**1. Python Inheritance (Miras/Kalıtım) Nedir?**

[Sıfırdan Python Dersleri Ders 22: OOP Inheritance | Kalıtım - 1](https://www.youtube.com/watch?v=yNMpXv-WuDc)

Büyük projelerde çalıştığınız zaman projenin alt modül yani alt parçacıklardan oluştuğunu ve bu alt modüllerin birçok özelliklerinin ana modülde zaten olduğunu göreceksiniz. Bunun için her seferinde alt modüller için, özellikleri ve işlevleri oluşturmanız gerekecek buda sizin için resmen bir işkenceye dönüşüyor olacak. İşte bunun için inheritance(miras alma) kavramı ortaya çıkıyor.

Inheritance Türkçe karşılığı kalıtım veya miras alma olan bu kavram nesne tabanlı programlama(OOP) ile gelen bir kavramdır. Kalıtım, bir sınıfın diğer bir sınıftan özelliklerini ve davranışlarını miras almasıdır. Miras alan sınıf, miras veren sınıfın tüm niteliklerine ve metodlarına erişebilir ve aynı zamanda kendi özel niteliklerini ve metodlarını da tanımlayabilir. Bu, kodun yeniden kullanılabilirliğini artırır ve sınıflar arasında hiyerarşik bir ilişki kurulmasını sağlar. Nesne tabanlı programlamayı anlatırken biz her nesne için yeni bir sınıf oluşturuyorduk ve her nesnenin özellik ve işlevlerini belirtiyorduk.  İşte burada çok daha pratik olan bir yöntem olan inheritance ile artık her nesne için ayrıca özellik ve işlev belirtmeniz gerek kalmıyor. Bir defa oluşturduğunuz özellikleri tekrar tekrar kullanıyorsunuz.

Hemen günlük hayattan bir örnek verelim. Örneğin elinizde bir taşıt nesnesi var; siz araba, motosiklet gibi nesneler oluşturacaksınız. Burada araba ve motor için tekrar özellikleri yazmanıza gerek yok çünkü araba ve motosiklet nesnesinin özellikleri zaten taşıt nesnesinin içinde var. Sebebi ise araba ve motosiklet nesneleri taşıt nesnesinin birer alt kümesi.  Kafanız karıştıysa hiç merak etmeyin örnekleri yaptığımız zaman çok daha iyi anlayacaksınız. Yukarıdaki en son araba sınıfını miras olarak başka bir sınıfa miras olarak aktaralım.

**class TicariAraba(Araba):**

**def \_\_init\_\_(self, renk, marka, model, yuk\_kapasitesi):**

**super().\_\_init\_\_(renk, marka, model)**

**self.yuk\_kapasitesi = yuk\_kapasitesi**

**def yuk\_tasi(self):**

**print(f"{self.marka} {self.model}, {self.yuk\_kapasitesi} ton yük taşıyor.")**

**ticari\_araba = TicariAraba("Beyaz", "Ford", "Transit", 3)**

**print(ticari\_araba.renk)  # Çıktı: Beyaz**

**ticari\_araba.hizlan(40)   # Çıktı: Ford Transit, 40 km/s hızlandı.**

**ticari\_araba.yuk\_tasi()   # Çıktı: Ford Transit, 3 ton yük taşıyor.**

Örneklerimizi çeşitlendirelim. Birçoğumuzun üniversite veya lisede gördüğü öğrenci bilgilendirme veya veli bilgilendirme sistemleri vardır. Bu sistemlerin birden fazla kullanıcı giriş sistemi vardır. Aşağıda örnek olarak bir üniversitenin öğrenci bilgilendirme sistemini ele alalım.



Gördüğünüz gibi sisteme birden fazla farklı kullanıcı girişi var ve bu kullanıcıların yaptıkları işlemlerin birçoğu hemen hemen aynı. Şimdi biz bu kullanıcıları için kodları yazalım.

**class Akademisyen:**

**def \_\_init\_\_(self, adi, soyadi, numarasi):**

**self.adi = adi**

**self.soyadi = soyadi**

**self.numarasi = numarasi**

**def giris(self):**

**print("Giriş Yapıldı")**

**def cikis(self):**

**print("Çıkış yapıldı")**

**class Personel:**

**def \_\_init\_\_(self, adi, soyadi, numarasi):**

**self.adi = adi**

**self.soyadi = soyadi**

**self.numarasi = numarasi**

**def giris(self):**

**print("Giriş Yapıldı")**

**def cikis(self):**

**print("Çıkış yapıldı")**

**class Ogrenci:**

**def \_\_init\_\_(self, adi, soyadi, numarasi):**

**self.adi = adi**

**self.soyadi = soyadi**

**self.numarasi = numarasi**

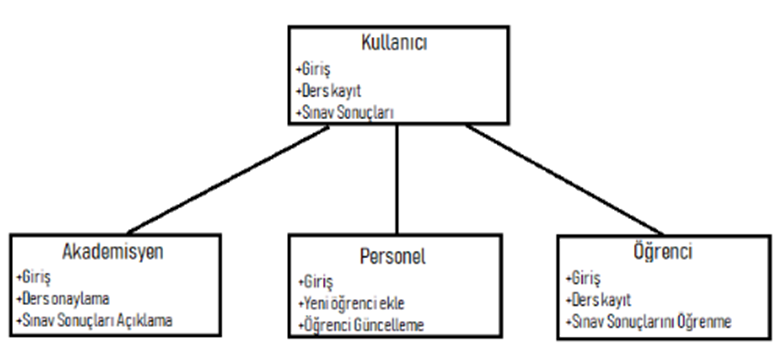
**def giris(self):**

**print("Giriş Yapıldı")**

**def cikis(self):**

**print("Çıkış yapıldı")**

Fark ettiyseniz OOP kullanmamıza rağmen yine aynı kodları tekrar tekrar yazmak zorunda kaldık. Şimdi birde miras alma kavramı ile yapalım.



Görsele baktığımız zaman Kullanıcı adında bir nesnemiz var bu nesnenin akademisyen, personel ve öğrenci adında alt nesneleri var.  Kullanıcı nesnesinin özellikleri diğer tüm alt nesnelerde var. Yani bu alt nesneler ana nesne olan kullanıcı nesnesinden türemiş, diğer bir adıyla miras alınmış hemen kodlarını yazalım o zaman çok daha iyi anlayacaksınız.

**Miras Alma Kullanımı**

Miras alma yöntemi çok kolaydır. Genel kullanım şekli şu şekildedir.

**Class sinif\_adi(miras\_alinacak\_sinif\_adi):**

**……….**

 Şimdi kodlarımızı yazalım;

**class Kullanici:**

**def \_\_init\_\_(self ,adi, soyadi, numara):**

**print("Kullanıcı sınıfı fonksiyonu")**

**self.adi = adi**

**self.soyadi = soyadi**

**self.numara = numara**

**def giris(self):**

**print("Giriş Yapıldı")**

**def cikis(self):**

**print("Çıkış yapıldı")**

**class Akademisyen(Kullanici):**

**pass**

**class Personel(Kullanici):**

**pass**

**class Ogrenci(Kullanici):**

**pass**

Kullanıcı isimli sınıfımızı oluşturduk ve diğer nesneleri bu sınıftan miras aldık. Alt sınıflar için özellik ve işlev eklemedik çünkü biz özellik ve işlevleri kullanıcı sınıfından miras alacağız hemen gösterelim;

Şimdi objeyi oluşturup çağıralım.

