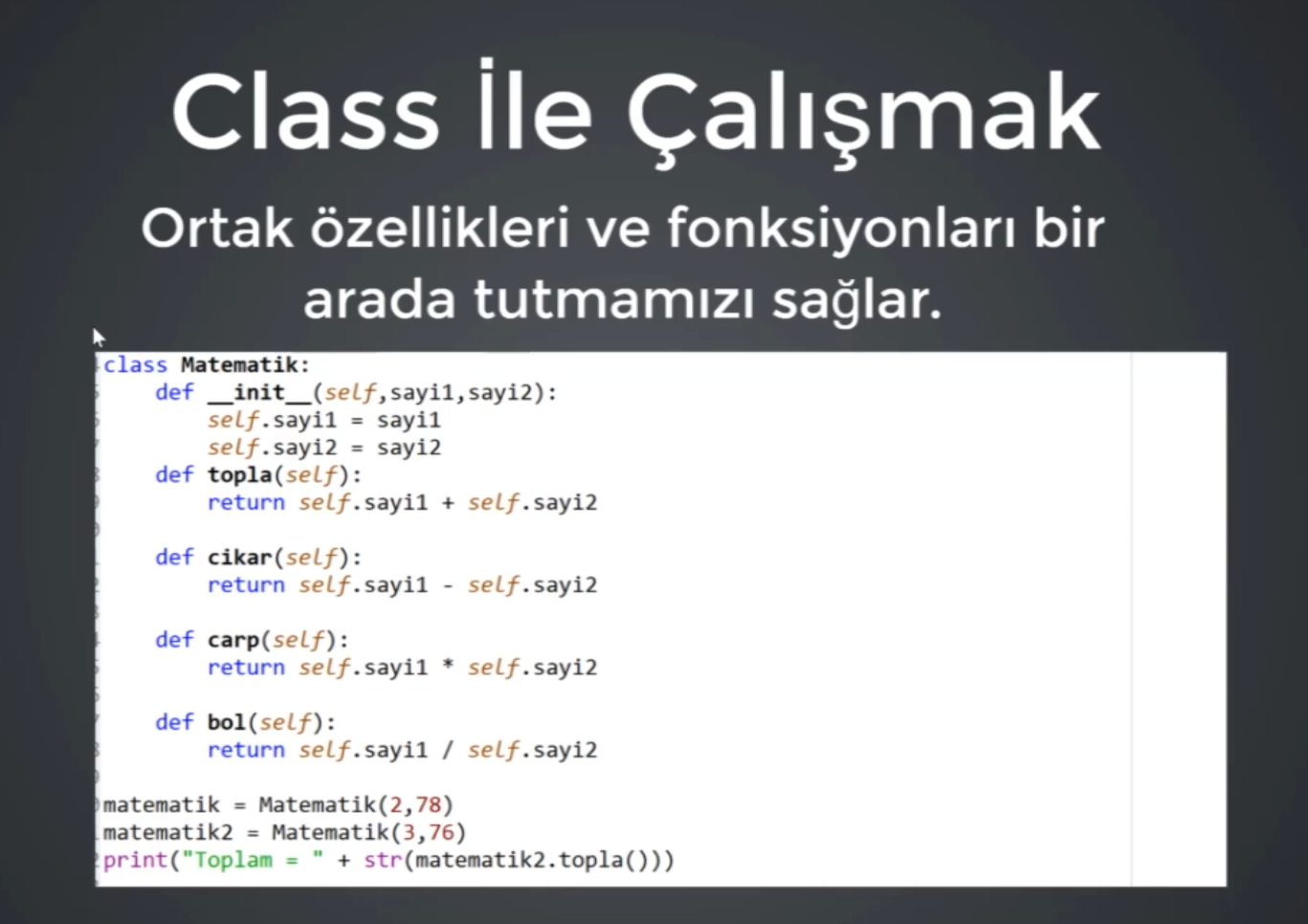
**Class (OOP) Nesne Tabanlı Programlama**

**Giriş**

**Class:** sınıflar, nesne üretmemizi sağlayan veri tipleridir.

Nesne tabanlı programlama, adından da anlaşılacağı gibi, nesneler (ve dolayısıyla sınıflar) temel alınarak gerçekleştirilen bir programlama faaliyetidir.



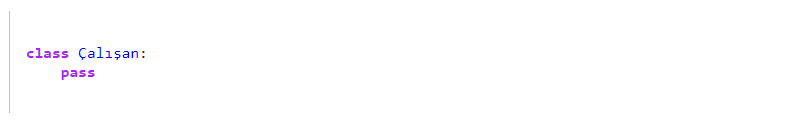
Örneğin Matematiksel işlemleri (topla, çarp, böl..) bunları bir fonksiyona (def) kaydediyoruz

Ve bu 4 adet fonksiyonu da Classların içine kaydediyoruz

Böylece sınıfların yapısı oluyor.

Örn. Matematiksel işlemlerin bulunduğu Matematik Class’ı veya Musteriler Class’ı (Musteri Ekle,Sil,Güncelle)

**Class Tanımlamak**

 **veya** 

**Class ne işe yarar? Amacı nedir?**

Belirli bir amacı yerine getirmek için yazılan kodlar zamanla genişleyebilir, üzerinde değişiklik yapmak gerekebilir.

Bu koşulları göz önünde bulundurduğumuz zaman kodlar üzerinde oynama yapmak veya ekleme, çıkarma yapmak oldukça zor ve karmaşıktır. (Örn. 1000 Satırlık kod)

Bu koşulları düzeltmek için kodlarımızı fonksiyon içerisinde kullanabiliriz fakat fonksiyonlarda şöyle bir durum söz konusudur.

Fonksiyonlara her yerden ulaşabilir fakat her yerden değiştirme işlemi yapamayız.

Class kullanmak tüm bu sorunları ortadan kaldırır ve kodumuzu daha okunaklı bir hale getirir.

**Class Tanımlamak**

Nesne tabanlı programlama yaklaşımı, özellikle birtakım ortak niteliklere ve davranış şekillerine sahip gruplar tanımlamak gerektiğinde son derece kullanışlıdır.

Mesela şöyle bir örnek düşünün: Diyelim ki çalıştığınız işyerinde, işe alınan kişilerin kayıtlarını tutan bir veritabanınız var. Bir kişi işe alındığında, o kişiye dair belli birtakım bilgileri bu veritabanına işliyorsunuz. Mesela işe alınan kişinin adı, soyadı, unvanı, maaşı ve buna benzer başka bilgiler…

Çalışmaya başlayacak kişileri temsil eden bir ‘Çalışan’ grubunu, bu grubun nitelikleri ile faaliyetlerini tutacak yapıyı ve bu grubun bütün öğelerinin taşıyacağı özellikleri nesne tabanlı programlama yaklaşımı ile kolayca kodlayabilirsiniz.

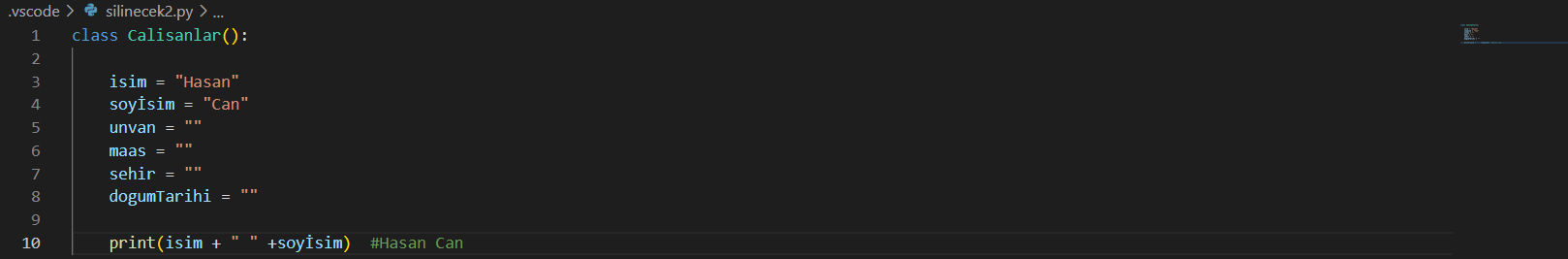
Aynı şekilde, mesela yazdığınız bir oyun programı için, bir ‘Asker’ grubunu nesne tabanlı programlama mantığı içinde tanımlayarak, bu grubun her bir üyesinin sahip olacağı nitelikleri, kabiliyetleri ve davranış şekillerini kodlayabilir; mesela askerlerin sağa sola nasıl hareket edeceklerini, hangi durumlarda puan/enerji/güç kazanacaklarını veya kaybedeceklerini, bir asker ilk kez oluşturulduğunda hangi özellikleri taşıyacağını ve aklınıza gelebilecek başka her türlü özelliği tek tek belirleyebilirsiniz.

**Sınıf Nitelikleri** (class attribute)

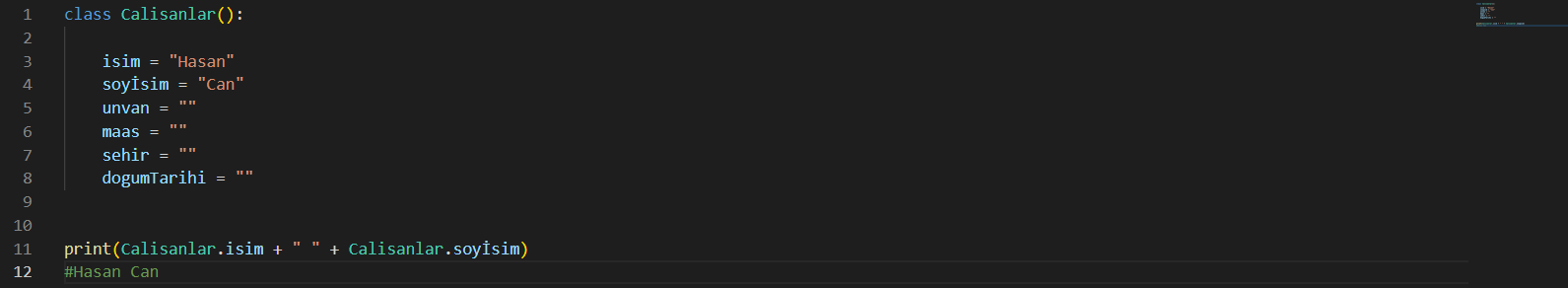
Bu sınıflara birtakım nitelikler ekleyerek bu sınıfları kullanışlı hale getirebiliriz.



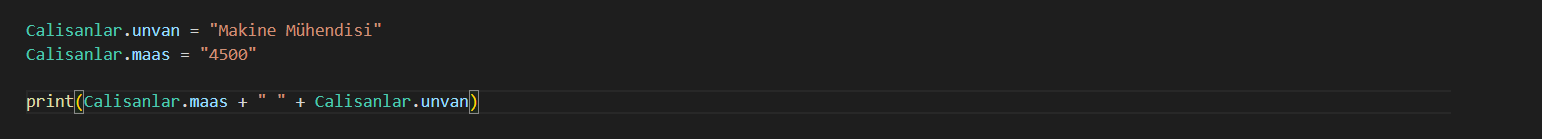
**NOT:** Class’ların çalışması için çağırılmasına (Fonks. gibi) gerek yoktur.



