılabilir’ dir. Python, ‘callable’ adlı bir fonksiyon sağlar.
* Bu fonksiyon yardımıyla bir nesnenin çağrılabilir olup olmadığını belirleyebiliriz.
* ‘callable’ fonksiyonu, bir argüman olarak iletilen nesnenin bir fonksiyon gibi çağrılıp çağrılamayacağını gösteren bir Boolean doğruluk değeri döndürür.
* Fonksiyonlara ek olarak, başka bir çağrılabilir biçimi daha gördük: sınıflar

def the\_answer(question):  
 return 42  
print("the\_answer: ", callable(the\_answer))

the\_answer: True

* \_\_call\_\_ yöntemi, sınıfın örneklerini ‘callable’ hale dönüştürmek için kullanılabilir.
* Fonksiyonlar çağrılabilir(callable) nesnelerdir. Çağrılabilir bir nesne, kullanılabilen ve bir fonksiyonmuş gibi davranan ancak bir fonksiyon olmayabilen bir nesnedir.
* \_\_call\_\_ yöntemini kullanarak, örnekleri çağrılabilir nesneler olacak şekilde sınıfları tanımlamak mümkündür.
* Örnek "bir fonksiyon gibi" olarak adlandırılırsa, yani parantezler kullanılarak \_\_call\_\_ yöntemi çağrılır. Aşağıdaki sınıf tanımı, bir sınıfı \_\_call\_\_ yöntemiyle tanımlamanın mümkün olan en basit yoludur.

class FoodSupply:  
 def \_\_call\_\_(self):  
 return "spam"  
foo = FoodSupply()  
bar = FoodSupply()  
print(foo(), bar())

spam spam

* Önceki sınıf örneği son derece basittir, ancak pratik açıdan işe yaramaz. Ne zaman sınıfın bir örneğini yaratsak, bir çağrılabilir alırız.
* Şimdi biraz daha kullanışlı olan bir sınıf tanımlayacağız. Bu örneği biraz geliştirelim:

class FoodSupply:  
 def \_\_init\_\_(self, \* incredients):  
 self.incredients = incredients  
 def \_\_call\_\_(self):  
 result = " ".join(self.incredients) + " plus delicious spam!"  
 return result  
f = FoodSupply("fish", "rice")  
print(f())  
g = FoodSupply("vegetables")  
print(g())

fish rice plus delicious spam!

vegetables plus delicious spam!

* Şimdi TriangleArea adında bir sınıf tanımlıyoruz. Bu sınıf, \_\_call\_\_method olan tek bir metoda sahiptir.
* \_\_call\_\_ yöntemi, üç kenarın uzunluğu verilmişse, keyfi bir üçgenin alanını hesaplar.

class TriangleArea:  
 def \_\_call\_\_(self, a, b, c):  
 p = (a + b + c) / 2  
 result = (p \* (p - a) \* (p - b) \* (p - c)) \*\* 0.5  
 return result  
area = TriangleArea()  
print(area(3, 4, 5))

6.0

* Burada, Örneklemeye parametreleri iletemeyiz,\_\_init\_\_ metodu tanımlamadık ve her örnek için \_\_call\_\_, üçgenin alanının değerini döndürür. Böylece her örnek alan fonksiyonu gibi davranır.
* Çok öğretici ve çok pratik olmayan iki örnekten sonra, aşağıdakilerle daha pratik bir örnek göstermek istiyoruz. Doğrusal denklemleri tanımlamak için kullanılabilecek bir sınıf tanımlıyoruz:

class StraightLines():  
 def \_\_init\_\_(self, m, c):  
 self.slope = m  
 self.y\_intercept = c  
 def \_\_call\_\_(self, x):  
 return self.slope \* x + self.y\_intercept  
#Örnekleme  
line = StraightLines(0.4, 3)  
for x in range(-5, 6):  
 print(x, line(x))

-5 1.0

-4 1.4

-3 1.7999999999999998

-2 2.2

-1 2.6

0 3.0

1 3.4

2 3.8

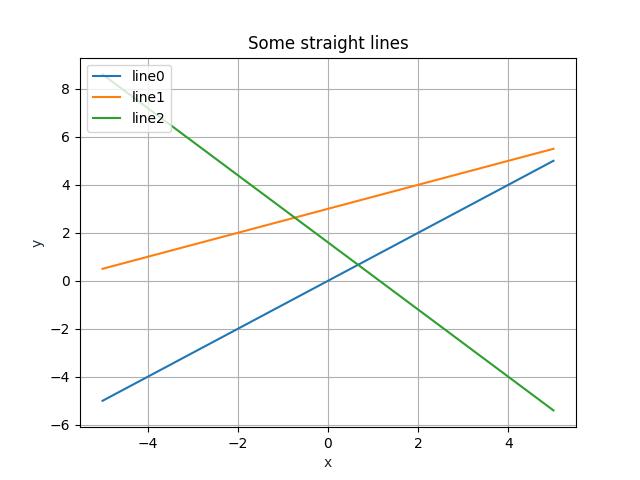
3 4.2

4 4.6

5 5.0

* Şimdi bazı düz çizgiler oluşturmak ve bunları matplotlib ile görselleştirmek için bu sınıfı kullanacağız:

lines = []  
lines.append(StraightLines(1, 0))  
lines.append(StraightLines(0.5, 3))  
lines.append(StraightLines(-1.4, 1.6))  
from matplotlib import pyplot as plt  
import numpy as np  
X = np.linspace(-5,5,100)  
for index, line in enumerate(lines):  
 line = np.vectorize(line)  
 plt.plot(X, line(X), label='line' + str(index))  
plt.title('Some straight lines')  
plt.xlabel('x', color='#1C2833')  
plt.ylabel('y', color='#1C2833')  
plt.legend(loc='upper left')  
plt.grid()  
plt.show()