**akademisyen = Akademisyen("Mustafa", "Kaya", 1236521)**

**personel = Personel("Mehmet", "Yıldız", 1539527)**

**ogrenci = Ogrenci("Can", "Demir", 1436589)**

**print("Akademisyen")**

**print(akademisyen.adi)**

**print(akademisyen.soyadi)**

**print(akademisyen.numara)**

**print("Personel")**

**print(personel.adi)**

**print(personel.soyadi)**

**print(personel.numara)**

**print("Öğrenci")**

**print(ogrenci.adi)**

**print(ogrenci.soyadi)**

**print(ogrenci.numara)**

Çıktıya bakalım;

**Akademisyen       Personel               Öğrenci**

**Mustafa                 Mehmet                Can**

**Kaya                       Yıldız                     Demir**

**1236521               1539527               1436589**

Gördüğünüz gibi kullanıcı nesnesindeki özellikleri ve işlevleri istediğiniz gibi çağırıp kullanabiliyorsunuz. Şimdi aklımıza bir de şu şekilde bir soru gelmiştir; ben miras aldığım nesnenin özelliklerini kullanabiliyorum ama ek özellik ve işlev eklemek istediğimde ne yapacağım?

[Sıfırdan Python Dersleri Ders 23: OOP Inheritance | Kalıtım - 2](https://www.youtube.com/watch?v=d4uTusVCoEo)

**Python Overriding**

Overriding Türkçe karşılığı üstüne yazmak anlamına geliyor. Biz sınıfları işlediğimiz zaman yapıcı olan fonksiyon \_\_init\_\_() fonksiyonu için oluşturduğumuz sınıfın referansı demiştik ve bu fonksiyon otomatik olarak çalışıyordu. Onun için biz yeni sınıfımızda miras aldığımız fonksiyonun özelliklerini değil yeni özellikleri kullanmak istediğimiz zaman yeni sınıf için \_\_init\_\_() fonksiyonu tanımlamamız gerekiyor. Hemen gösterelim.

**class Kullanici:**

**def \_\_init\_\_(self, adi, soyadi, numara):**

**print("Kullanıcı sınıfı fonksiyonu")**

**self.adi = adi**

**self.soyadi = soyadi**

**self.numara = numara**

**def giris(self):**

**print("Giriş Yapıldı")**

**def cikis(self):**

**print("Çıkış yapıldı")**

**class Akademisyen(Kullanici):**

**def \_\_init\_\_(self, adi, soyadi, numara, dogum\_tarihi):**

**print("Akademisyen sınıfı fonksiyonu")**

**self.adi= adi**

**self.soyadi=soyadi**

**self.numara= numara**

**self.dugum\_tarihi = dogum\_tarihi**

**akademisyen = Akademisyen("Mustafa", "Kaya", 1236521, 1997)**

**print(akademisyen.adi)**

**akademisyen.giris()**

Kodumuzu çalıştıralım;

**Akademisyen sınıfı fonksiyonu**

**Mustafa**

**Giriş Yapıldı**

Şeklinde bir çıktı aldık. Biz artık Kullanıcı sınıfının \_\_init\_\_()  fonksiyonu değil akademisyen sınıfının \_\_init\_\_() fonksiyonunu kullanıyoruz. Ama Kullanıcının diğer işlevlerini yani fonksiyonlarını da kullanabiliyoruz. Şimdi siz diyeceksiniz  hani OOP ve miras alma(inheritance) kod tekrarını önlemek ve az kod yazmak için kullanıyorduk.

Kullanıcı içinde oluşturduğumuz özellikleri tekrar oluşturduk. Bunun için bir çözüm yok mu elbette var.

**super()**

Türkçe karşılığı üst olan bu fonksiyon içinde bulunduğu sınıfın üst bir sınıftan miras alındığını ve oradaki özellikleri kullanabilmenizi sağlıyor.

Hemen gösterelim;

Genel kullanımı şu şekildedir;

**super().\_\_init\_\_(ust\_sinif\_parametreleri)**

Hemen örneğimizi yapalım;

**class Kullanici:**

**def \_\_init\_\_(self, adi, soyadi, numara):**

**print("Kullanıcı sınıfı fonksiyonu")**

**self.adi = adi**

**self.soyadi = soyadi**

**self.numara = numara**

**def giris(self):**

**print("Giriş Yapıldı")**

**def cikis(self):**

**print("Çıkış yapıldı")**

**class Akademisyen(Kullanici):**

**def \_\_init\_\_(self, adi, soyadi, numara, dogum\_tarihi):**

**print("Akademisyen sınıfı fonksiyonu")**

**super().\_\_init\_\_(adi, soyadi, numara)**

**self.dogum\_tarihi = dogum\_tarihi**

**akademisyen = Akademisyen("Mustafa", "Kaya", 1236521, 1997)**

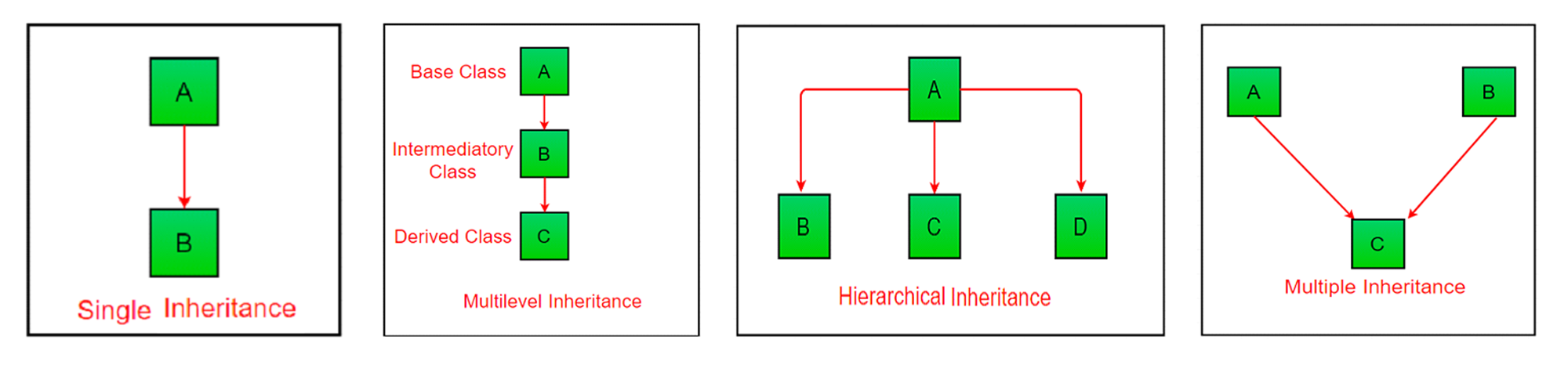
Burada yukarıda oluşturduğumuz ad, soyad ve numara özelliklerini tekrar oluşturmadık. super() fonksiyonu ile miras aldık.  Kodları çalıştıralım;

**Akademisyen sınıfı fonksiyonu**

**Kullanıcı sınıfı fonksiyonu**

Gördüğünüz gibi hem kullanıcı sınıfının \_\_init\_\_() fonksiyonu çalıştı hem de Akademisyen sınıfının yapıcı metodu \_\_init\_\_() fonksiyonu çalıştı.