Fakat sınıf niteliklerinin ne zaman çalışacağını kendiniz kontrol etmek isterseniz, bu nitelikleri sınıf dışında kullanabilirsiniz:



**NOT:** Eğer isterseniz bu sınıfa yeni sınıf nitelikleri de ekleyebilirsiniz:



Ancak burada şöyle bir sorun var: Biz yukarıdaki gibi doğrudan sınıf adını kullanarak öğelere eriştiğimizde kodlarımız tek kullanımlık olmuş oluyor. Yani bu şekilde ancak tek bir Çalışan() nesnesi (‘nesne’ kavramına ilerde değineceğiz), dolayısıyla da tek bir çalışan oluşturma imkanı elde edebiliyoruz. Ama biz, mantıken, sınıf içinde belirtilen özellikleri taşıyan, Ahmet, Mehmet, Veli, Selim, Selin ve buna benzer, istediğimiz sayıda çalışan oluşturabilmeliyiz. Peki ama nasıl?

Daha önce de söylediğimiz gibi, sınıflar belli birtakım ortak özelliklere sahip gruplar tanımlamak için biçilmiş kaftandır.

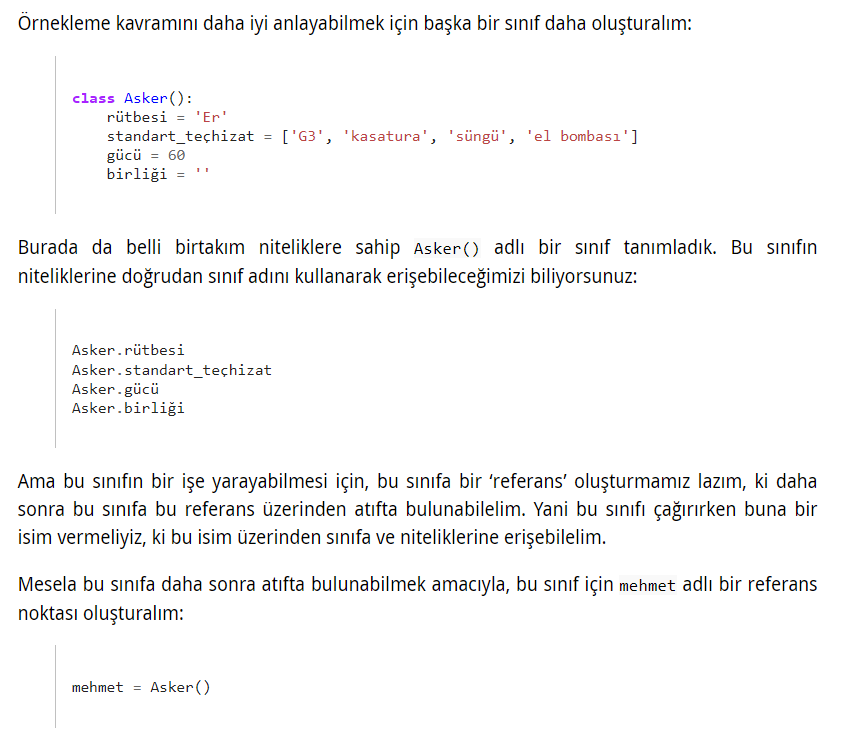
Burada da, her bir çalışan için ortak birtakım nitelikler tanımlayan Çalışan() adlı bir sınıf oluşturduk.

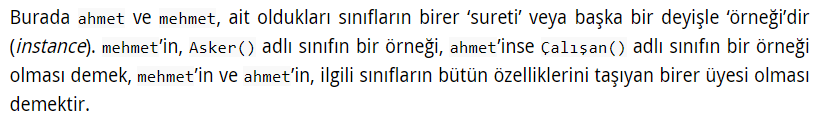
Ancak elbette bu sınıfın bir işe yarayabilmesi için, biraz önce de değindiğimiz gibi, bu sınıfı temel alarak, bu sınıfta belirtilen nitelikleri taşıyan birden fazla sınıf üyesi meydana getirebilmemiz lazım.

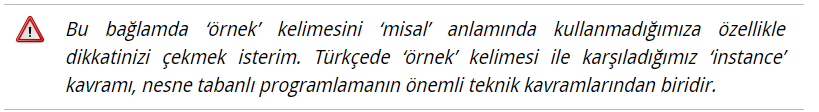


Burada sınıfımızı Onur adlı bir değişkene atadık.

İşte bu işleme teknik dilde ‘örnekleme’ veya ‘örneklendirme’ (instantiation) adı verilir. Bu işlemi fonksiyon çağırma ile kıyaslayabiliriz: Python programlama dilinde bir fonksiyonu kullanışlı hale getirme işlemine ‘çağırma’, bir sınıfı kullanışlı hale getirme işlemine ise ‘örnekleme’ adı veriyoruz.







İnstance : Örnek

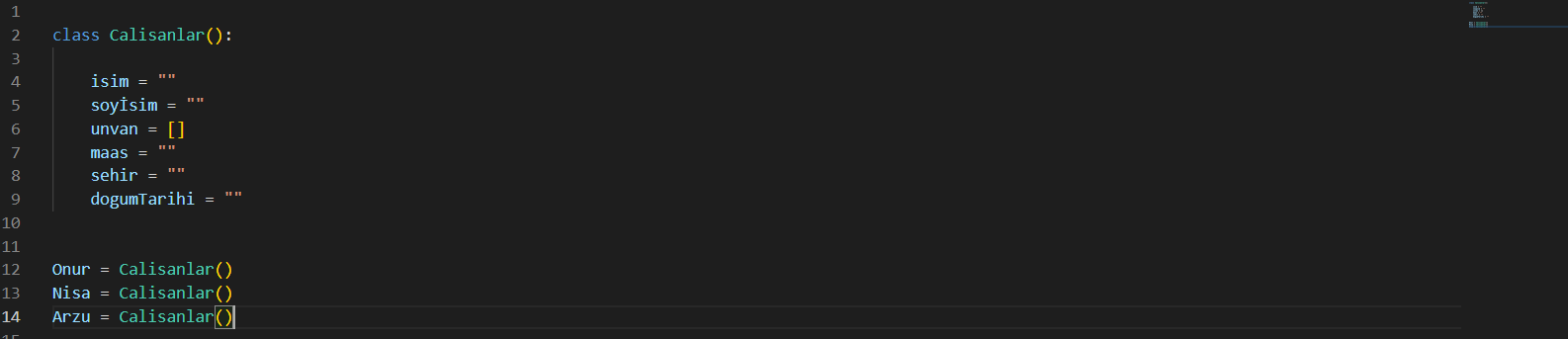
 Şimdi tüm bu bilgileri edindikten sonra Classların bir sorunundan bahsedeceğiz.

Eğer Liste olarak bir atama yapsaydık, yani mutebla (değiştirilebilir) bir sorunla karşı karşıya kalacaktık.

Bir örnekleme atadığımız nitelik, diğer tüm örneklemlere de aktarılacaktı

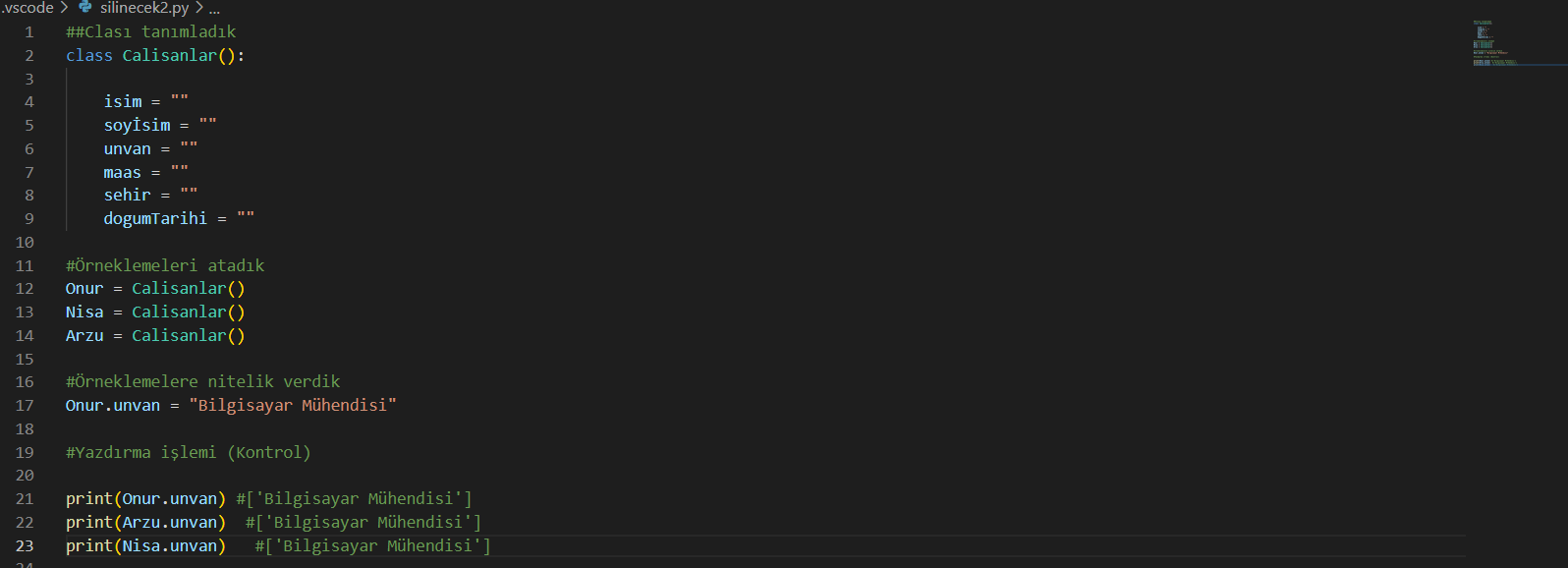
Değişkenler immutebla’dır yani değiştirilemezdir.

Bu yüzden değişken kullanınca aynı sorunla karşı karşıya kalmıyoruz!



Bu şekilde 3 adet örneklem atadığımızı düşünecek olursak ve tüm bu çalışanların farklı nitelikleri, özellikleri olduğunu varsayarsak

Buna göre atama işlemi yapalım.



Halbuki sadece Onur için bir atama yapmıştık ama bu diğer tüm örneklemlere de aktarıldı.

Gördüğünüz gibi, yalnızca “Onur” örneğine eklemek istediğimiz ‘Bilgisayar Mühendisi unvanı diğer tüm örneklere de eklenmiş. Ancak normal şartlarda arzu edilen bir şey değildir bu.

Zira bu durum aslında programımızdaki bir tasarım hatasına işaret eder. Peki ama bu durumun sebebi nedir?