* Bir sonraki örneğimiz de heyecan verici. **FuzzyTriangleArea** sınıfı, alan hesaplamalarında bulanık bir davranış uygulayan bir \_\_call\_\_ yöntemini tanımlar.
* Sonuç, p olasılığı ile doğru olmalıdır, ör. 0.8.
* Sonuç doğru değilse, sonuç ± v % aralığında olacaktır. Örneğin. 0.1.

import random  
class FuzzyTriangleArea:  
 def \_\_init\_\_(self, p=0.8, v=0.1):  
 self.p, self.v = p, v  
 def \_\_call\_\_(self, a, b, c):  
 p = (a + b + c) / 2  
 result = (p \* (p - a) \* (p - b) \* (p - c)) \*\* 0.5  
 if random.random() <= self.p:  
 return result  
 else:  
 return random.uniform(result-self.v,  
 result+self.v)  
area1 = FuzzyTriangleArea()  
area2 = FuzzyTriangleArea(0.5, 0.2)  
for i in range(12):  
 print("1.alan: {}\n2.alan: {} ".format(area1(3,4,5),area2(3,4,5)))

1.alan: 6.0

2.alan: 6.0

1.alan: 6.0

2.alan: 6.0

1.alan: 6.0

2.alan: 6.101940561020764

1.alan: 6.0

2.alan: 6.0

1.alan: 6.0986309130092975

2.alan: 6.187827438419611

1.alan: 6.0

2.alan: 5.812006557132677

1.alan: 6.0

2.alan: 6.0

1.alan: 6.0

2.alan: 5.890626487446459

1.alan: 6.0

2.alan: 5.960159837452206

1.alan: 6.0

2.alan: 5.884123715877461

1.alan: 6.0

2.alan: 6.1319921084266635

1.alan: 6.0

2.alan: 6.104986266266179

* Aşağıdaki örnek, çift terimli polinom fonksiyonları oluşturabileceğimiz bir sınıfı tanımlar:

**POLİNOM SINIFLARI**

Pythonda Polinom Fonksiyonlar

* Python'da polinom fonksiyonlarını uygulamak kolaydır. Örnek olarak bu bölümün girişinde verilen polinom fonksiyonunu tanımlıyoruz. 
* Bu polinom işlevi için Python kodu şöyle görünür:

def p(x):  
 return x\*\*4 - 4\*x\*\*2 + 3\*x  
for x in [-1, 0, 2, 3.4]:  
 print(x, p(x))

-1 -6

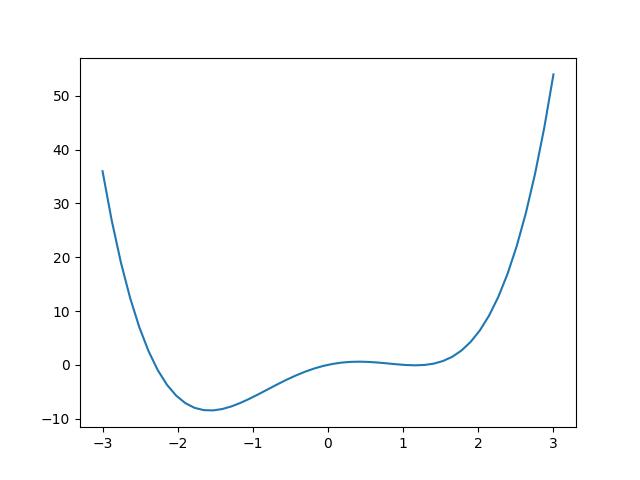
0 0

2 6

3.4 97.59359999999998

* Şimdi grafik çizdirelim:

import numpy as np  
import matplotlib.pyplot as plt  
X = np.linspace(-3, 3, 50, endpoint=True)  
F = p(X)  
plt.plot(X, F)  
plt.show()



* Şimdi polinom fonksiyonları için bir sınıf tanımlayacağız.
* Bir polinom, katsayıları tarafından benzersiz bir şekilde belirlenir. Bu, polinom sınıfımızın bir örneğinin katsayıları tanımlamak için bir listeye veya demete ihtiyacı olduğu anlamına gelir.