**Kalıtım Türleri:**



**a.   Single Inheritance:**

**Single inheritance (tek kalıtım),** bir sınıfın yalnızca bir tane başka bir sınıftan kalıtım alabileceği bir kalıtım türüdür. Bu durumda, alt sınıf (türetilmiş sınıf), yalnızca bir üst sınıfın (temel sınıf) özelliklerini ve davranışlarını miras alır.

Single inheritance'ı daha iyi anlamak için basit bir örnek verelim. Örneğin, bir hayvanlar sınıfı düşünelim ve bu sınıfta genel özellikler ve davranışlar bulunsun. Ardından, bu hayvanlar sınıfından türeyen bir alt sınıf olan köpekler sınıfını ele alalım. Köpekler, hayvanlar sınıfının özelliklerini ve davranışlarını miras alır, ancak aynı zamanda kendilerine özgü bazı özelliklere ve davranışlara da sahip olabilirler.

Hayvanlar sınıfı, tüm hayvanların ortak özelliklerini ve davranışlarını içerebilir. Örneğin, tüm hayvanların bir isimleri olabilir, beslenme şekilleri olabilir, belirli sesler çıkarabilirler ve hareket edebilirler. Bu sınıf, temel sınıf olarak adlandırılır ve köpekler sınıfı tarafından kalıtım alınacak olan sınıftır.

Köpekler sınıfı, hayvanlar sınıfından kalıtım alır ve bu şekilde hayvanlar sınıfının tüm özelliklerini ve davranışlarını miras alır. Örneğin, köpeklerin de bir ismi olabilir, beslenme şekilleri olabilir, sesler çıkarabilirler ve hareket edebilirler. Ancak köpekler, aynı zamanda diğer hayvanlardan farklı olarak havlayabilirler, kuyruk sallayabilirler ve insanlara sadık olabilirler gibi ek özelliklere ve davranışlara da sahip olabilirler.

Bu örnekte, hayvanlar sınıfı temel sınıftır ve köpekler sınıfı türetilmiş sınıftır. Köpekler sınıfı, hayvanlar sınıfının özelliklerini ve davranışlarını kalıtım yoluyla alır ve kendi özelliklerini ve davranışlarını ekler. Bu, single inheritance'ın bir örneğidir çünkü köpekler sınıfı yalnızca bir üst sınıf olan hayvanlar sınıfından kalıtım almaktadır.

Bu şekilde, single inheritance programlamada kod tekrarını azaltır ve daha düzenli ve sürdürülebilir bir kod oluşturmanıza olanak tanır. Aynı zamanda, sınıf hiyerarşisini oluşturarak farklı sınıflar arasında ilişkiler kurmanıza olanak sağlar.

**class Hayvanlar:**

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

**self.isim = isim**

**def hareket\_et(self):**

**print("Hareket ediliyor.")**

**def beslen(self):**

**print("Besleniliyor.")**

**class Kopekler(Hayvanlar):**

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

**super().\_\_init\_\_(isim)**

**def havla(self):**

**print("Hav hav!")**

**def kuyruk\_salla(self):**

**print("Kuyruk sallanıyor.")**

**def sadik\_ol(self):**

**print("Sahibine sadık olunuyor.")**

**# Hayvanlar sınıfından miras alan Kopekler sınıfı**

**# Özelliklerini ve davranışlarını gösterelim.**

**hayvan = Hayvanlar("Genel Hayvan")**

**hayvan.hareket\_et()  # Hareket ediliyor.**

**hayvan.beslen()      # Besleniliyor.**

**kopek = Kopekler("Karabaş")**

**kopek.hareket\_et()   # Hareket ediliyor.**

**kopek.beslen()       # Besleniliyor.**

**kopek.havla()        # Hav hav!**

**kopek.kuyruk\_salla() # Kuyruk sallanıyor.**

**kopek.sadik\_ol()     # Sahibine sadık olunuyor.**

Yukarıdaki örnekte, **Hayvanlar** adında bir üst sınıf (temel sınıf) tanımlanmıştır. Bu sınıf, **isim**, **hareket\_et()**ve **beslen()** gibi genel özelliklere ve davranışlara sahiptir.

**Kopekler** sınıfı ise **Hayvanlar** sınıfından miras almıştır (**Hayvanlar** sınıfı parantez içinde belirtilmiştir). **Kopekler** sınıfı, **Hayvanlar** sınıfının tüm özelliklerini ve davranışlarını miras alırken, kendine özgü **havla(), kuyruk\_salla() ve sadik\_ol()** gibi ek özellikler ve davranışlar eklemiştir.

Sonuç olarak, yukarıdaki kod örneğinde Hayvanlar sınıfından bir nesne (hayvan) ve Kopekler sınıfından bir nesne (kopek) oluşturulmuştur. Her bir nesne, kendi sınıflarına ait yöntemleri çağırabilir ve özelliklerini kullanabilir.

**b.   Multilevel Inheritance**

**Multilevel inheritance (çok seviyeli kalıtım),** birden fazla kalıtım seviyesine sahip olan bir kalıtım türüdür. Bu durumda, bir alt sınıfın (türetilmiş sınıf), bir üst sınıftan (temel sınıf) kalıtım aldığı ve aynı zamanda başka bir sınıfın da alt sınıfı olabileceği bir hiyerarşi oluşur.

Multilevel inheritance'ı daha iyi anlamak için basit bir örnek verelim. Örneğin, bir "Şekil" sınıfı düşünelim. Bu sınıf, genel şekillerin özelliklerini ve davranışlarını içerebilir. Ardından, "Dikdörtgen" adında bir alt sınıf oluşturalım, bu sınıf "Şekil" sınıfından kalıtım alsın. Son olarak, "Kare" adında bir sınıf oluşturalım, bu sınıf da "Dikdörtgen" sınıfından kalıtım alsın.

"Şekil" sınıfı, genel şekillerin özelliklerini ve davranışlarını içerir. Örneğin, bir şeklin rengi olabilir veya alanını hesaplayabilir. Bu sınıf, temel sınıf olarak adlandırılır.

"Dikdörtgen" sınıfı, "Şekil" sınıfından kalıtım alır ve bu şekilde "Şekil" sınıfının özelliklerini ve davranışlarını miras alır. Örneğin, bir dikdörtgenin genişliği ve yüksekliği olabilir, alanını hesaplayabilir ve çizilebilir. Bu sınıf, "Şekil" sınıfından türetilen bir alt sınıftır.

"Kare" sınıfı ise "Dikdörtgen" sınıfından kalıtım alır ve aynı zamanda "Dikdörtgen" sınıfının alt sınıfıdır. Bu durumda, "Kare" sınıfı, "Dikdörtgen" sınıfının özelliklerini ve davranışlarını miras alırken, kendine özgü ek özelliklere ve davranışlara da sahip olabilir. Örneğin, bir karenin kenar uzunluğu olabilir ve "Kare" şekli olarak çizilebilir.