Hatırlarsanız, sınıf niteliklerinden bahsederken, bu niteliklerin önemli bir özelliğinin, sınıf çağrılmadan çalışmaya başlamaları olduğunu söylemiştik.

Sınıf niteliklerinin bir başka önemli özelliği de, bu niteliklere atanan değerlerin ve eğer yapılabiliyorsa bu değerler üzerinde sonradan yapılan değişikliklerin o sınıfın bütün örneklerini etkiliyor olmasıdır.

Eğer ilgili sınıf niteliği; karakter dizisi, demet ve sayı gibi değiştirilemeyen (immutable) bir veri tipi ise bu sınıf niteliği üzerinde zaten değişiklik yapamazsınız.

Yaptığınız şey ancak ilgili sınıf niteliğini yeniden tanımlamak olacaktır.

Ancak eğer sınıf niteliği, liste, sözlük ve küme gibi değiştirilebilir (mutable) bir veri tipi ise bu nitelik üzerinde yapacağınız değişiklikler bütün sınıf örneklerine yansıyacaktır.

Yazdığınız program açısından bu özellik arzu ettiğiniz bir şey olabilir veya olmayabilir.

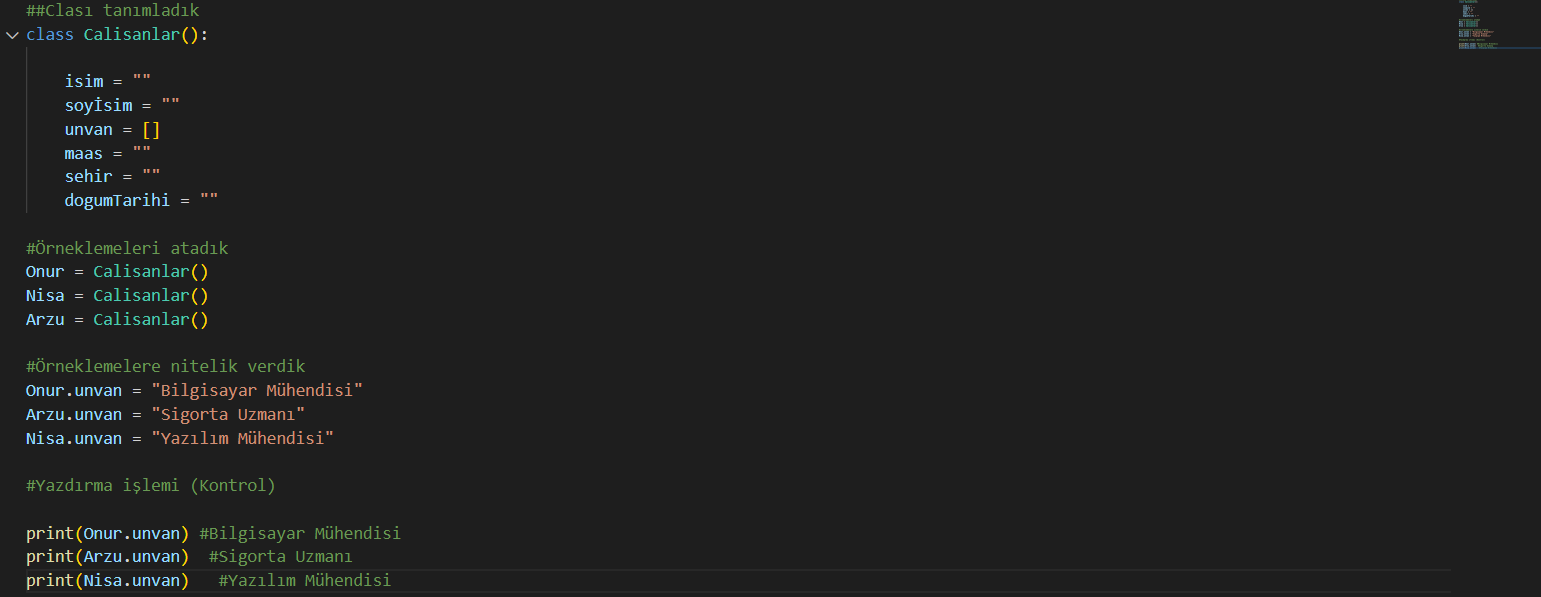
Önemli olan, sınıf niteliklerinin bu özelliğinin farkında olmanız ve kodlarınızı bu bilgi çerçevesinde yazmanızdır.

**Sonradan Eklenen Bir NOT: (Aklında bulunsun bu da)**

**Sadece bir örnekleme** (instance’e) atama yaparsak, diğer örneklemlerde bu atamadan etkileniyor ve bu örneklemin özelliklerini kendi üzerine alıyor

Fakat diğer örneklemlere de farklı atamalar yaparsak o zaman bu özellik devre dışı kalıyor ve her örneklem kendine has özellikleri devralarak devam ediyor.

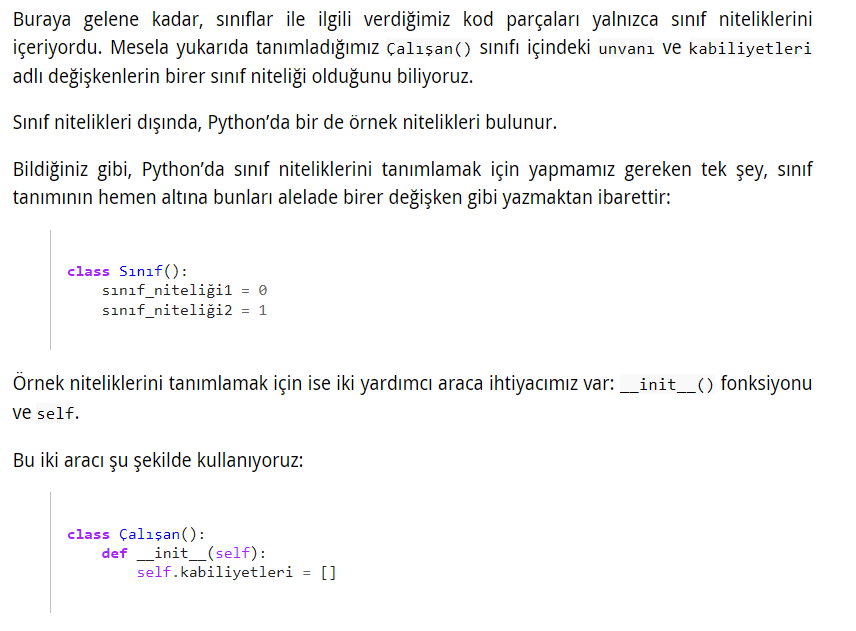
Örneğin, Nisa adlı örnekleme hiçbir atama yapmazsak ve diğer Onur ve Arzu örneklemlerine atama yaparsak, Nisa örnekleme “” Null kalıyor



* Peki o halde biz değerinin her örnekte ortak değil de her örneğe özgü olmasını istediğimiz nitelikleri nasıl tanımlayacağız?
* Elbette sınıf nitelikleri yerine örnek nitelikleri denen başka bir kavramdan yararlanarak…

**Örnek Nitelikleri** (instance attributes)

**\_\_init\_\_ Fonksiyonu ve self**



**NOT:** Hatırlarsanız, sınıf niteliklerini anlatırken bunların önemli bir özelliğinin, sınıfın çağrılmasına gerek olmadan çalışmaya başlaması olduğunu söylemiştik

Örnek nitelikleri ise farklıdır:

Mesela



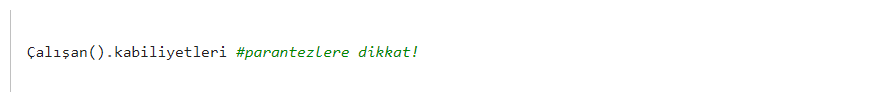
Bu kodları çalıştırdığınızda herhangi bir çıktı almazsınız.

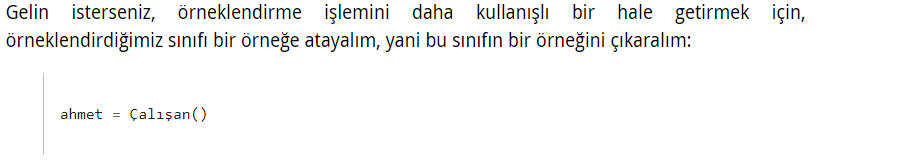
Bu kodların çıktı verebilmesi için sınıfımızı şu şekilde mutlaka örneklememiz lazım:



**NOT:** Ayrıca sınıf niteliklerinin aksine, örnek niteliklerine sınıf adları üzerinden erişemeyiz.

Bunun için şu şekilde bir kullanım yapmalıyız.





… Tüm bu anlatılanların genel bir özeti ve anlatımı

1. Def \_\_init\_\_(): Fonksiyonu, Class içerisinde bulunan gömülü bir fonksiyondur.

Siz def \_\_init\_\_ kullansanız da kullanmasanız da arka planda çalışacaktır ama bu fonksiyona bir girdi yapmamışsanız eğer haliyle çıktı da alamayacaksınızdır.

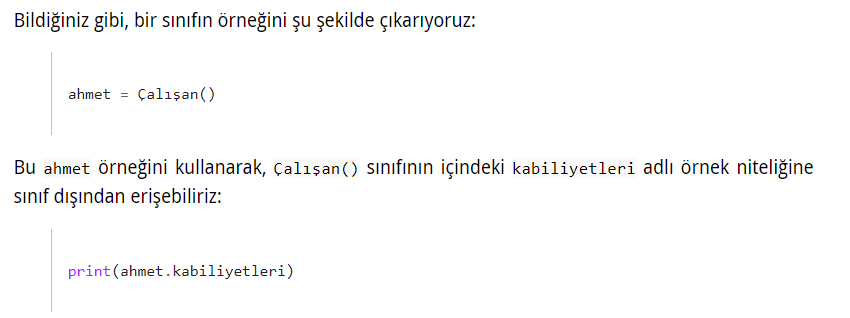
Self’i kullanabilmek için def \_\_init\_\_ fonksiyonuna ihtiyacınız vardır.

1. Self ingilizce de “kendi”, “öz” gibi anlamlar için kullanılmaktadır.

Eğer bir Örnek Niteliği (instance attiribute) kullanacaksınız, Self yazmalısınız çünkü örnek nitelikleri “unique” olmalıdır.

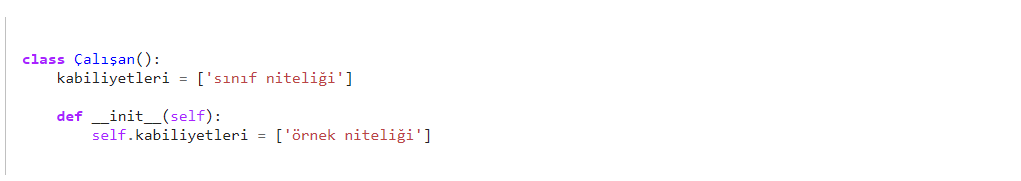
Diğer Örneklere etki etmemelidir.