class Polynomial:  
 def \_\_init\_\_(self, \*coefficients):  
 """ input: coefficients are in the form a\_n, ...a\_1, a\_0 """  
 self.coefficients = list(coefficients) # tuple is turned into a list  
 def \_\_repr\_\_(self):  
 """  
 method to return the canonical string representation  
 of a polynomial."""  
 return "Polynomial" + str(tuple(self.coefficients))

p = Polynomial(1, 0, -4, 3, 0)  
print(p)

Polynomial(1, 0, -4, 3, 0)

* Bir Polinomun kurallı dizi gösterimini repr işleviyle tanımladıktan sonra, insanlar için daha kolay bir sürüm oluşturmak için kullanılabilecek bir çıktı da tanımlamak istiyoruz.
* Fonksiyonu güzel bir şekilde yazdırmak için kullanılabilecek bir LaTex temsili yazıyoruz:

class Polynomial:  
 def \_\_init\_\_(self, \*coefficients):  
 self.coefficients = list(coefficients) # tuple is turned into a list  
 def \_\_repr\_\_(self):  
 return "Polynomial" + str(tuple(self.coefficients))  
 def \_\_str\_\_(self):  
 def x\_expr(degree):  
 if degree == 0:  
 res = ""  
 elif degree == 1:  
 res = "x"  
 else:  
 res = "x^" + str(degree)  
 return res  
 degree = len(self.coefficients) - 1  
 res = ""  
 for i in range(0, degree + 1):  
 coeff = self.coefficients[i]  
 # nothing has to be done if coeff is 0:  
 if abs(coeff) == 1 and i < degree:  
 # 1 in front of x shouldn't occur, e.g. x instead of 1x  
 # but we need the plus or minus sign:  
 res += f"{'+' if coeff > 0 else '-'}{x\_expr(degree - i)}"  
 elif coeff != 0:  
 res += f"{coeff:+g}{x\_expr(degree - i)}"  
 return res.lstrip('+') # removing leading '+'  
polys = [Polynomial(1, 0, -4, 3, 0),  
 Polynomial(2, 0),  
 Polynomial(4, 1, -1),  
 Polynomial(3, 0, -5, 2, 7),  
 Polynomial(-42)]  
# output suitable for usage in LaTeX:  
for count, poly in enumerate(polys):  
 print(f"$p\_{count} = {str(poly)}$")

$p\_0 = x^4-4x^2+3x$

$p\_1 = 2x$

$p\_2 = 4x^2+x-1$

$p\_3 = 3x^4-5x^2+2x+7$

$p\_4 = -42$

* Buraya kadar polinomları tanımladık ama aslında ihtiyacımız olan şey polinom fonksiyonları.
* Bu amaçla, ‘callable’ yöntemini tanımlayarak Polinom sınıfının örneklerini çağrılabilir hale getiriyoruz.

class Polynomial:  
 def \_\_init\_\_(self, \*coefficients):  
 """ input: coefficients are in the form a\_n, ...a\_1, a\_0  
 """  
 self.coefficients = list(coefficients) # tuple is turned into a list  
 def \_\_call\_\_(self, x):  
 res = 0  
 for index, coeff in enumerate(self.coefficients[::-1]):  
 res += coeff \* x \*\* index  
 return res  
#Artık sınıfımızın bir örneğini  
# bir fonksiyon gibi çağırmak mümkün.

* Artık sınıfımızın bir örneğini bir fonksiyon gibi çağırmak mümkün. Bunu bir argümanla çağırırız ve – çağrılabilir(callable) olan - örneği bir polinom işlevi gibi davranır:

p = Polynomial(3, 0, -5, 2, 1)  
print(p)  
for x in range(-3, 3):  
 print(x, p(x))

<\_\_main\_\_.Polynomial object at 0x00000260A34E9FD0>

-3 193

-2 25

-1 -3

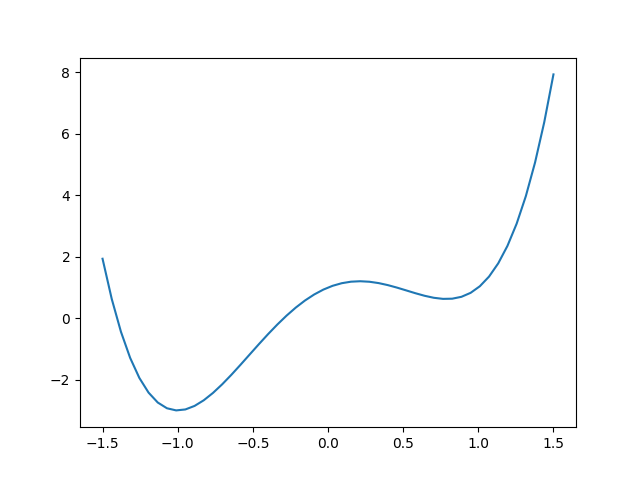
0 1

1 1

2 33

* Sadece eğlence için, önceden tanımlanmış işlevi çizelim:

import matplotlib.pyplot as plt  
import numpy as np  
X = np.linspace(-1.5, 1.5, 50, endpoint=True)  
F = p(X)  
plt.plot(X, F)  
plt.show()