Bu örnekte, "Şekil" sınıfı temel sınıftır, "Dikdörtgen" sınıfı türetilmiş bir sınıftır ve "Kare" sınıfı ise hem "Dikdörtgen" sınıfından türetilmiş bir alt sınıftır hem de "Şekil" sınıfından kalıtım almıştır.

Bu şekilde, multilevel inheritance programlamada farklı kalıtım seviyeleri arasında bir hiyerarşi oluşturarak kodun daha düzenli ve sürdürülebilir olmasını sağlar.

**class Sekil:**

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

**self.renk = renk**

**def alan\_hesapla(self):**

**print("Alan hesaplanıyor.")**

**def ciz(self):**

**print("Şekil çiziliyor.")**

**class Dikdortgen(Sekil):**

**def \_\_init\_\_(self, renk, genislik, yukseklik):**

**super().\_\_init\_\_(renk)**

**self.genislik = genislik**

**self.yukseklik = yukseklik**

**def alan\_hesapla(self):**

**alan = self.genislik \* self.yukseklik**

**print(f"Dikdörtgen alanı: {alan}")**

**def ciz(self):**

**print("Dikdörtgen çiziliyor.")**

**class Kare(Dikdortgen):**

**def \_\_init\_\_(self, renk, kenar\_uzunlugu):**

**super().\_\_init\_\_(renk, kenar\_uzunlugu, kenar\_uzunlugu)**

**def alan\_hesapla(self):**

**alan = self.genislik \*\* 2**

**print(f"Kare alanı: {alan}")**

**def ciz(self):**

**print("Kare çiziliyor.")**

**# Sekil sınıfından miras alan Dikdortgen sınıfı,**

**# Dikdortgen sınıfından miras alan Kare sınıfı**

**# Özelliklerini ve davranışlarını gösterelim.**

**sekil = Sekil("Mavi")**

**sekil.alan\_hesapla()  # Alan hesaplanıyor.**

**sekil.ciz()          # Şekil çiziliyor.**

**dikdortgen = Dikdortgen("Yeşil", 5, 10)**

**dikdortgen.alan\_hesapla()  # Dikdörtgen alanı: 50**

**dikdortgen.ciz()          # Dikdörtgen çiziliyor.**

**kare = Kare("Kırmızı", 7)**

**kare.alan\_hesapla()  # Kare alanı: 49**

**kare.ciz()          # Kare çiziliyor.**

Yukarıdaki örnekte, **Sekil** adında bir üst sınıf (temel sınıf) tanımlanmıştır. Bu sınıf, **renk, alan\_hesapla() ve ciz()** gibi genel özelliklere ve davranışlara sahiptir.

**Dikdortgen** sınıfı, **Sekil** sınıfından kalıtım alır ve bu şekilde **Sekil** sınıfının tüm özelliklerini ve davranışlarını miras alır. **Dikdortgen** sınıfı ayrıca **genislik ve yukseklik** gibi özelliklere sahiptir. **alan\_hesapla() ve ciz()** metotlarını kendi özelleştirilmiş şekilde tanımlar.

**Kare** sınıfı ise **Dikdortgen** sınıfından kalıtım alır ve aynı zamanda **Dikdortgen** sınıfının alt sınıfıdır. Bu durumda, **Kare** sınıfı, **Dikdortgen** sınıfının özelliklerini ve davranışlarını miras alırken, **alan\_hesapla() ve ciz()** metotlarını kendi özelleştirilmiş şekilde tanımlar.

Sonuç olarak, yukarıdaki kod örneğinde **Sekil** sınıfından bir nesne (sekil), **Dikdortgen** sınıfından bir nesne (dikdortgen) ve **Kare** sınıfından bir nesne (kare) oluşturulmuştur. Her bir nesne, kendi sınıflarına ait yöntemleri çağırabilir ve özelliklerini kullanabilir.

**c.   Hierarchical Inheritance**

**Hierarchical inheritance (hiyerarşik kalıtım),** birden fazla alt sınıfın aynı üst sınıftan kalıtım aldığı bir kalıtım türüdür. Bu durumda, bir üst sınıf (temel sınıf), birden fazla alt sınıf tarafından miras alınır ve her alt sınıf, üst sınıfın özelliklerini ve davranışlarını miras alırken kendi özelleştirilmiş özelliklere ve davranışlara da sahip olabilir.

Hierarchical inheritance'ı daha iyi anlamak için basit bir örnek verelim. Örneğin, bir "Hayvan" sınıfı düşünelim. Bu sınıf, genel hayvanların özelliklerini ve davranışlarını içerebilir. Ardından, "Kopek" ve "Kedi" gibi alt sınıflar oluşturalım, bu alt sınıflar "Hayvan" sınıfından kalıtım alsın.

"Hayvan" sınıfı, genel hayvanların özelliklerini ve davranışlarını içerir. Örneğin, bir hayvanın ismi, beslenme şekli, hareket etme yeteneği gibi özellikler bu sınıfta yer alabilir. Bu sınıf, temel sınıf olarak adlandırılır.

"Kopek" ve "Kedi" sınıfları ise "Hayvan" sınıfından kalıtım alır ve bu şekilde "Hayvan" sınıfının özelliklerini ve davranışlarını miras alır. Ancak her bir alt sınıf kendi özelleştirilmiş özelliklere ve davranışlara da sahip olabilir. Örneğin, "Kopek" sınıfında "havla()" metodu, "Kedi" sınıfında ise "miyavla()" metodu olabilir.

Bu örnekte, "Hayvan" sınıfı temel sınıftır ve "Kopek" ve "Kedi" sınıfları, "Hayvan" sınıfından türetilmiş alt sınıflardır. Her iki alt sınıf da "Hayvan" sınıfının özelliklerini ve davranışlarını miras alırken, kendi özelleştirilmiş özelliklere ve davranışlara da sahip olabilirler.

Hierarchical inheritance, kod tekrarını önler ve sınıf hiyerarşisini oluşturarak farklı alt sınıflar arasında ilişkiler kurmayı sağlar. Bu şekilde, daha düzenli ve sürdürülebilir bir kod oluşturulmasına yardımcı olur.

**class Hayvan:**

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

**self.isim = isim**

**def hareket\_et(self):**

**print("Hareket ediliyor.")**

**def beslen(self):**

**print("Besleniliyor.")**

**class Kopek(Hayvan):**

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

**super().\_\_init\_\_(isim)**

**def havla(self):**

**print("Hav hav!")**

**class Kedi(Hayvan):**

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

**super().\_\_init\_\_(isim)**

**def miyavla(self):**

**print("Miyav!")**

**# Hayvan sınıfından miras alan Kopek ve Kedi sınıfları**

**# Özelliklerini ve davranışlarını gösterelim.**

**hayvan = Hayvan("Genel Hayvan")**

**hayvan.hareket\_et()  # Hareket ediliyor.**

**hayvan.beslen()      # Besleniliyor.**

**kopek = Kopek("Karabaş")**

**kopek.hareket\_et()   # Hareket ediliyor.**

**kopek.beslen()       # Besleniliyor.**

**kopek.havla()        # Hav hav!**

**kedi = Kedi("Pamuk")**

**kedi.hareket\_et()    # Hareket ediliyor.**

**kedi.beslen()        # Besleniliyor.**

**kedi.miyavla()       # Miyav!**

Yukarıdaki örnekte, **Hayvan** adında bir üst sınıf (temel sınıf) tanımlanmıştır. Bu sınıf, **isim, hareket\_et() ve beslen()** gibi genel özelliklere ve davranışlara sahiptir.