Devam…

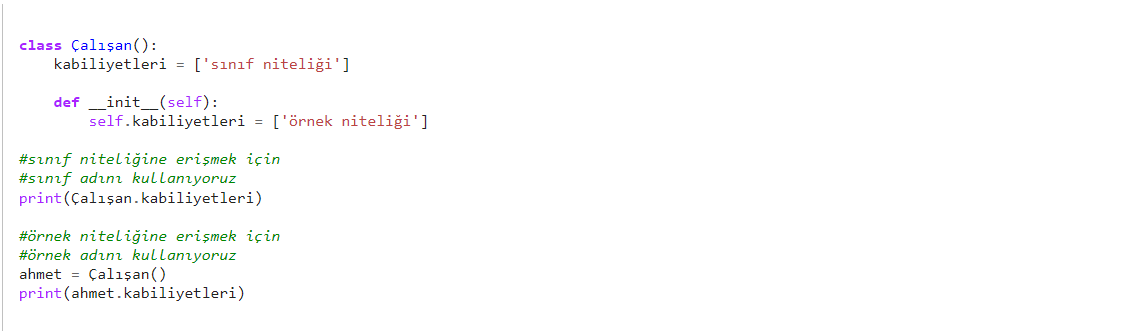


İşte self kelimesi, yukarıdaki kodda yer alan ahmet kelimesini temsil ediyor. Yani ahmet.kabiliyetleri şeklinde bir kod yazabilmemizi sağlayan şey, \_\_init\_\_() fonksiyonu içinde belirttiğimiz self kelimesidir.

**Sınıf Nitelikleri ve Örnek Niteliklerinin Aynı Classta Kullanılması**



**Çağırma İşlemleri**



Ancak elbette, aynı adı taşıyan bir sınıf niteliği ile bir örnek niteliğini aynı sınıf içinde tanımlamak daha baştan iyi bir fikir değildir

**Örnek Metotları** (instance methods)



**Örnek Metotları ve Yapılabilecek İşlemler**

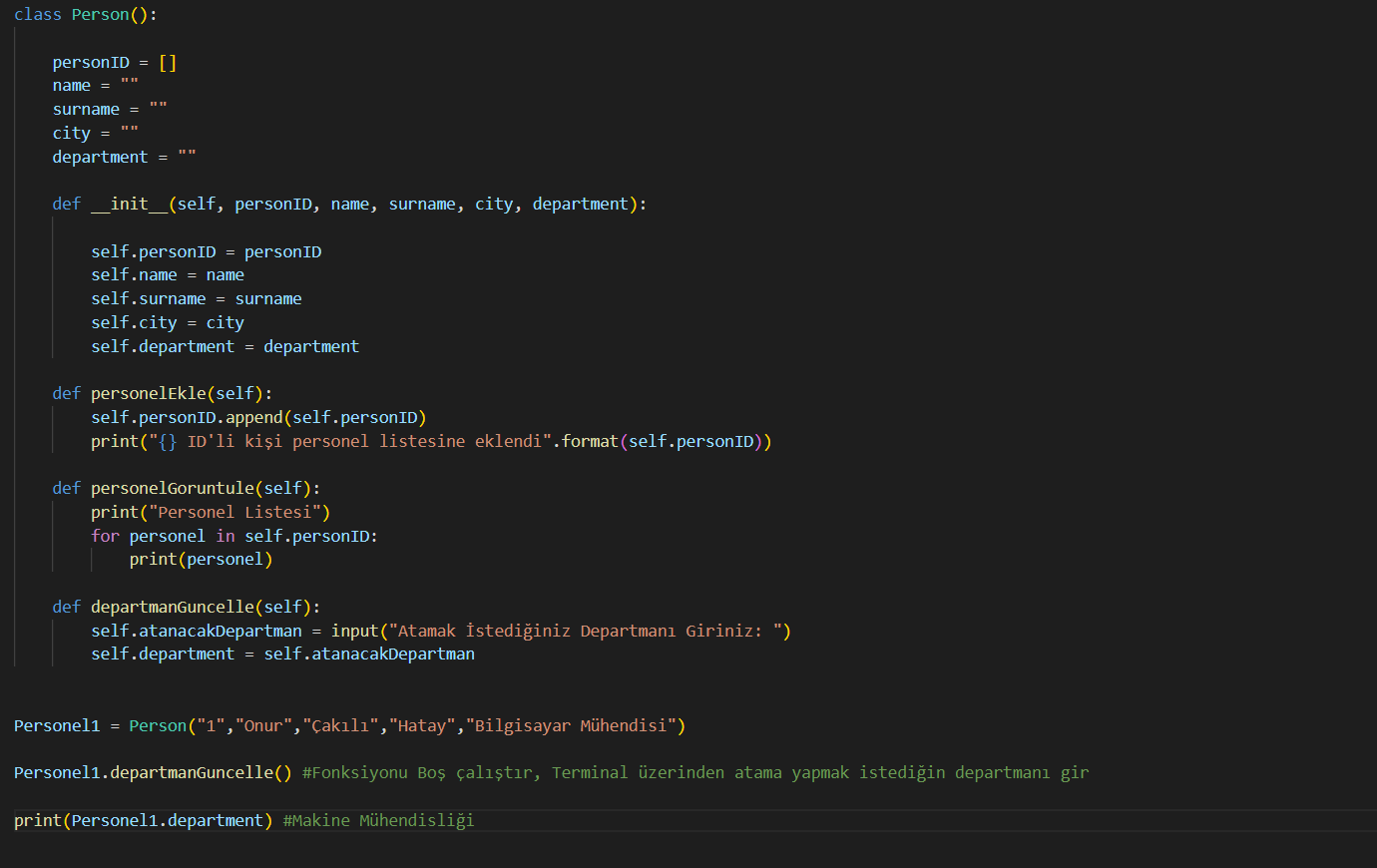
1. Tanımladığımız sınıfın niteliklerine, çıkardığımız örnekler üzerinden erişebiliriz:



1. Yine bu örnekler üzerinden, bu nitelikleri değiştirebiliriz de:



**Örnek1 Üzerinden Genel Bakış**





**Örnek2**



**NOT:** Örnek MetotlarındaKullanılan Fonksiyonlar ve Parametreler için “self” yazmayı unutma.

-Ardından ise kolaylık sağlaması için bir dipnot.

Bir parametre için self kullandıysan eğer (self.parametre), o parametreyi kullandığın her yerde onu artık self.parametre olarak kullanırsın

(Self.name = name) kullanımından sonra artık name, diye bir şey ortada kalmamıştır.

Onun adı artık self.name olmuştur.

.

.

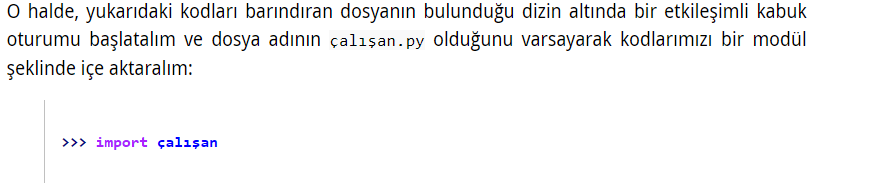
.

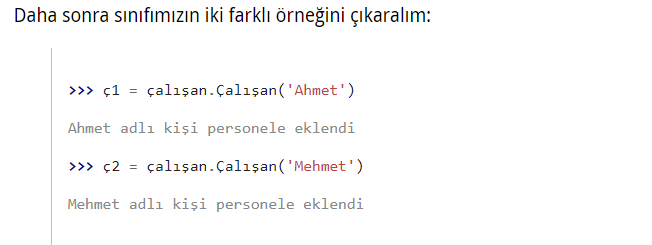
Devam

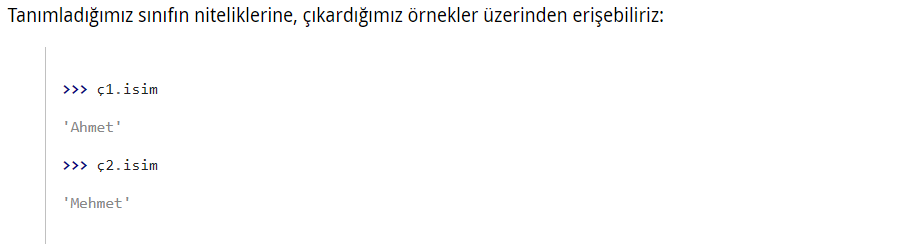
.

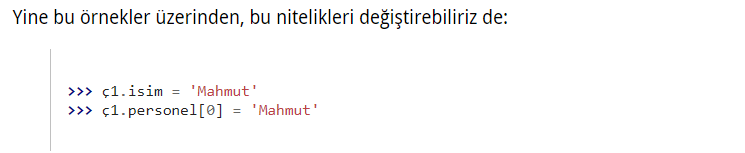
.

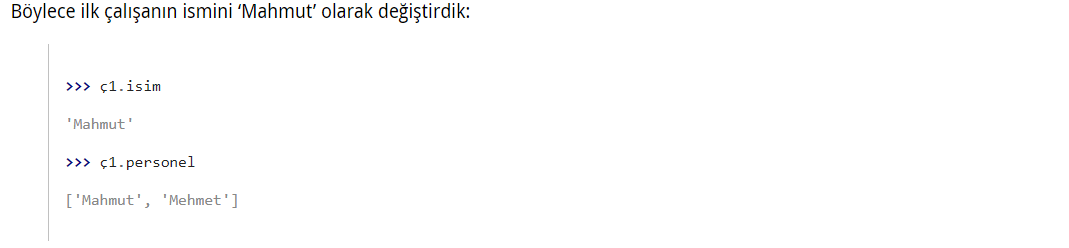
.

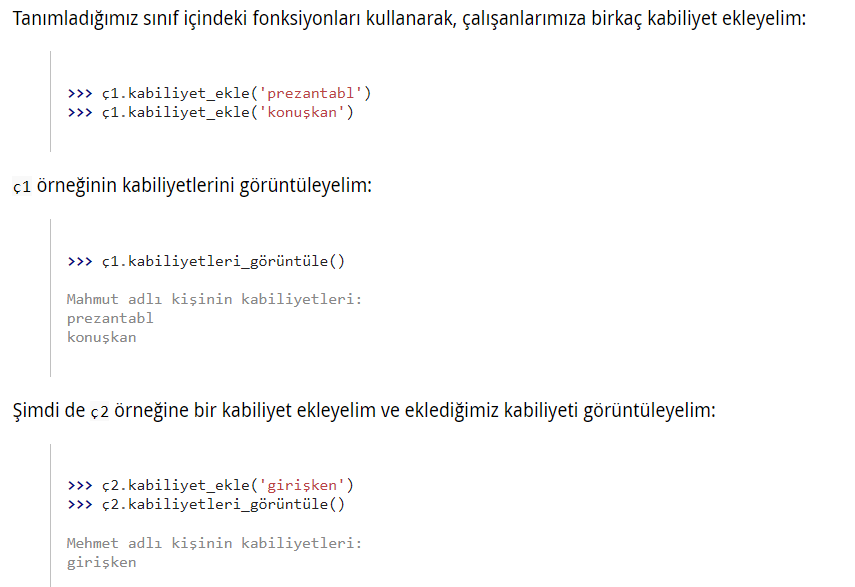














**Örnek**



**NOT:**

if \_\_Name\_\_ == ‘\_\_main\_\_’ YAPISI

<https://medium.com/@buraksekili0/python-name-main-yap%C4%B1s%C4%B1-e4a083b5723e>

\_\_name\_\_ yapısı, kodları derlediğiniz belgenin adını baz alan bir modüldür.