* Polinomları aşağıdaki gibi farklı şekilde ifade edebiliriz ve buna polynomial2() diyelim. Polynomial() ve Polynomial2() sınıflarının deneysel olarak eşdeğer olduklarını görmek için sınıf tanımımızı yeniden yazıyoruz.
* 

class Polynomial2:  
 def \_\_init\_\_(self, \*coefficients):  
 self.coefficients = list(coefficients) # tuple is turned into a list  
 # The \_\_repr\_\_ and \_\_str\_\_ method can be included here,  
 # but is not necessary for the immediately following code  
 def \_\_call\_\_(self, x):  
 res = 0  
 for coeff in self.coefficients:  
 res = res \* x + coeff  
 return res

* Sonuçları karşılaştırarak her iki sınıfı da test edeceğiz:

p1 = Polynomial(-4, 3, 0)  
p2 = Polynomial2(-4, 3, 0)  
res = all((p1(x)==p2(x) for x in range(-10, 10)))  
print(res)

True

* Polinomlar için toplama ve çıkarmalar tanımlamak mümkündür. Tek yapmamız gereken, her iki polinomdan aynı üslü katsayıları toplamak veya çıkarmak.
* Toplama ve çıkarma için gerekli olan \_\_add\_\_ ve \_\_sub\_\_ yöntemlerini eklemeden önce, bir generatur zip\_longest() ekliyoruz.
* İki parametre için Zip'e benzer şekilde çalışır, ancak yineleyicilerden biri tükenirse durmaz, bunun yerine "dolgu değeri" kullanır.

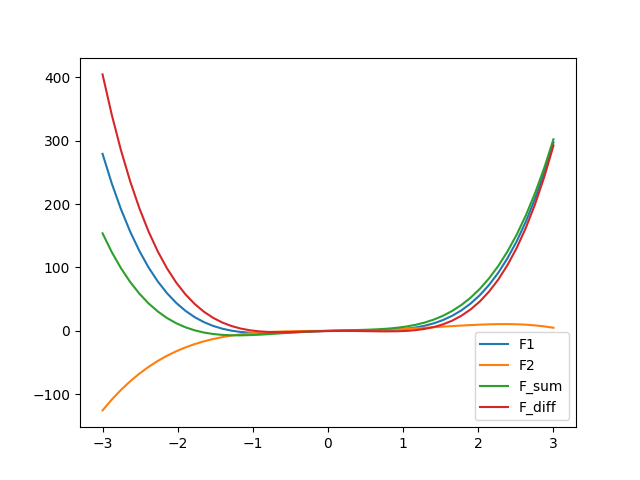
p1 = (2,)  
p2 = (-1, 4, 5)  
import itertools as itt  
for x in itt.zip\_longest(p1, p2, fillvalue=0):  
 print(x)

(2, -1)

(0, 4)

(0, 5)

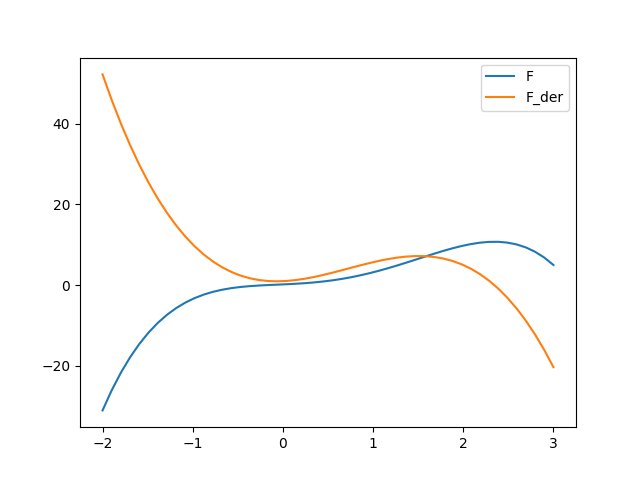
import numpy as np  
import matplotlib.pyplot as plt  
from itertools import zip\_longest  
class Polynomial:  
 def \_\_init\_\_(self, \*coefficients):  
 """ input: coefficients are in the form a\_n, ...a\_1, a\_0  
 """  
 self.coefficients = list(coefficients) # tuple is turned into a list  
 def \_\_repr\_\_(self):  
 """  
 method to return the canonical string representation  
 of a polynomial.  
 """  
 return "Polynomial" + str(self.coefficients)  
 def \_\_call\_\_(self, x):  
 res = 0  
 for coeff in self.coefficients:  
 res = res \* x + coeff  
 return res  
 def degree(self):  
 return len(self.coefficients)  
 def \_\_add\_\_(self, other):  
 c1 = self.coefficients[::-1]  
 c2 = other.coefficients[::-1]  
 res = [sum(t) for t in zip\_longest(c1, c2, fillvalue=0)]  
 return Polynomial(\*res[::-1])  
 def \_\_sub\_\_(self, other):  
 c1 = self.coefficients[::-1]  
 c2 = other.coefficients[::-1]  
  
 res = [t1 - t2 for t1, t2 in zip\_longest(c1, c2, fillvalue=0)]  
 return Polynomial(\*res[::-1])  
p1 = Polynomial(4, 0, -4, 3, 0)  
p2 = Polynomial(-0.8, 2.3, 0.5, 1, 0.2)  
  
p\_sum = p1 + p2  
p\_diff = p1 - p2  
  
X = np.linspace(-3, 3, 50, endpoint=True)  
F1 = p1(X)  
F2 = p2(X)  
F\_sum = p\_sum(X)  
F\_diff = p\_diff(X)  
plt.plot(X, F1, label="F1")  
plt.plot(X, F2, label="F2")  
plt.plot(X, F\_sum, label="F\_sum")  
plt.plot(X, F\_diff, label="F\_diff")  
  
plt.legend()  
plt.show()