**Kopek** sınıfı, **Hayvan** sınıfından kalıtım alır ve bu şekilde **Hayvan** sınıfının tüm özelliklerini ve davranışlarını miras alır. Kopek sınıfı ayrıca **havla()** gibi özelliklere de sahiptir.

**Kedi** sınıfı ise **Hayvan** sınıfından kalıtım alır ve aynı zamanda **Hayvan** sınıfının alt sınıfıdır. Bu durumda, **Kedi** sınıfı, **Hayvan** sınıfının özelliklerini ve davranışlarını miras alırken, kendine özgü **miyavla()** gibi ek özelliklere de sahiptir.

Sonuç olarak, yukarıdaki kod örneğinde **Hayvan** sınıfından bir nesne (hayvan), **Kopek** sınıfından bir nesne (kopek) ve **Kedi** sınıfından bir nesne (kedi) oluşturulmuştur. Her bir nesne, kendi sınıflarına ait yöntemleri çağırabilir ve özelliklerini kullanabilir.

**d.   Multiple Inheritance**

**Multiple inheritance (çoklu kalıtım),** bir sınıfın birden fazla üst sınıftan kalıtım alabileceği bir kalıtım türüdür. Bu durumda, bir alt sınıf, birden fazla üst sınıfın özelliklerini ve davranışlarını miras alır.

Multiple inheritance'ı daha iyi anlamak için basit bir örnek verelim. Örneğin, "ProgramlamaDili" ve "MuzikEnstrumani" gibi iki farklı sınıf düşünelim. "ProgramlamaDili" sınıfı, bir programlama dilinin özelliklerini ve davranışlarını temsil edebilir (örneğin, adı, yaratıcısı, popülerliği). "MuzikEnstrumani" sınıfı ise bir müzik enstrümanının özelliklerini ve davranışlarını temsil edebilir (örneğin, adı, türü, çalma şekli).

Daha sonra, "Python" adında bir sınıf oluşturalım ve hem "ProgramlamaDili" sınıfından hem de "MuzikEnstrumani" sınıfından kalıtım alsın.

"Python" sınıfı, "ProgramlamaDili" sınıfından programlama dilinin özelliklerini ve davranışlarını, "MuzikEnstrumani" sınıfından ise müzik enstrümanının özelliklerini ve davranışlarını miras alır.

Bu örnekte, "Python" sınıfı, "ProgramlamaDili" ve "MuzikEnstrumani" sınıflarından kalıtım alır ve hem programlama dilinin özelliklerini hem de müzik enstrümanının özelliklerini içerir.

Multiple inheritance, farklı sınıflar arasında ilişkiler kurmayı ve özellikleri birleştirmeyi sağlar. Bununla birlikte, dikkatli kullanılmalıdır, çünkü çakışmalar ve karmaşıklıklara yol açabilir.

**class ProgramlamaDili:**

**def \_\_init\_\_(self, adi, yaratici, populerlik):**

**self.adi = adi**

**self.yaratici = yaratici**

**self.populerlik = populerlik**

**def bilgileri\_goster(self):**

**print("Programlama Dili Bilgileri:")**

**print(f"Adı: {self.adi}")**

**print(f"Yaratıcı: {self.yaratici}")**

**print(f"Popülerlik: {self.populerlik}")**

**class MuzikEnstrumani:**

**def \_\_init\_\_(self, adi, turu, calma\_sekli):**

**self.adi = adi**

**self.turu = turu**

**self.calma\_sekli = calma\_sekli**

**def bilgileri\_goster(self):**

**print("Müzik Enstrümanı Bilgileri:")**

**print(f"Adı: {self.adi}")**

**print(f"Türü: {self.turu}")**

**print(f"Çalma Şekli: {self.calma\_sekli}")**

**class Python(ProgramlamaDili, MuzikEnstrumani):**

**def \_\_init\_\_(self, adi, yaratici, populerlik, turu, calma\_sekli, versiyon):**

**ProgramlamaDili.\_\_init\_\_(self, adi, yaratici, populerlik)**

**MuzikEnstrumani.\_\_init\_\_(self, adi, turu, calma\_sekli)**

**self.versiyon = versiyon**

**def bilgileri\_goster(self):**

**print("Python Bilgileri:")**

**print(f"Adı: {self.adi}")**

**print(f"Yaratıcı: {self.yaratici}")**

**print(f"Popülerlik: {self.populerlik}")**

**print(f"Türü: {self.turu}")**

**print(f"Çalma Şekli: {self.calma\_sekli}")**

**print(f"Versiyon: {self.versiyon}")**

**# Python sınıfı, hem ProgramlamaDili sınıfından hem de MuzikEnstrumani sınıfından**

**# özellikleri ve davranışları miras alır.**

**python = Python("Python", "Guido van Rossum", "Yüksek", "Genel Amaçlı", "Yazılı", "3.9.0")**

**python.bilgileri\_goster()**

**# Çıktı:**

**# Python Bilgileri:**

**# Adı: Python**

**# Yaratıcı: Guido van Rossum**

**# Popülerlik: Yüksek**

**# Türü: Genel Amaçlı**

**# Çalma Şekli: Yazılı**

**# Versiyon: 3.9.0**

Yukarıdaki örnekte, **ProgramlamaDili** ve **MuzikEnstrumani** adında iki üst sınıf (temel sınıf) tanımlanmıştır. Her biri kendi özelliklerini ve **bilgileri\_goster()** metodunu içerir.

**Python** sınıfı, **ProgramlamaDili** ve **MuzikEnstrumani** sınıflarından kalıtım alırken, kendi özelliklerini de içerir. Bu durumda, **Python** sınıfı, **ProgramlamaDili** ve **MuzikEnstrumani** sınıflarının özelliklerini ve **bilgileri\_goster()** metodunu miras alırken, kendi özelleştirilmiş versiyon özelliğine sahiptir.

Yukarıdaki kod örneğinde, Python sınıfından bir nesne (python) oluşturulmuştur. Bu nesne, hem **ProgramlamaDili** sınıfının hem de **MuzikEnstrumani** sınıfının özelliklerini ve **bilgileri\_goster()** metodunu kullanabilir. Aynı zamanda Python sınıfının kendine özgü versiyon özelliğine de erişebilir.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**2. Python Polymorphism (Çok Biçimlilik) Nedir?**

Hepiniz rotada gezinmek için GPS kullanmış olmalısınız. Aynı varış noktası için trafiğe bağlı olarak bu kadar farklı rotayla karşılaşmanız şaşırtıcı değil mi, programlama açısından buna 'polimorfizm' deniyor. Bir görevin birkaç farklı şekilde gerçekleştirilebildiği böyle bir OOP metodolojisidir. Basit bir ifadeyle, bir nesnenin birden fazla biçim almasına izin veren bir özelliğidir. Python'da polimorfizm (çok biçimlilik), bir nesnenin farklı şekillerde davranabilme yeteneğidir. Bu, farklı sınıflar tarafından aynı yöntemlerin kullanılmasını ve nesnelerin aynı arayüzü kullanarak farklı davranışlar sergilemesini sağlar. Polimorfizm, nesne yönelimli programlamanın önemli bir prensibidir ve kodun daha esnek, yeniden kullanılabilir ve genişletilebilir olmasını sağlar.