Normal standartlar altında çalıştırılan belgelerin adı “\_\_main\_\_”

Fakat belirli bir modül altında çalışan belgeler, içerisinde bulunduğu modülün adını alır.

Print(\_\_name\_\_) modülünü kullanarak test edersen eğer üzerinde çalıştığın belgenin adını göreceksindir.

Eğer normal standartlarda bir belge ise adı \_\_main\_\_

Değilse, modülün adıdır.

Örneğin Çalışanlar.py modülünü import etmişsen artık dosyanın adı

\_\_name\_\_ = ‘Çalışanlar.py’ olacaktır.

**Sınıf Metotları** (class methods)

Sınıfın herhangi bir örneğine bağlı olmayan bir işlem yapan

ama anlamsal olarak da sınıfla ilişkili olduğu için sınıf dışında bırakmak istemediğiniz fonksiyonları birer sınıf metodu olarak tanımlayabiliriz.

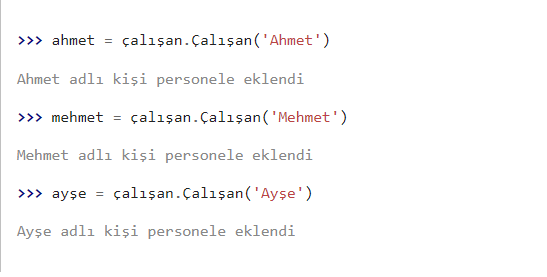
**Örnek**

Eğer örnek olarak kullandığımız Class’ımızın içerisine Personel Sayısını gösteren bir Örnek Metodu eklemek istersek

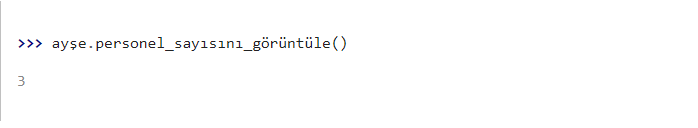
Personel = [ ] = Sınıf Niteliğidir



Bu örnek metodunu kullanmak için mutlaka bir örnek tanımlamak zorundayız.



Personel Sayısını gösteren fonksiyonu çalıştırmak için ise şöyle bir kullanım yapmak zorundayız.



**Not:**

1. Eğer Personeller listemizde hiçbir personel mevcut değilse, bu fonksiyonu çalıştıramayız.
2. Mevcut olsa dahi bu Fonksiyonu kullanmak için Örnek Metot üzerinden bir işlem yapmak zorundayız

Halbuki bizim yaptığımız iş Sınıf’ın tamamını ilgilendiren bir özellik.

Tanım: Sınıfın herhangi bir örneğine bağlı olmayan bir işlem yapan

ama anlamsal olarak da sınıfla ilişkili olduğu için sınıf dışında bırakmak istemediğiniz fonksiyonları birer sınıf metodu olarak tanımlayabiliriz.

Bu sebeplerden dolayı Örnek Metotlarından sonra Sınıf Metotlarına ihtiyaç duyuyoruz.

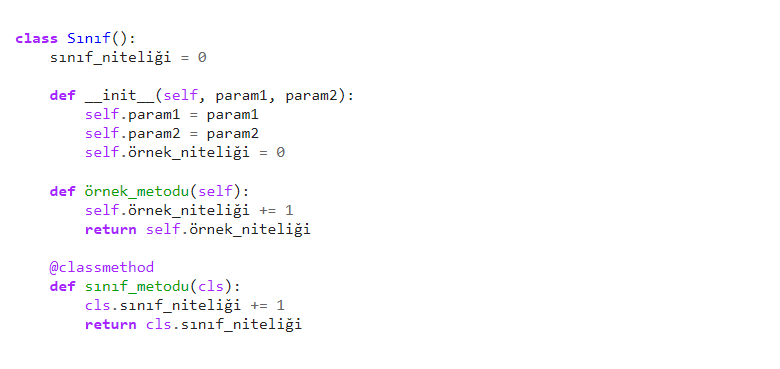
**Sınıf Metodu** (class methods) **Tanımlamak ve Kullanmak**

Örnek Metotları için gerekli olan şeyler def \_\_init\_\_ fonksiyonu ve Self komutuydu

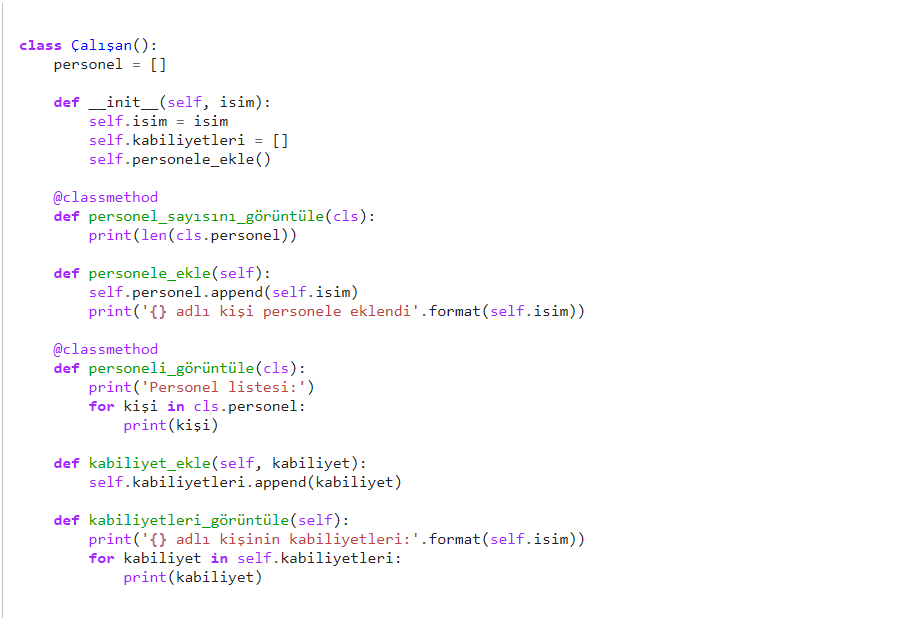
Sınıf Metotları için gerekli olan şeyler ise

@classmethod ve cls

**Örnek**



**Örnek**



**Not:**

Yukarıda personel\_sayısını\_görüntüle() adlı fonksiyonun yanı sıra, personeli\_görüntüle() adlı fonksiyonu da bir sınıf metodu haline getirdik.

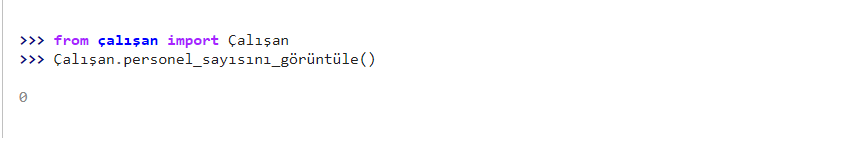
Çünkü tıpkı personel\_sayısını\_görüntüle() fonksiyonu gibi, personeli\_görüntüle() fonksiyonu da aslında tek tek örneklerden ziyade sınıfın genelini ilgilendiriyor.

Dolayısıyla bu fonksiyona da sınıf adı üzerinden erişebilmek gayet makul ve mantıklı bir iştir.

.

.

**Kullanımı**



**Alternatif İnşacılar** (alternative constructor)

Sınıf metotlarının, işimize yarayabilecek bir başka özelliği ise, bunların bir ‘alternatif inşacı’ (alternative constructor) olarak kullanılabilecek olmasıdır.

[**https://python-istihza.yazbel.com/nesne\_tabanli\_programlama2.html#alternatif-insacilar**](https://python-istihza.yazbel.com/nesne_tabanli_programlama2.html#alternatif-insacilar)

**Statik Metotlar**

Python’da örnek metotları ve sınıf metotları dışında bir de statik metotlar bulunur.

Bildiğiniz gibi, örnek nitelikleri üzerinde işlem yapacağımız zaman örnek metotlarını kullanıyoruz.

Aynı şekilde sınıf nitelikleri üzerinde işlem yapacağımız zaman ise sınıf metotlarından faydalanıyoruz.

Örnek metotları içinde herhangi bir örnek niteliğine erişmek istediğimizde self kelimesini kullanıyoruz.

Sınıf metotları içinde bir sınıf niteliğine erişmek için ise cls kelimesini kullanıyoruz.

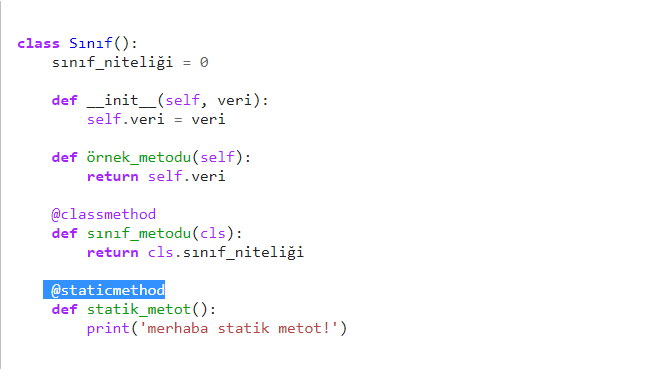
Eğer bir sınıf içindeki herhangi bir fonksiyonda örnek veya sınıf niteliklerinin hiçbirine erişmeniz gerekmiyorsa, statik metotları kullanabilirsiniz.

**@staticmethod Bezeyicisi**

Burada örnek\_metodu(), self yardımıyla örnek niteliklerine erişiyor. sınıf\_metodu() ise cls yardımıyla sınıf niteliklerine erişiyor.

Sınıf metodu tanımlamak için ayrıca @classmethod bezeyicisini de kullanıyoruz.

İşte eğer sınıf içinde tanımlayacağınız fonksiyon herhangi bir örnek ya da sınıf niteliği üzerinde herhangi bir işlem yapmayacaksa şöyle bir şey yazabilirsiniz:



**Not:** Gördüğünüz gibi, statik metotları tanımlamak için @staticmethod bezeyicisini kullanıyoruz.

Statik metotlar, ilk parametre olarak self veya cls benzeri kelimeler almaz.