*   şeklinde türevi ifade edebiliriz.

from itertools import zip\_longest  
import numpy as np  
import matplotlib.pyplot as plt  
class Polynomial:  
 def \_\_init\_\_(self, \*coefficients):  
 self.coefficients = list(coefficients) # tuple is turned into a list  
 def \_\_repr\_\_(self):  
 """  
 method to return the canonical string representation  
 of a polynomial.  
 """  
 return "Polynomial" + str(self.coefficients)  
 def \_\_call\_\_(self, x):  
 res = 0  
 for coeff in self.coefficients:  
 res = res \* x + coeff  
 return res  
 def degree(self):  
 return len(self.coefficients)  
 def \_\_add\_\_(self, other):  
 c1 = self.coefficients[::-1]  
 c2 = other.coefficients[::-1]  
 res = [sum(t) for t in zip\_longest(c1, c2, fillvalue=0)]  
 return Polynomial(\*res)  
 def \_\_sub\_\_(self, other):  
 c1 = self.coefficients[::-1]  
 c2 = other.coefficients[::-1]  
 res = [t1 - t2 for t1, t2 in zip\_longest(c1, c2, fillvalue=0)]  
 return Polynomial(\*res)  
 def derivative(self):  
 derived\_coeffs = []  
 exponent = len(self.coefficients) - 1  
 for i in range(len(self.coefficients) - 1):  
 derived\_coeffs.append(self.coefficients[i] \* exponent)  
 exponent -= 1  
 return Polynomial(\*derived\_coeffs)  
 def \_\_str\_\_(self):  
 def x\_expr(degree):  
 if degree == 0:  
 res = ""  
 elif degree == 1:  
 res = "x"  
 else:  
 res = "x^" + str(degree)  
 return res  
 degree = len(self.coefficients) - 1  
 res = ""  
 for i in range(0, degree + 1):  
 coeff = self.coefficients[i]  
 # nothing has to be done if coeff is 0:  
 if abs(coeff) == 1 and i < degree:  
 # 1 in front of x shouldn't occur, e.g. x instead of 1x  
 # but we need the plus or minus sign:  
 res += f"{'+' if coeff > 0 else '-'}{x\_expr(degree - i)}"  
 elif coeff != 0:  
 res += f"{coeff:+g}{x\_expr(degree - i)}"  
 return res.lstrip('+') # removing leading '+'  
p = Polynomial(-0.8, 2.3, 0.5, 1, 0.2)  
p\_der = p.derivative()  
X = np.linspace(-2, 3, 50, endpoint=True)  
F = p(X)  
F\_derivative = p\_der(X)  
plt.plot(X, F, label="F")  
plt.plot(X, F\_derivative, label="F\_der")  
plt.legend()  
plt.show()

* Biraz kurcalarsak;



p = Polynomial(1, 2, -3, 4, -55)  
p2 = Polynomial(1, 2, 3)  
p\_der = p.derivative()  
print(p)  
print(p\_der)  
print(p2)  
p3 = p + p2  
print(p3)

x^4+2x^3-3x^2+4x-55

4x^3+6x^2-6x+4

x^2+2x+3

-52x^4+6x^3-2x^2+2x+1

**Python ABC Modülü Kullanımı**

## Bu modül sınıfların hiyerarşisini belirler, sürdürülebilir, programcı dostu, bakımı kolay sistemler tasarlamaya yardımcı olur.

## Bu ders içerisinde, soyut temel sınıfların avantajlarını ve bunları python’un standard bir modülü olan ABC modülü ile nasıl yapabileceğimizi öğreneceğiz.

## ABC modülü, ‘Abstract base class’ kelimelerinin baş harflerinden türetilmiştir.

* Guido van Rossum tarafından eklendiğini, kullanmanın bir zorunluluk olmadığını ancak herşeyin bir obje olduğu Python dilinde bu tür yaklaşımları kullanmanın, sürdürebilir, bakımı kolay sistemler tasarlamakta çok faydalı olacağını söylemek gerekir.

## Abstract Base Class yaklaşımı, bir işlemi yapan tek bir sınıf oluşturarak, bu işlemi yapmak isteyen diğer sınıfların kendisinden miras alınmasını ister

## Oluşturulan bu tek sınıf, diğerlerine hangi methodları oluşturması konusunda bir yol haritası çizer, herhangi bir implementasyon barındırmaz. Tüm işlemler, miras aldıktan sonra yeni oluşturulan sınıf içerisinde yapılmalıdır.