Python'da polimorfizm aşağıdaki alt başlıklar altında incelenebilir:

[Python Nesne Tabanlı Programlama: Polymorphism (14)](https://www.youtube.com/watch?v=l-FFlHZljvI)

**a.  Çok biçimli fonksiyonlar:**

Python'da fonksiyonlar polimorfik olabilir. Aynı isimle farklı parametrelerle tanımlanmış birden çok fonksiyon oluşturabilir ve bu fonksiyonlar arasında geçiş yapabilirsiniz. Hangi fonksiyonun çağrılacağı, fonksiyonun verilen parametrelere göre otomatik olarak belirlenir.

Örnek:

**def toplama(x, y):**

**return x + y**

**def cikarma(x, y):**

**return x - y**

**def carpma(x, y):**

**return x \* y**

**print(toplama(2, 3))  # 5**

**print(cikarma(5, 2))  # 3**

**print(carpma(4, 3))   # 12**

**b.  Çok biçimli sınıflar:**

Python'da sınıflar da polimorfik olabilir. Aynı adı taşıyan bir metodu farklı şekillerde uygulayabilen farklı sınıflar tanımlayabilirsiniz. Bu, aynı yöntem çağrısının, sınıfın türüne göre farklı davranışlar sergilemesini sağlar.

Örnek:

**class Sekil:**

**def alan(self):**

**pass**

**class Dikdortgen(Sekil):**

**def \_\_init\_\_(self, uzunluk, genislik):**

**self.uzunluk = uzunluk**

**self.genislik = genislik**

**def alan(self):**

**return self.uzunluk \* self.genislik**

**class Kare(Sekil):**

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

**self.kenar = kenar**

**def alan(self):**

**return self.kenar \*\* 2**

**sekil1 = Dikdortgen(4, 5)**

**sekil2 = Kare(4)**

**print(sekil1.alan())  # 20**

**print(sekil2.alan())  # 16**

**c.   Polimorfik parametreler:**

Bir fonksiyon veya metoda, farklı türlerden nesneleri parametre olarak geçirebilirsiniz. Fonksiyon, geçilen nesnenin türüne göre farklı işlemler gerçekleştirebilir. Bu, farklı sınıfların aynı işlemi gerçekleştiren farklı yöntemleri olduğunda çok kullanışlıdır.

Örnek:

**class Kedi:**

**def beslen(self):**

**print("Kedi, mamayı yiyor.")**

**class Kopek:**

**def beslen(self):**

**print("Köpek, kemiği yiyor.")**

**def hayvan\_besle(hayvan):**

**hayvan.beslen()**

**kedi = Kedi()**

**kopek = Kopek()**

**hayvan\_besle(kedi)   # Kedi, mamayı yiyor.**

**hayvan\_besle(kopek)  # Köpek, kemiği yiyor.**

Bu örneklerde görüldüğü gibi, polimorfizm sayesinde aynı yöntem çağrısı farklı nesnelerde farklı sonuçlar üretebilir. Bu, kodun daha esnek ve genişletilebilir olmasını sağlar ve farklı sınıfların aynı arayüzü kullanarak birbiriyle uyumlu çalışmasını sağlar.

**d.  Compile-Time Polymorphism (Derleme Zamanı Polimorfizmi)**

Programın derleme zamanında çözümlenen bir polimorfizm türüdür.

Örnek:

**class Employee1():**

**def name(self):**

**print("Harshit is his name")**

**def salary(self):**

**print("3000 is his salary")**

**def age(self):**

**print("22 is his age")**

**class Employee2():**

**def name(self):**

**print("Rahul is his name")**

**def salary(self):**

**print("4000 is his salary")**

**def age(self):**

**print("23 is his age")**

**def func(obj)://Method Overloading**

**obj.name()**

**obj.salary()**

**obj.age()**

**obj\_emp1 = Employee1()**

**obj\_emp2 = Employee2()**

**func(obj\_emp1)**

**func(obj\_emp2)**

Çıktı:

**Harshit is his name**

**3000 is his salary**

**22 is his age**

**Rahul is his name**

**4000 is his salary**

**23 is his age**

Bu örnekte, "method overloading" (aşırı yükleme yöntemi) kullanılarak compile-time polymorphism uygulanmaktadır.

Örnekte, **Employee1** ve **Employee2** adında iki sınıf tanımlanmıştır. Her iki sınıf da aynı isimli **name(), salary(), ve age()** adlı üç yönteme sahiptir. Ancak her bir sınıf, bu yöntemleri farklı şekillerde uygular.

Daha sonra **func(obj)** adında bir fonksiyon tanımlanmıştır. Bu fonksiyon, parametre olarak herhangi bir nesne alabilir. Fonksiyon içerisinde parametre olarak geçilen nesnenin **name(), salary(), ve age()** yöntemleri çağrılır.

Son olarak, **Employee1** ve **Employee2** sınıflarından örnekler oluşturulur **(obj\_emp1 ve obj\_emp2)**. Bu örnekler, **func()** fonksiyonuna parametre olarak geçilir.

Derleme zamanında, **func()** fonksiyonunun parametresi olan nesne türüne göre, ilgili sınıfın yöntemleri çağrılır. Bu sayede, **func(obj\_emp1)** çağrısında **Employee1** sınıfının yöntemleri çalışırken, **func(obj\_emp2)** çağrısında **Employee2** sınıfının yöntemleri çalışır.

Bu örnekte compile-time polymorphism kullanılarak method overloading yapılmıştır. Yani aynı isme sahip farklı yöntemler tanımlanmış ve hangi yöntemin çağrılacağı, derleme zamanında belirlenmiştir. Bu, kodun daha verimli ve optimize edilebilir olmasını sağlar.

**e.  Run-Time Polymorphism (Çalışma Zamanı Polimorfizmi)**

Programın çalışma zamanında çözümlenen bir polimorfizm türüdür.

Örnek:

**class Employee():**

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

**self.name = name**

**self.age = age**

**self.salary = salary**

**self.id = id**

**def earn(self):**

**pass**

**class ChildEmployee1(Employee):**

**def earn(self): //Run-time polymorphism**

**print("no money")**

**class ChildEmployee2(Employee):**

**def earn(self):**

**print("has money")**

**c = ChildEmployee1**

**c.earn(Employee)**

**d = ChildEmployee2**

**d.earn(Employee)**

Çıktı:

**no money, has money**

Bu örnekte, "method overriding" (yöntem geçersiz kılma) kullanılarak run-time polymorphism uygulanmaktadır.

Örnekte, **Employee** adında bir ana sınıf (base class) tanımlanmıştır. Bu sınıfın bir yapıcı metodu **(\_\_init\_\_()) ve earn()** adlı bir yöntemi bulunmaktadır. **earn()** yöntemi, alt sınıflar tarafından geçersiz kılınmak üzere bir yer tutucu yöntem olarak tanımlanmıştır.

Daha sonra **ChildEmployee1** ve **ChildEmployee2** adında iki alt sınıf (derived class) tanımlanmıştır. Her iki sınıf da **earn()** yöntemini geçersiz kılar ve farklı davranışlar sergiler. **ChildEmployee1** sınıfında **earn()** yöntemi "no money" mesajını, **ChildEmployee2** sınıfında ise "has money" mesajını yazdırır.