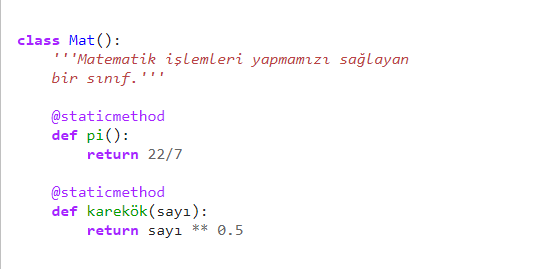
Çünkü bu tür sınıfların örnek veya sınıf nitelikleri ile herhangi bir işi yoktur.

Peki statik metotlar ne işe yarar?

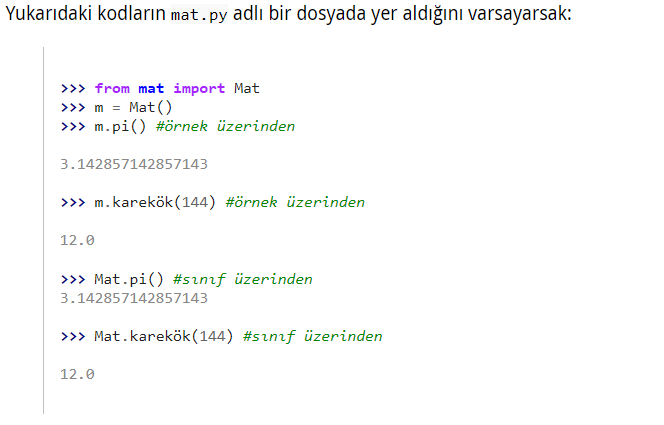
Bu metotlar sınıf metotlarına çok benzer.

Tıpkı sınıf metotlarında olduğu gibi, anlamsal olarak sınıfla ilgili olan, ancak sınıf metotlarının aksine bu sınıfın herhangi bir niteliğine erişmesine gerek olmayan fonksiyonları, sınıf dışına atmak yerine, birer statik metot olarak sınıf içine yerleştirebiliriz.

Örnek



**Not:** Statik metotları hem örnekler hem de sınıf adları üzerinden kullanabiliriz.



**Not:** Statik Metotlar ile yapılacak işlemleri Örnek veya Sınıf metotları ile de yapabiliriz.

Özellikle Statik Metotlar ve Sınıf Metotları birbirine çok benzer fakat gerek olmadığı halde ‘cls’ kullanmış oluruz

Statik Metotlar çok sık kullanılan araçlar değildir fakat bilmekte yarar var.

# **Nesne Tabanlı Programlama (Devamı)**

**Nesneler**

Kabaca, Python’da belli birtakım metotlara ve/veya niteliklere sahip olan öğelere nesne adı verilir.

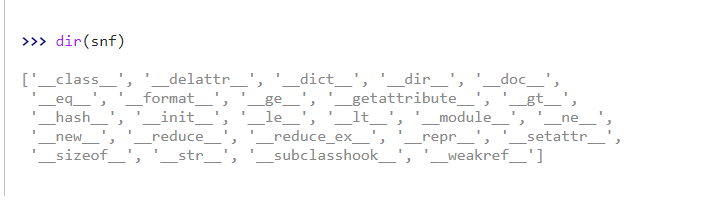
Yani ‘nesne’ kelimesi, içinde birtakım metot ve/veya nitelikler barındıran öğeleri tanımlamak için kullanılan bir tabirden, basit bir isimlendirmeden ibarettir.

**Örnek**



İşte bu kodlardaki sınıf = Sınıf() komutu ile bir nesne üretmiş olduk.

Nesnemizin adı da ‘sınıf’. Teknik olarak ifade edersek, sınıf örneği, Sınıf() adlı sınıfın bütün nitelik ve metotlarını bünyesinde barındıran bir nesnedir.

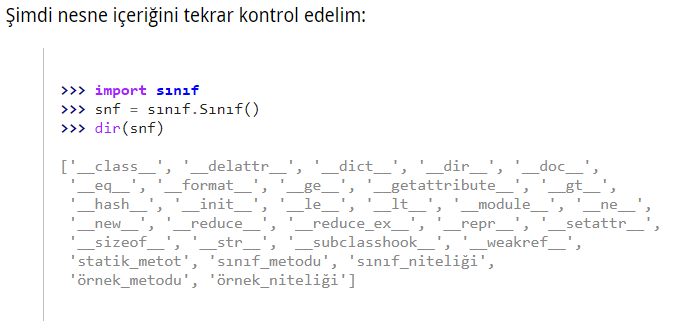


Gördüğünüz gibi, biz boş bir sınıf tanımlamış olsak da, snf nesnesi öntanımlı olarak yine de bazı nitelik ve metotlara sahip.

İşte Python’da, yukarıdaki gibi birtakım nitelik ve metotlara sahip olan bu tür öğelere ‘nesne’ adı veriyoruz.

**Örnek**





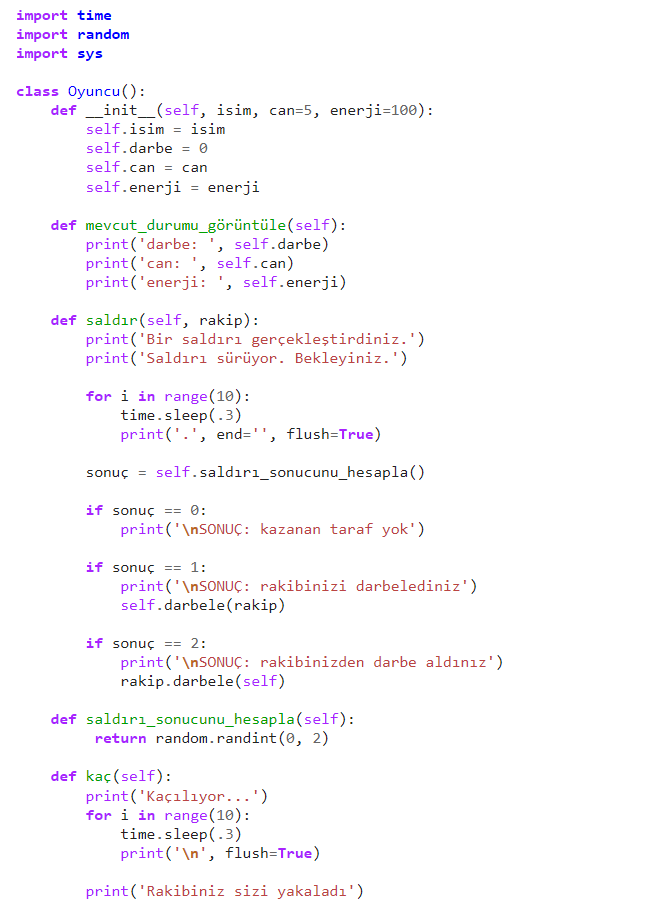
Gördüğünüz gibi, kendi tanımladığımız nitelik ve metotlar da snf adlı nesne içine eklenmiş…

İşte snf adlı sınıf örneğinin,

yukarıda gösterilen birtakım durum ve davranışlara sahip olmasından yola çıkarak, snf örneğinin bir nesne olduğunu söylüyoruz.

**Tanım:** Python’da belli bir durumu/niteliği/metodu/davranışı olan elemanlara/öğelere nesne (object) adı veriyoruz.

**Oyun Bilgisi**





**Her şey bir nesnedir**

Python’da her şey bir nesnedir. Gerçekten de (if, def, and, or gibi deyim ve işleçler hariç) Python’da her şey bir nesnedir.

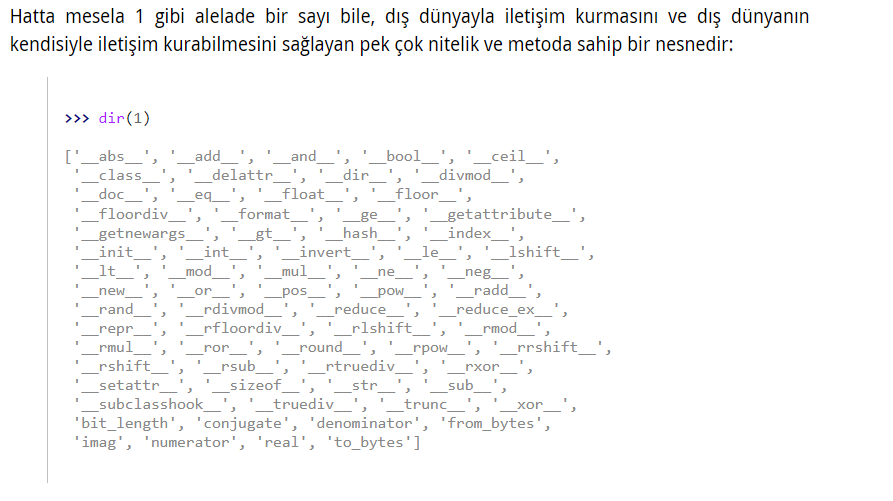
Hatırlarsanız nesnenin ne olduğunu tanımlarken,

belli bir durumda bulunan ve belli birtakım davranışları olan öğelere nesne adı verildiğini söylemiştik.

İşte Python’daki her şey, bu tanım doğrultusunda bir nesnedir.







İşte konuya bu noktadan baktığımızda, Python’da her şey bir nesnedir.

Yani Python’daki her şeyle, sahip oldukları metotlar ve nitelikler aracılığıyla etkileşebilirsiniz.

Python’ın bu özelliğini bilmek, muhatap olduğunuz programlama dilini ve onun kabiliyetlerini tanımak açısından önemlidir.

Python’da her şeyin bir nesne olduğunu anladığınız anda,

{'a': 0, 'b': 1} gibi bir kodla yalnızca basit bir sözlük tanımlamadığınızı, bunun arka planında, bu sözlükle etkileşim kurmanızı sağlayacak koca bir mekanizma bulunduğunu bilirsiniz

**Birinci Sınıf Öğeler** (first Class)

Tüm ögelerin, anı özellikleri ve işlevleri gerçekleştirebiliyor olması durumudur.

Örneğin, fonksiyonlar da bir değişkene atanabilir, listeler de bir değişkene atanabilir.

Bu durumda iki öge de aynı özellikleri barındırmaktadır.

Yani eşit özelliktedirler.

# **Nesne Tabanlı Programlama (Devamı)**

**Sınıf Üyeleri**

Python’da bir sınıf içinde bulunan nitelikler, değişkenler, metotlar, fonksiyonlar ve buna benzer başka veri tipleri, o sınıfın üyelerini meydana getirir.