## Bu tür yaklaşımlar da rahatlıkla sisteme yeni bir servis entegre etmeyi kolaylaştırır ve bunu yaparken de sizin doğru yolda ilerlemenize yardımcı olur.

## Örnek; ‘Sms’ sınıfı oluşturalım. Bir çok Sms firması ile çalışabiliriz. Bugün ‘XFirması’ ile çalışıyorum, yarın ‘YFirması’ daha iyi bir teklif verir ona geçebilirim. Onda kredim biter ‘ZFirması’na geçebilirim.

from abc import ABCMeta, abstractmethod  
class Sms(metaclass=ABCMeta):  
 def create\_token(self):pass  
 @abstractmethod  
 def phone(self):pass  
 @abstractmethod  
 def message(self):pass  
 @abstractmethod  
 def send(self):pass

## “ABC\_tutorial\_base” olarak kaydedildi.

from ABC\_tutorial\_base import Sms  
class XCompany(Sms):  
 def phone(self):  
 print("Telefon girildi")  
 def send(self):  
 print("Mesaj yollandı") def message(self):  
 print("Mesaj alındı.")  
x=XCompany() # örnek oluşturdum  
x.create\_token()  
x.send()

## Örnek. “Stranc taşları” sınıfı oluşturalım. At, fil, şah, piyon gibi satranç taşlarını bu sınıftan türetelim. Devamında satranç taşlarının nasıl hareket ettiği, tahtada kaç tane bulunduğunu gibi özelliklerini ‘abstract method’ olarak tanımlayalım.

from abc import ABCMeta,abstractmethod  
class ChessPieces(metaclass=ABCMeta):  
 @abstractmethod  
 def show\_move(self):  
 pass  
 @abstractmethod  
 def show\_count\_in\_board(self):  
 pass

from Chess import ChessPieces  
class Knight(ChessPieces):  
 def show\_count\_in\_board(self):  
 print("4 tane bulunur")  
x=Knight()  
x.show\_count\_in\_board()

TypeError: Can't instantiate abstract class Knight with abstract method show\_move

* Yukarıdaki şekilde hata elde ettik. Bunun sebebi, @abstractmethod konulan “Show\_move” metodunun ‘Knight’ sınıfında bulunmamasıdır. Aşağıdaki tanımlama ile hatadan kurtuluruz.

from Chess import ChessPieces  
class Knight(ChessPieces):  
 def show\_count\_in\_board(self):  
 print("4 tane bulunur")  
 def show\_move(self):  
 print("L şeklinde gider.")  
knight=Knight()  
knight.show\_count\_in\_board()  
knight.show\_move()

4 tane bulunur

L şeklinde gider.

* Knight, Bishop, Pawn sınıfları oluşturalım. Türettiğimiz sınıflar kendine özgü metotlar da içerebilir. Pawn() sınıfına “Show\_special\_skill” isimli bir metot ekleyelim.

from Chess import ChessPieces  
class Knight(ChessPieces):  
 def show\_count\_in\_board(self):  
 print("4 tane bulunur.")  
 def show\_move(self):  
 print("L şeklinde gider.")  
class Bishop(ChessPieces):  
 def show\_move(self):  
 print("Çarpraz gider.")  
 def show\_count\_in\_board(self):  
 print("4 tane bulunur.")  
class Pawn(ChessPieces):  
 def show\_move(self):  
 print("Düz gider.")  
 def show\_count\_in\_board(self):  
 print("16 tane bulunur.")  
 def show\_special\_skill(self):  
 print("Vezire dönüşür.")  
knight=Knight()  
knight.show\_count\_in\_board()  
knight.show\_move()  
bishop=Bishop()  
bishop.show\_count\_in\_board()  
bishop.show\_move()  
pawn=Pawn()  
pawn.show\_move()  
pawn.show\_count\_in\_board()  
pawn.show\_special\_skill()

4 tane bulunur.

L şeklinde gider.

4 tane bulunur.

Çarpraz gider.

Düz gider.

16 tane bulunur.

Vezire dönüşür.

METACLASS

**Tanım:** Nesne yönelimli programlamada, bir metasınıf, örnekleri sınıf olan bir sınıftır. Sıradan bir sınıfın belirli nesnelerin davranışını tanımlaması gibi, bir meta sınıf da belirli sınıfların ve bunların örneklerinin davranışını tanımlar. Tüm nesne yönelimli programlama dilleri meta sınıfları desteklemez.

Python'daki bir metasınıf, bir sınıfın nasıl davranacağını tanımlayan bir sınıf sınıfıdır.

Bir sınıfın kendisi bir metasınıf örneğidir.

Python'da bir sınıf, sınıfın örneğinin nasıl davranacağını tanımlar.

Metasınıfları iyi anlamak için Python sınıflarıyla çalışma konusunda önceden deneyime sahip olmak gerekir.

Metasınıflara daha derinlemesine dalmadan önce, birkaç kavramı aradan çıkaralım.