Son olarak, **ChildEmployee1** ve **ChildEmployee2** sınıflarından örnekler oluşturulur **(c ve d)**. Bu örnekler, **earn()** yöntemini çağırır.

Çalışma zamanında, **c.earn()** çağrısında **ChildEmployee1** sınıfının **earn()** yöntemi çalışırken, **d.earn()** çağrısında **ChildEmployee2** sınıfının **earn()** yöntemi çalışır. Bu, run-time polymorphism sayesinde nesnelerin aynı arayüzü kullanarak farklı davranışlar sergilemesini sağlar.

Bu örnekte run-time polymorphism kullanılarak method overriding yapılmıştır. Yani alt sınıflar, üst sınıfın aynı adlı yöntemlerini geçersiz kılar ve kendi özelleştirilmiş yöntemlerini uygular. Hangi yöntemin çağrılacağı, çalışma zamanında belirlenir. Bu, kodun esnek ve dinamik olmasını sağlar ve çalışma zamanında doğru yöntemin çağrılmasını sağlar.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**3. Python Abstraction (Soyutlama) Nedir?**

İnternet bankacılığı veya başka bir işlem kullanarak sinemadan bir sinema bileti rezervasyonu yaptığınızı varsayalım. PIN'in nasıl üretildiğine veya doğrulamanın nasıl yapıldığına dair prosedürü bilmiyorsunuz. Buna programlama açısından 'soyutlama' denir, temel olarak yalnızca belirli bir sürecin uygulama ayrıntılarını göstermeniz ve ayrıntıları kullanıcıdan gizlemeniz anlamına gelir.

Soyutlama, verilerin ve işlevlerin birleştirilerek daha büyük bir yapı oluşturmasını sağlar. Aşağıda soyutlamanın alt başlıklarını ve açıklamalı örneklerini bulabilirsiniz:

[#Python Dersleri: 62 - #OOP - ABSTRACTION (SOYUTLAMA) (2023)](https://www.youtube.com/watch?v=_dBDC7kGLZU)

**a.  Soyut Veri Tipleri:**

Soyut veri tipleri, verilerin iç yapısını gizleyerek sadece verilerle yapılan işlemlerin kullanıcıya sunulmasını sağlar. Örnek olarak, liste, yığın (stack), kuyruk (queue) gibi veri yapıları soyut veri tiplerine örnek olarak verilebilir. Kullanıcı, bu veri yapılarına eleman ekleyebilir, çıkarabilir veya üzerinde işlemler yapabilir, ancak veri yapısının nasıl uygulandığına dair ayrıntıları bilmek zorunda değildir.

Örnek:

**class Stack:**

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

**self.items = []**

**def push(self, item):**

**self.items.append(item)**

**def pop(self):**

**return self.items.pop()**

**stack = Stack()**

**stack.push(10)**

**stack.push(20)**

**print(stack.pop())  # 20**

Bu örnekte, Stack adlı bir sınıf oluşturulmuştur. Kullanıcı, push() ve pop() yöntemlerini kullanarak eleman ekleyebilir ve çıkarabilir. Ancak, veri yapısının arkasında listelerin kullanıldığı bilgisine gerek duyulmaz.

**b.  Soyut Sınıflar:**

Soyut sınıflar, diğer sınıfların temel alınması için tasarlanmış soyut bir yapıdır. Soyut sınıflar, kendileri bir örneği oluşturulamayan ve alt sınıflar tarafından genişletilmek üzere tasarlanan sınıflardır. Soyut sınıflarda, ortak yöntemlerin tanımlanması ve alt sınıflar tarafından uygulanması sağlanır.

Örnek:

**from abc import ABC, abstractmethod**

**class Shape(ABC):**

**@abstractmethod**

**def area(self):**

**pass**

**class Rectangle(Shape):**

**def \_\_init\_\_(self, width, height):**

**self.width = width**

**self.height = height**

**def area(self):**

**return self.width \* self.height**

**rectangle = Rectangle(4, 5)**

**print(rectangle.area())  # 20**

Bu örnekte, Shape adında bir soyut sınıf tanımlanmıştır. Bu sınıf, area() adlı soyut bir yönteme sahiptir. Rectangle adlı bir alt sınıf, Shape sınıfını genişleterek area() yöntemini uygular. Alt sınıflar, soyut sınıftan türetilerek ve soyut yöntemleri uygulayarak somut bir işlevsellik sağlar.

Soyutlama, karmaşık sistemleri anlaşılır ve yönetilebilir hale getirir. Kullanıcılar, soyutlamanın sunduğu arayüzlerle etkileşimde bulunabilirken, ayrıntıları bilmeden işlevleri kullanabilirler. Bu da kodun yeniden kullanılabilirliğini artırır ve daha iyi bir kod yapısı oluşturulmasına yardımcı olur.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**4. Python Kapsülleme (Encapsulation) Nedir?**

[PYTHON-Encapsulation (Kapsülleme) Nedir?](https://www.youtube.com/watch?v=QHcl22_zbw0)

Kapsülleme, methotların ve variable’ların erişimlerini kısıtlamak anlamına geliyor. Biz metotlara veya variable’lara direk erişme ve değiştirme kısıtı özelliğini ekliyoruz. Peki bunu niye yapıyoruz? Çünkü yazdığımız kodların değiştirilmemesi  yada değiştirdiğimiz değerlerin kontrollü olarak değiştirilmesi gerektiği için. İşte tam bu durumlarda kapsülleme sihirli bir değnek gibi imdadımıza yetişiyor. Şimdi örneklerle gösterelim.

**class RegisterCourse:**

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

**self.name = "Ahmet"**

**self.surname = "Kaleli"**

**self.exam1 = 74**

**self.exam2 = 80**

**register = RegisterCourse()**

**print("İsim : ",register.name)**

**print("Soyisim : ",register.surname)**

**print("Exam 1 : ",register.exam1)**

**print("Exam 2 :",register.exam2)**

RegisterCourse isminde sınıf oluşturduk.  Şimdi sınıfımızın değişkenlerini görüntüleyelim

**İsim : Ahmet**

**Soyisim : Kaleli**

**Exam 1 : 74**

**Exam 2 : 80**

Yalnız burada bir problem var. Bu öğrencinin bilgilerine dışarıdan ulaşabiliyoruz hatta değiştirebiliyoruz.

**register.exam1 = 50**

**register.exam2 = 60**

Hemen bilgilere tekrar bakalım;

**İsim :  Ahmet**

**Soyisim :  Kaleli**

**Exam 1 :  50**

**Exam 2 : 60**

Gördüğünüz gibi öğrencinin kişisel bilgilerine ulaşabiliyor ve değiştirebiliyoruz. Peki bu kişinin bilgilerine dışarıdan ulaşılmasını ve değiştirilmesini nasıl kısıtlayabiliriz.

**Python Encaplusation (Kapsülleme) Kullanımı**

Kişinin bilgilerini global olarak oluşturduğumuz için herkes tarafından erişilebilir durumdadır. Bunun için biz verilerimizi ve metotlarımızı private(özel) olarak tanımlamamız gerekiyor. Nasıl yapıldığını hemen görelim.