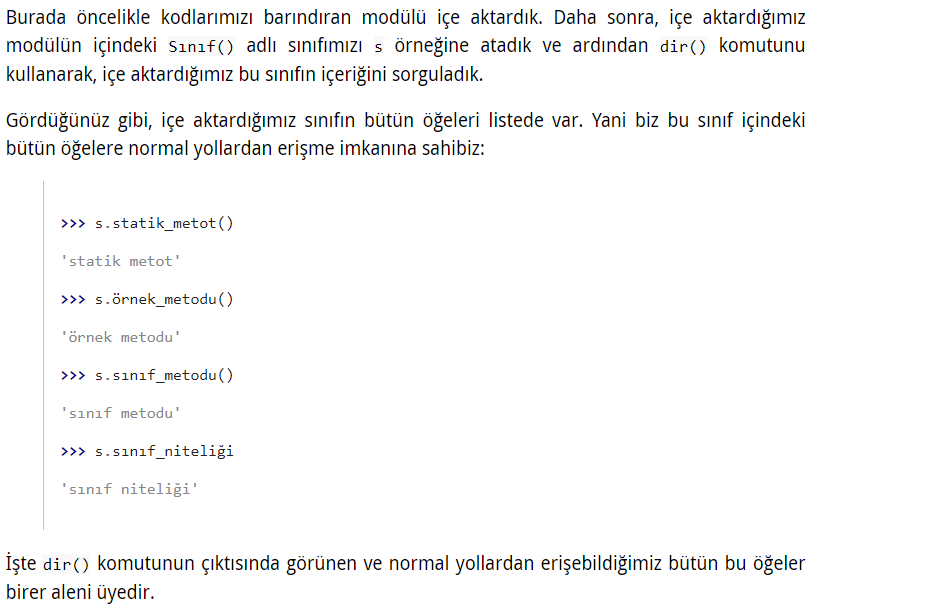
Bir sınıfın üyelerini genel olarak üçe ayırarak inceleyebiliriz:

* Aleni üyeler (public members)
* Gizli üyeler (private members)
* Yarı-gizli üyeler (semi-private members).

**Aleni Üyeler**

Eğer bir sınıf üyesi dışarıya açıksa, yani bu üyeye sınıf dışından normal yöntemlerle erişilebiliyorsa bu tür üyelere ‘aleni üyeler’ adı verilir.

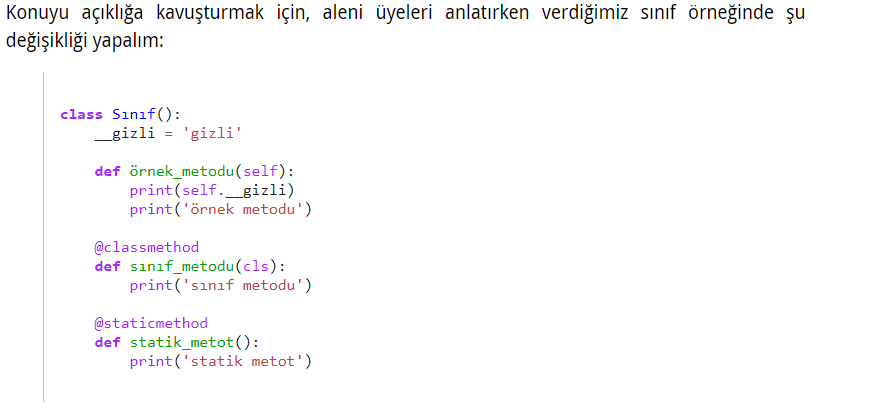




**Gizli Üyeler**

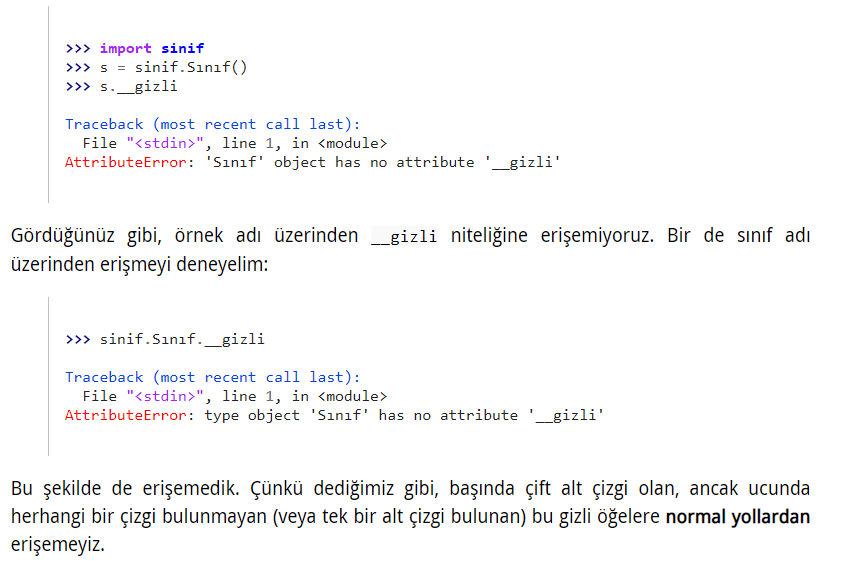
Eğer kodlarınızda, sınıfın yalnızca iç işleyişini ilgilendiren, bu yüzden de dışarıdan erişilmesine gerek olmadığını veya erişilirse problem çıkacağını düşündüğünüz birtakım öğeler varsa bunları dışarıya kapatarak bir ‘gizli üye’ haline getirmek isteyebilirsiniz.

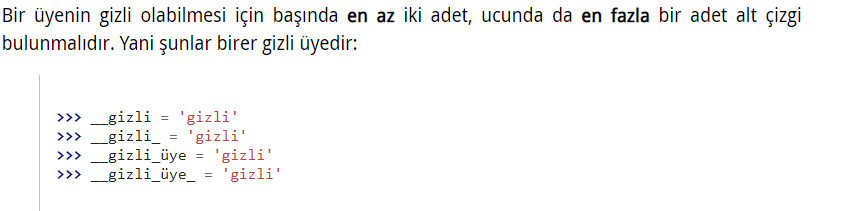
Aleni üyelerin aksine gizli üyeler dışarıya açık değildir. Gizli üyelere, normal yöntemleri kullanarak sınıf dışından erişemeyiz.



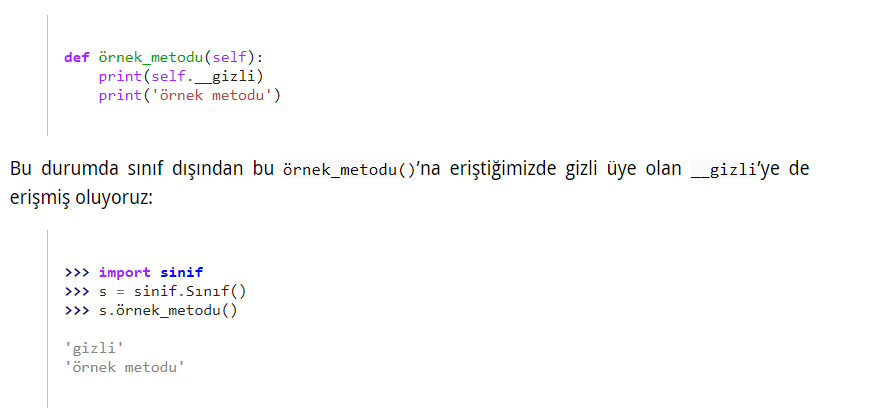
Python’da baş tarafında yukarıdaki gibi iki adet alt çizgi olan, ancak uç tarafında alt çizgi bulunmayan (veya yalnızca tek bir alt çizgi bulunan) bütün öğeler birer gizli üyedir

Dışarıya kapalı olan bu gizli üyelere, normal yöntemleri kullanarak sınıf dışından erişemezsiniz.



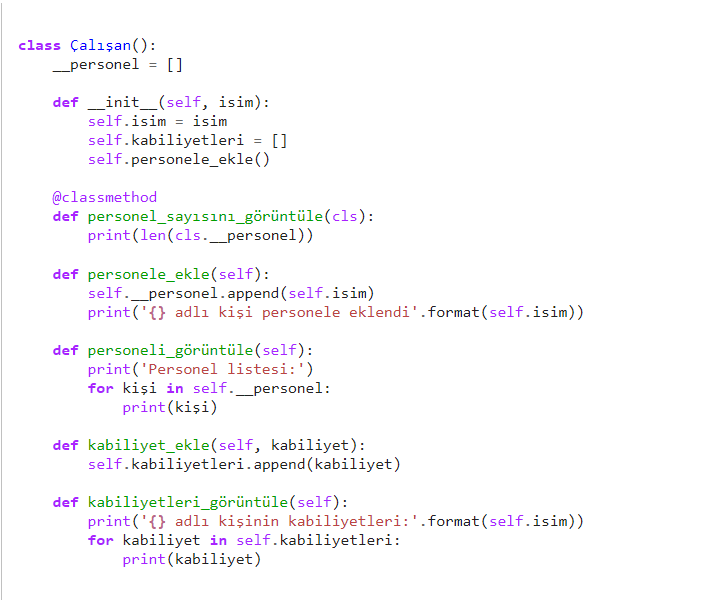


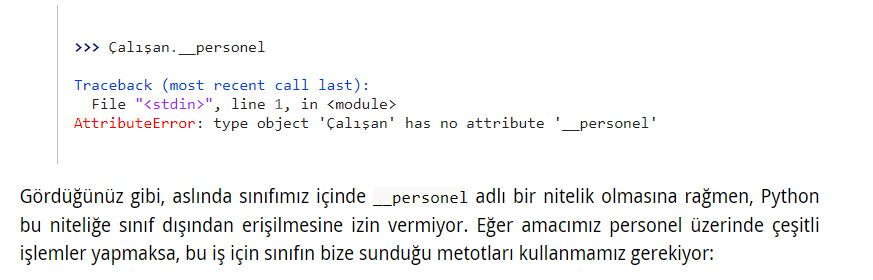
**NOT:** Gizli üyeler yalnızca **sınıf dışına** kapalıdır. Bu üyelere **sınıf içinden** rahatlıkla erişebiliriz.

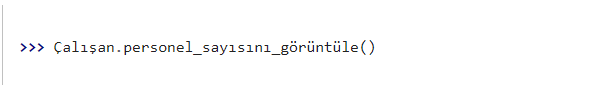


**Neden gizli üye kullanırız?**

Bu kodlarda yer alan personel listesinin usulsüz bir şekilde kullanılmasını önlemek amacıyla bu listeyi bir gizli üye haline getirebilirsiniz:







**NOT:** Gizli üyelere erişmek için Örnek Metotları veya Sınıf Metotlarını kullanabiliriz.

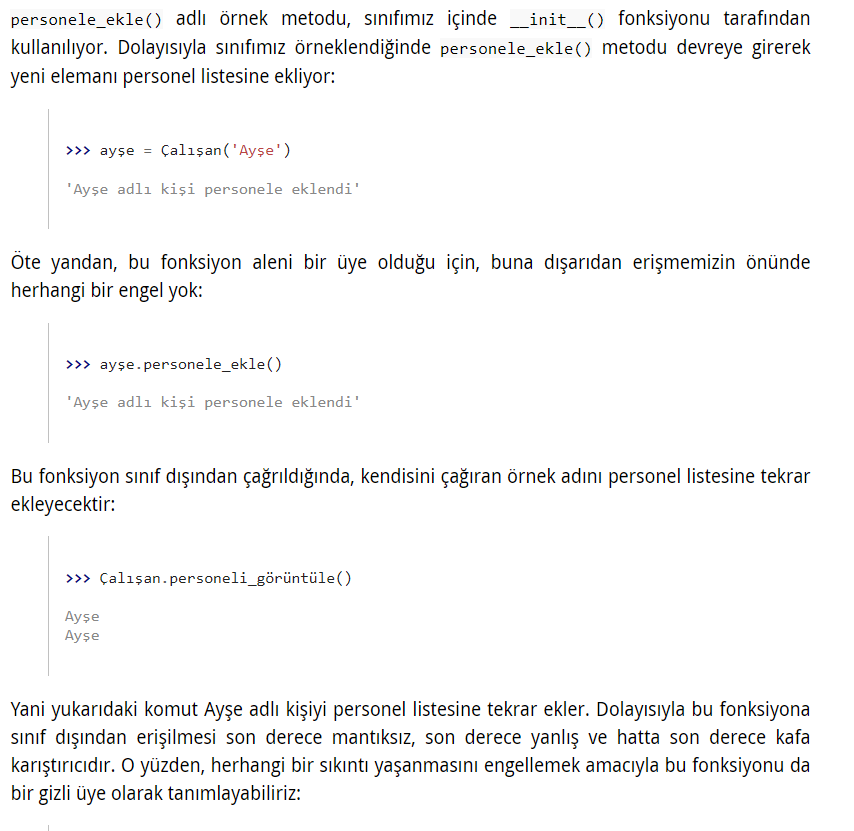
Metotların içerisine dahil eder ve sonrasında bu Metotları import ederek çağırabiliriz.