**Python'da her şey bir Nesnedir**

class TestClass():  
 pass  
my\_test\_class=TestClass() # nesne oluşturduk  
print(my\_test\_class)

<\_\_main\_\_.TestClass object at 0x000002AF86942EB0>

**Python Sınıfları Dinamik Olarak Oluşturulabilir**

Python'da ‘**type’**, bir nesnenin türünü bulmamızı sağlar. Yukarıda oluşturduğumuz nesne tipini kontrol etmeye devam edebiliriz.

print(type(TestClass))

<class 'type'>

print(type(type))

<class 'type'>

* Bekle, az önce ne oldu?
* Yukarıda yarattığımız nesnenin türünün class olmasını beklerdik ama değil.
* Bu düşünceye tutun. Birkaçını daha fazla ele alacağız.
* Ayrıca type tipinin de type olduğunu fark ederiz. Bu bir type örneğidir. ‘type’'ın yaptığı bir başka sihirli şey de dinamik olarak sınıflar oluşturmamızı sağlamaktır. Bunu nasıl yapacağımızı aşağıda gösterelim.
* Aşağıda gösterilen **DataCamp** sınıfı, ‘type’ kullanılarak aşağıda gösterildiği gibi oluşturulacaktır:

class DataCamp():  
 pass  
DataCampClass = type('DataCamp', (), {})  
print(DataCampClass)  
print(DataCamp())

<class '\_\_main\_\_.DataCamp'>

<\_\_main\_\_.DataCamp object at 0x000001B266269FA0>

* Yukarıdaki örnekte DataCamp, sınıf adı iken DataCampClass, sınıf referansını tutan değişkendir.
* **‘type’** kullanırken, aşağıda gösterildiği gibi bir sözlük kullanarak sınıfın özniteliklerini iletebiliriz:

PythonClass = type('PythonClass', (), {'start\_date': 'August 2018', 'instructor': 'John Doe'} )  
print(PythonClass.start\_date, PythonClass.instructor)  
print(PythonClass)

August 2018 John Doe

<class '\_\_main\_\_.PythonClass'>

* PythonClass'ımızın DataCamp sınıfından miras almasını istiyorsak, type kullanarak sınıfı tanımlarken onu ikinci argümanımıza iletiyoruz.

PythonClass = type('PythonClass', (DataCamp,), {'start\_date': 'August 2018', 'instructor': 'John Doe'} )

print(PythonClass)

<class '\_\_main\_\_.PythonClass'>

* Artık bu iki kavram yoldan çıktığına göre, Python'un sınıfları bir metasınıf kullanarak oluşturduğunu anlıyoruz.
* Python'da her şeyin bir nesne olduğunu gördük, bu nesneler metasınıflar tarafından oluşturulur.
* Ne zaman bir sınıf yaratmak için class çağırsak, perde arkasında sınıfı yaratma sihrini yapan bir metasınıf vardır. Dizeleri oluşturan **str**'ye ve tamsayılar oluşturan **int**'ye benzer.
* Python'da, **\_\_\_class\_\_**özniteliği mevcut örneğin tipini kontrol etmemizi sağlar.
* Aşağıda bir string oluşturalım ve tipini kontrol edelim.

article = 'metaclasses'  
print(article.\_\_class\_\_)

<class 'str'>

* Ayrıca **type(article)** kullanarak ‘type’’ı kontrol edebiliriz.

print(type(article))

<class 'str'>

* str tipini kontrol ettiğimizde **‘type’** olduğunu da öğreniyoruz.

print(type(str))

<class 'type'>

* Float, int, list, Tuple ve dict tipini kontrol ettiğimizde benzer bir çıktı elde edeceğiz.
* Bunun nedeni, tüm bu nesnelerin ‘type’ türünde olmasıdır.

print(type(list),type(float), type(dict), type(tuple))

<class 'type'> <class 'type'> <class 'type'> <class 'type'>

* ‘type’ın sınıflar oluşturduğunu zaten gördük. Bu nedenle**, \_\_class\_\_** öğesinin **\_\_class\_\_** öğesini kontrol ettiğimizde, ‘**type**’ döndürmelidir.

print(article.\_\_class\_\_.\_\_class\_\_)

<class 'type'>

**Özel Meta Sınıflar Oluşturma**

Python'da, sınıf tanımında metaclass anahtar sözcüğünü ileterek sınıf oluşturma sürecini özelleştirebiliriz. Bu, bu anahtar kelimeyi daha önce kullanmış bir sınıfı devralarak da yapılabilir.

class MyMeta(type):  
 pass

class MyClass(metaclass=MyMeta):  
 pass

class MySubclass(MyClass):  
 pass

print(type(MyMeta))  
print(type(MyClass))  
print(type(MySubclass))

* Aşağıda MyMeta sınıfının tipinin type olduğunu ve MyClass ve MySubClass tipinin MyMeta olduğunu görebiliriz.

<class 'type'>

<class '\_\_main\_\_.MyMeta'>

<class '\_\_main\_\_.MyMeta'>

* Bir sınıf tanımlarken ve hiçbir metasınıf tanımlanmadığında, default type metaclass kullanılacaktır.
* Bir metasınıf verilmişse ve bu bir type() örneği değilse, o zaman doğrudan metasınıf olarak kullanılır.