Basit bir işlem, sadece oluşturduğumuz verilerin önüne “\_\_” ekliyoruz,  hemen deneyelim;

**class RegisterCourse:**

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

**self.name = "Ahmet"**

**self.surname = "Kaleli"**

**self.\_\_exam1 = 74**

**self.\_\_exam2 = 80**

**register = RegisterCourse()**

**print("İsim : ",register.name)**

**print("Soyisim : ",register.surname)**

**print("Exam 1 : ",register.exam1)**

**print("Exam 2 :",register.exam2)**

Şu şekilde bir hata aldık değil mi?

**AttributeError: 'RegisterCourse' object has no attribute 'exam1'**

Biz sınav bilgilerini private(özel) yaptığımız için sınav bilgilerine ulaşamadığından hata verdi. Artık bu değerlere yalnız bu sınıf içinde erişebiliriz. Peki diğer sınıflardan nasıl erişeceğiz? Yazıyı okumaya devam edelim. Peki aynı işlevleri methodlar içinde yapabiliyor muyuz? Hemen bir örnekle gösterelim.

**class RegisterCourse:**

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

**self.name = "Ahmet"**

**self.surname = "Kaleli"**

**self.\_\_exam1 = 74**

**self.\_\_exam2 = 80**

**self.courses = []**

**def add(self, course):**

**self.courses.append(course)**

**register = RegisterCourse()**

**register.add("Database Managment")**

**print("İsim : ",register.name)**

**print("Soyisim : ",register.surname)**

**print(register.courses)**

Gördüğünüz gibi Öğrencinin aldığı dersleri görebiliyor ve ders ekleyebiliyoruz. Şimdi bu method’da dışarıdan erişimi ve değiştirmeyi kısıtlayalım.

**class RegisterCourse:**

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

**self.name = "Ahmet"**

**self.surname = "Kaleli"**

**self.\_\_exam1 = 74**

**self.\_\_exam2 = 80**

**self.courses = []**

**def \_\_add(self,course):**

**self.courses.append(course)**

**register = RegisterCourse()**

**register.add("Database Managment")**

**AttributeError: 'RegisterCourse' object has no attribute 'add'**

Yine benzer bir hata aldık. add() diye bir metodumuz olduğu halde dışarıdan erişim olmadığı için (bunu methodun başına \_\_ ekleyerek gerçekleştirdik ) erişemiyoruz.

Peki aklımıza şu gelebilir. Tamam sen benim dışarıdan exam1 ve exam2 değişkenlerine erişimini engelledin. Ben artık exam1 ve exam2 değişkenini hem okuyamıyorum hem de değiştiremiyorum. Ama ben en azından değeri okumak istiyorum, değiştirmezsem de olur

**Kapsüllenmiş Verilere Dışarıdan Erişim**

Peki, bunun nasıl yapacağız? Hemen hemen birçok programlama dilinde sıkla karşılaştığımız getter ve setter methodlarına benzer bir yapı ile yapacağız o zaman deneyelim.

**class RegisterCourse:**

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

**self.name = "Ahmet"**

**self.surname = "Kaleli"**

**self.\_\_exam1 = 74**

**self.\_\_exam2 = 80**

**self.\_\_courses = []**

**def \_\_add(self,course):**

**self.courses.append(course)**

**def getExam1(self):**

**return self.\_\_exam1**

**register = RegisterCourse()**

**print("İsim : ",register.name)**

**print("Soyisim : ",register.surname)**

**print("Exam 1 : ",register.getExam1())**

Hemen Çıktıya bakalım.

**İsim : Ahmet**

**Soyisim : Kaleli**

**Exam 1 : 4**

Gördüğünüz gibi bu şekilde exam1 bilgisine ulaşmış olduk. Bu arada getExam1 ismini biz oluşturduk, sizde başka isim verebilirsiniz. Bu yapı genel bir yapı olduğu için bu şekilde yazmanız (değişkenin başına get koyarak) kodlarınızın okunabilirliği artırabilir.

Peki değişken private olsa da ben hem erişmek hem de değerini değiştirmek istiyorum diyelim. Az önce sadece getter metodu eklemiştik şimdi birde setter methodu ekleyelim.

**class RegisterCourse:**

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

**self.name = "Ahmet"**

**self.surname = "Kaleli"**

**self.\_\_exam1 = 74**

**self.\_\_exam2 = 80**

**self.\_\_courses = []**

**def \_\_add(self,course):**

**self.courses.append(course)**

**def getExam1(self):**

**return self.\_\_exam1**

**def setExam1(self, newVal):**

**self.\_\_exam1 = newVal**

**register = RegisterCourse()**

**print("İsim : ",register.name)**

**print("Soyisim : ",register.surname)**

**print("Exam 1 : ",register.getExam1())**

Hemen çıktıya bakalım;

**İsim : Ahmet**

**Soyisim : Kaleli**

**Exam 1 : 74**

Gördüğünüz gibi not bilgisi değişmedi. Çünkü biz setExam1() fonksiyonunu çağırmadık hemen oluşturduğumuz setExam1() fonksiyonunu çağıralım.

**register.setExam1(40)**

Çıktıya bakalım;

**İsim : Ahmet**

**Soyisim : Kaleli**

**Exam 1 : 40**

Bu şekilde fonksiyonumuzu çağırmış olduk. Artık private değerleri setter fonksiyonu ile değiştirebilirsiniz.

Peki direk erişemiyorum ama  setter methodu ile değişkeni istediğim gibi değiştirebildim. Neden getter setter ile uğraşayım. Neden private yapıp setter methodu kullanma gereği duydum?  Çünkü direk eriştiğimde herhangi bir kontrol yapmadan değişkenin değerini değiştirebilirim. Ama bu şekilde setter fonksiyonu kullandığımda içerisine istediğim validasyonları koyabilirim. Örneğin exam1 değerine direk erişimde – (negatif) değerler verebilirken setter methodunda negatif değer geldiğinde bunu direk exam1 değerine eşitlemeden kullanıcıyı uyarabilirim veya hata fırlatabilirim. Hemen bir örnekle gösterelim;

**def setExam(self,newVal):**

**if newVal<0 or newVal>100 :**

**raise ValueError("Sınav puanı 0'dan küçük ve 100'den büyük bir değer olamaz ")**

Setter fonksiyonumuzu yazdık şimdi programımızın tamamını görelim.

**class RegisterCourse:**

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

**self.name = "Ahmet"**

**self.surname = "Kaleli"**

**self.\_\_exam1 = 74**

**self.\_\_exam2 = 80**

**self.\_\_courses = []**

**def \_\_add(self,course):**

**self.courses.append(course)**

**def getExam(self):**

**return self.\_\_exam1**

**def setExam(self,newVal):**

**if newVal<0 or newVal>100 :**

**raise ValueError("Sınav puanı 0'dan küçük ve 100'den büyük bir değer olamaz ")**

**register = RegisterCourse()**

**register.setExam(-10)**

Şimdi çıktıya bakalım;

**ValueError: Sınav puanı 0'dan küçük ve 100'den büyük bir değer olamaz**

Biz burada sınav puanının 0’dan küçük ve 100’den büyük değer olma durumlarını kontrol etmiş olduk.