Böylece gizli üyelere erişmiş oluruz.

.

.

.





Bu şekilde personele\_ekle() fonksiyonunu da dışarıya kapatmış olduk. Artık bu fonksiyon da, olması gerektiği gibi, yalnızca sınıf içinde kullanılabilecek.

**İsim Bulandırma**  (name mangling)

Python’da ‘gizli’ olarak adlandırdığımız öğeler aslında o kadar da gizli değildir…

Çünkü Python’da gerçek anlamda gizli ve dışarıya tamamen kapalı üyeler bulunmaz.

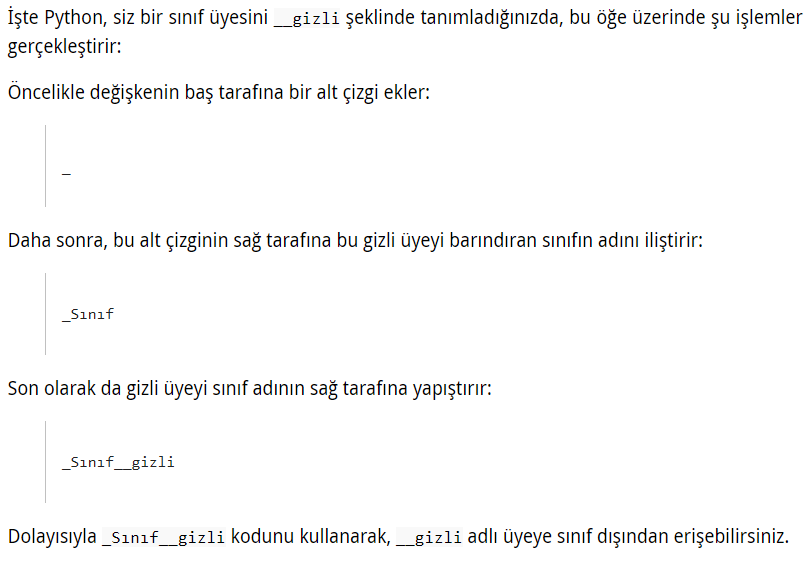
Her ne kadar yukarıdaki örneklerde üyeleri dışarıya kapatmak için kullandığımız alt çizgi işaretleri ilgili değişkeni gizlese de, bunu tamamen erişilmez hale getirmez.

Dediğimiz gibi, Python’da gerçek anlamda dışa kapalı sınıf üyeleri bulunmadığı için biz bu üyelere **bir şekilde** erişme imkanına sahibiz.

Python, kodlar içinde gizli bir üye ile karşılaştığında özel bir ‘isim bulandırma’ (name mangling) işlemi gerçekleştirir ve ilgili gizli üyenin görünüşünü değiştirir.

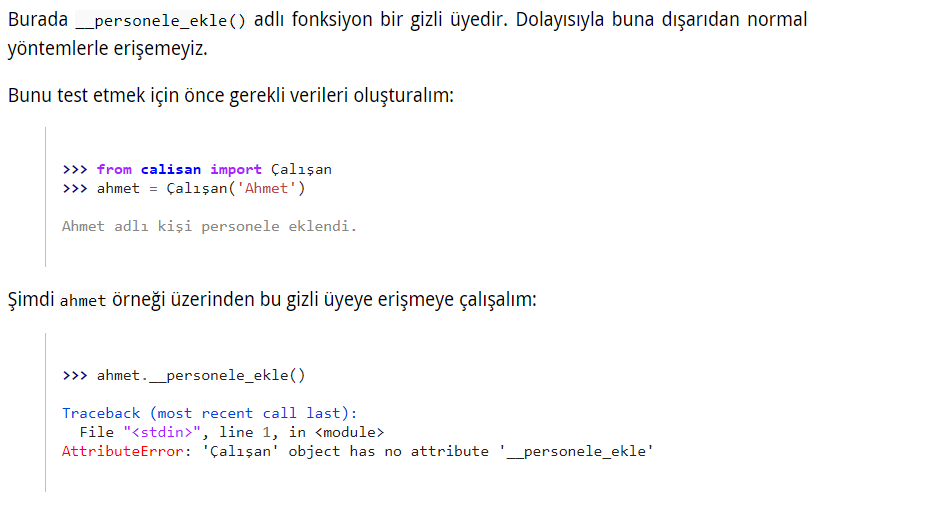


.



**Örnek2**



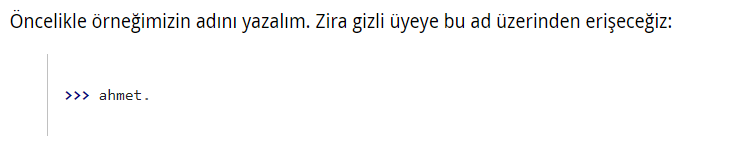


-Gördüğünüz gibi, Python bu üyeye normal yollardan erişmemize izin vermiyor.

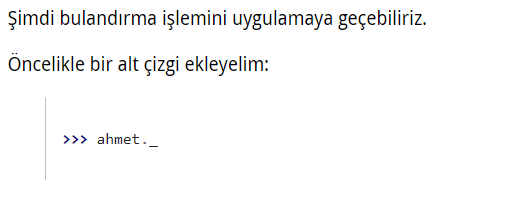
Ama biz biliyoruz ki, Python bu üyeyi gizlerken özel bir isim bulandırma işlemi gerçekleştiriyor.

Bu bulandırma işleminin nasıl gerçekleştirildiğini bildiğimize göre gizli üyeye erişebiliriz.

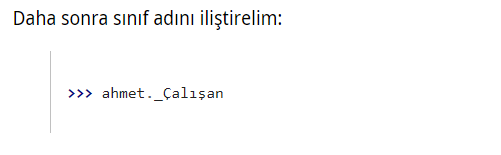
1. Adım



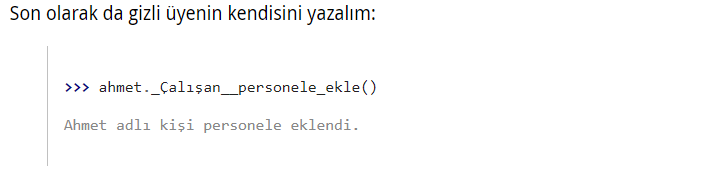
1. Adım



1. Adım

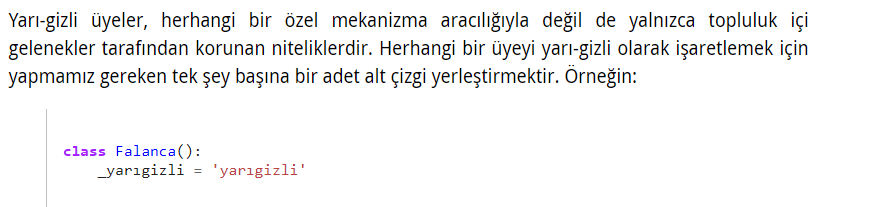


1. Adım



**Yarı – Gizli Üyeler** (semi-private members)

Python’da bir de yalnızca topluluk içi gelenekler tarafından korunan ‘yarı-gizli’ üyeler (semi-private members) vardır. İşte bu bölümde, bir gizli üye türü olan yarı-gizli üyelerden söz edeceğiz.



**NOT:**

Buradaki \_yarıgizli adlı niteliğe sınıf içinden veya dışından erişmemizi engelleyen veya zorlaştıran hiçbir mekanizma bulunmaz.

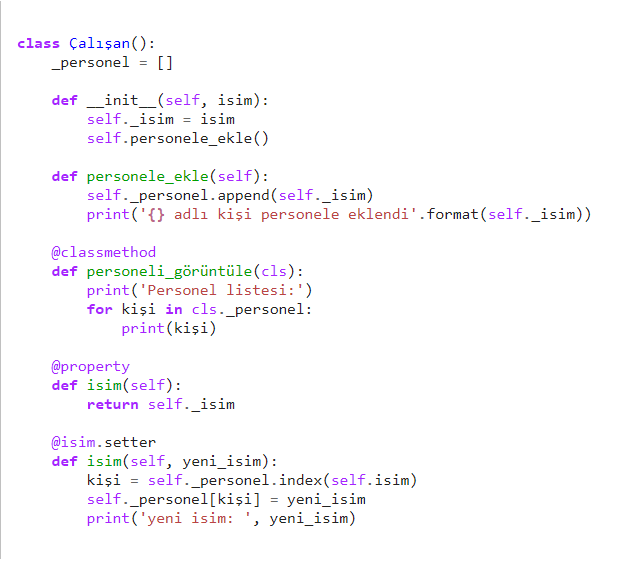
Ama biz bir sınıf içinde tek alt çizgi ile başlayan bir öğe gördüğümüzde, bunun sınıfın iç işleyişine ilişkin bir ayrıntı olduğunu, sınıf dışından bu öğeyi değiştirmeye kalkışmamamız gerektiğini anlarız.

**@property Bezeyicisi**

Pythonda Class içerisindeki bir değişkeni (name = “Onur”) değiştirmek istersek eğer,

Yaptığımız bu değişiklik, Class’ın içinde ismi tutan **Liste de** değişime uğramayacaktır.

Bu şekilde bir değişikliğin, Liste içerisinde de gerçekleşmesi için @proptery bezeyicisi kullanılmaktadır.



-İlk olarak property edeceğimiz değişkeni, Classta gösterdik

Ardından ise @isim.setter (Ayarlamak) şeklinde bir gösterge ile fonksiyonun ne yapacağını kodladık

Bu işlemden sonra, Hem değişkenin index’i hem de kendisi uygun şekilde Python tarafından değiştirilecektir.