**\_\_new\_\_ and \_\_init\_\_\_**

Metasınıflar ayrıca aşağıda gösterilen iki yoldan biriyle tanımlanabilir. Aralarındaki farkı aşağıda açıklayacağız.

class MetaOne(type):  
 def \_\_new\_\_(cls, name, bases, dict):  
 pass  
class MetaTwo(type):  
 def \_\_init\_\_(self, name, bases, dict):  
 pass

* \_\_new\_\_, sınıf oluşturulmadan önce dict veya base tuple'ları tanımlamak istendiğinde kullanılır.
* \_\_new\_\_'nin dönüş değeri genellikle bir cls örneğidir. \_\_new\_\_, değişmez türlerin alt sınıflarının örnek oluşturmayı özelleştirmesine izin verir.
* Sınıf oluşturmayı özelleştirmek için özel metasınıflarda geçersiz kılınabilir. \_\_init\_\_, genellikle nesneyi başlatmak için oluşturulduktan sonra çağrılır.

REALPYTHTON METACLASS

New Style Class

class Foo:

pass

obj = Foo()

obj.\_\_class\_\_

type(obj)

obj.\_\_class\_\_ is type(obj)

class Foo:

pass

obj = Foo()

obj.\_\_class\_\_

type(obj)

obj.\_\_class\_\_ is type(obj)

Type and Class

Remember that, [in Python, everything is an object.](https://web.archive.org/web/20151210024637/https:/mail.python.org/pipermail/python-list/2015-June/691689.html) Classes are objects as well. As a result, a class must have a type. What is the type of a class?

Consider the following:

class Foo:

pass

x = Foo()

type(x)

type(Foo)

The type of x is class Foo, as you would expect. But the type of Foo, the class itself, is type. In general, the type of any new-style class is type.

The type of the built-in classes you are familiar with is also type:

for t in int, float, dict, list, tuple:

print(type(t))

For that matter, the type of type is type as well (yes, really):

type(type)

type is a metaclass, of which classes are instances. Just as an ordinary object is an instance of a class, any new-style class in Python, and thus any class in Python 3, is an instance of the type metaclass.

In the above case:

* x is an instance of class Foo.
* Foo is an instance of the type metaclass.
* type is also an instance of the type metaclass, so it is an instance of itself

## Defining a Class Dynamically

The built-in type() function, when passed one argument, returns the type of an object. For new-style classes, that is generally the same as the [object’s \_\_class\_\_ attribute](https://docs.python.org/3/library/stdtypes.html" \l "instance.__class__):

type(3)

type(['foo', 'bar', 'baz'])

t = (1, 2, 3, 4, 5)

type(t)

class Foo:

pass

type(Foo())

You can also call type() with three arguments—type(<name>, <bases>, <dct>):

* <name> specifies the class name. This becomes the \_\_name\_\_ attribute of the class.
* <bases> specifies a tuple of the base classes from which the class inherits. This becomes the \_\_bases\_\_ attribute of the class.
* <dct> specifies a [namespace dictionary](https://realpython.com/python-namespaces-scope/" \l "python-namespace-dictionaries) containing definitions for the class body. This becomes the \_\_dict\_\_ attribute of the class.

Calling type() in this manner creates a new instance of the type metaclass. In other words, it dynamically creates a new class.

### **Example 1**

In this first example, the <bases> and <dct> arguments passed to type() are both empty. No [inheritance](https://realpython.com/inheritance-composition-python/) from any parent class is specified, and nothing is initially placed in the namespace dictionary. This is the simplest class definition possible:

Foo = type('Foo', (), {})

x = Foo()

x

class Foo:

pass

x = Foo()

x

### **Example 2**

Here, <bases> is a tuple with a single element Foo, specifying the parent class that Bar inherits from. An attribute, attr, is initially placed into the namespace dictionary:

Bar = type('Bar', (Foo,), dict(attr=100))

x = Bar()

x.attr

x.\_\_class\_\_

x.\_\_class\_\_.\_\_bases\_\_

class Bar(Foo):

attr = 100

x = Bar()

x.attr

x.\_\_class\_\_

x.\_\_class\_\_.\_\_bases\_\_

### **Example 3**

This time, <bases> is again empty. Two objects are placed into the namespace dictionary via the <dct> argument. The first is an attribute named attr and the second a function named attr\_val, which becomes a method of the defined class:

Foo = type(

'Foo',

(),

{

'attr': 100,

'attr\_val': lambda x : x.attr

}

)

x = Foo()

x.attr

x.attr\_val()

class Foo:

attr = 100

def attr\_val(self):

return self.attr

x = Foo()

x.attr

x.attr\_val()

**Metasınıf Kullanarak Singleton Tasarımı**

Singleton(tekil kalıp), yazılım mühendisliğinde bir sınıf tipinden sadece tek bir nesnenin yaratılmasına izin veren bir yazılım tasarım örüntüsüdür. Bu kavram bazen daha da geliştirilerek sınıfın belirli sayıda nesnesinin yaratılması şeklinde genelleştirilebilir. Örneğin, oluşturulacak nesne sayısı beş ile sınırlanabilir.