

Kendi Özel Sınıflarınızı Oluşturma

- Sınıflar yeni veri tipleridir
- Oluşturacağınız çoğu uygulama genellikle özel sınıflar kullanmaz veya sadece birkaçını kullanır
- Endüstride bir geliştirme ekibinin bir parçası haline gelirsiniz, binlerce hatta binlerce sınıf içeren uygulamalar üzerinde çalışabilirsiniz

Kalıtım (Inheritance)

- Yeni bir sınıf oluştururken, önceden tanımlanmış bir temel sınıfın (aynı zamanda bir üst sınıf olarak da adlandırılır) özelliklerini (değişkenler) ve yöntemlerini (fonksiyonların sınıf versiyonları) devralabilirsiniz.
- Yeni sınıf türetilmiş sınıf (veya alt sınıf) olarak adlandırılır.
- Daha sonra türetilmiş sınıfı, uygulamanızın belirli ihtiyaçlarını karşılamak için özelleştirebilirsiniz.

▼ Çok biçimlilik (Polymorphism)

- Sizi "özel durumda" değil, "genel durumda" programlamaya olanak tanır
- Aynı yöntem çağrısını farklı tiplere ait nesnelere gönderebilirsiniz
- Her bir nesne, "doğru işlemi" yaparak yanıt verir
- Bu nedenle, aynı yöntem çağrısı "çoklu biçim" alır, bu yüzden "polimorfizm" terimi kullanılır.

```
1 class Sinif1:
2     # yapıcı (constructor) sınıf ilk oluşturulan yapılacak işlemlerin tanımlandığı özel bir metottur.
3     # bu sınıfın nesne/örnek oluştururken oluştururken çağrılan ilk metottur.
4     def __init__(self,c):
5         # self nesnenin kendisi yani this diğer dillerdeki this parametresinin karşılığıdır
6         self.x=c #nesnenin x değişkeninin değeri
7     def __repr__(self):
8         return f"repr ben {self.x} "
9
10    def __str__(self):
11        return f"str  ben {self.x} "
12
13
14
```

```
1 d1=Sinif1(10)
2 print("adres",d1) # nesnenin adresini yazar
3 x=5
4 print("nesnenin x değeri ",d1.x) # nesnenin x değişkenini yazar → (10)
5 d2=Sinif1(10)
6 print(d2) # → __str__
7 d2 # __repr__
```

```
adres str  ben 10
nesnenin x değeri  10
str  ben 10
repr ben 10
```

```
1 d2.y=20 # bu şekilde tanımlanabilir
2 d2.y
```

```
20
```

```
1
```

```
1 # kalıtım (Inheritance)
2 class Sinif2(Sinif1):
3     def __init__(self):
4         pass
```

```
1
```

```
1 l=[1,2,3]
2 dir(l) # __metod__ olanlar özel metodlardır
```

```
['_add_',
 '_class_',
 '_class_getitem_',
 '_contains_',
 '_delattr_',
 '_delitem_',
```

```

['__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__gt__',
 '__hash__',
 '__iadd__',
 '__imul__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__reversed__',
 '__rmul__',
 '__setattr__',
 '__setitem__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'append',
 'clear',
 'copy',
 'count',
 'extend',
 'index',
 'insert',
 'pop',
 'remove',
 'reverse',
 'sort']

```

```

1 class Deneme:
2     pass
3 if __name__ == '__main__':
4     x=Deneme()
5     print(x) # tip ve adress
6     print(type(x)) # class tipi → <class '__main__.Deneme'>
7     print(x.__class__.__name__)# → Deneme
8
9     print(dir(x))
10

```

```

<__main__.Deneme object at 0x000001DC3C9C9280>
<class '__main__.Deneme'>
Deneme
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__

```

- Person Sınıfı Oluşturulur
- Sınıfın `__init__` metodu, sınıfın bir örneği oluşturulduğunda otomatik olarak çağrılır.
- fonk adlı bir fonksiyonu vardır. Bu fonksiyon, sınıf örneği tarafından çağrılır ve merhaba yazdırır.
- `@classmethod` ifadesinden sonra `fonk2` adında bir sınıf metodu tanımlanmıştır. Bu metod, sınıfın kendisine uygulanan işlemler için kullanılır. `cls` adında bir argüman alır ve bu argüman, sınıfın kendisini temsil eder.
- Özetle, `Deneme` sınıfı, bir örnek niteliği olan `isim` ve örnek fonksiyon `fonk` içeren bir sınıftır. Sınıf ayrıca, sınıf niteliği `a` ve sınıf fonksiyonu `fonk2` içerir.

```

1 class Person():
2     a = 0 # sınıf niteliği
3
4     def __init__(self, isim):
5         self.isim = isim # örnek niteliği
6         Person.a += 1
7
8
9
10    def fonk(self): # örnek yöntemi
11        print("merhaba")
12
13    @classmethod # sınıf yöntemi
14    def fonk2(cls, string):
15        isim, age = string.split(",")

```

```

16     if not hasattr(Person, 'age'):
17         Person.age = []
18     if not hasattr(Person, 'isim'):
19         Person.isim = []
20     Person.age.append(age)
21     Person.isim.append(isim)
22
23     return cls(isim)
24 @classmethod # sınıf metodu
25 def yaz(cls):
26     print(f"yaşlar : {Person.age}")
27     print(f"isimler : {Person.isim}")
28     print(f"a : {Person.a}")

```

```

1 d_obj1=Person.fonk2("damla,18")
2 d_obj2=Person.fonk2("Sami,19")
3 d_obj3=Person.fonk2("cafer,20")
4 Person.yaz()
5 d_obj1.fonk()
6 d_obj2.isim

```

```

yaşlar : ['18', '19', '20']
isimler : ['damla', 'Sami', 'cafer']
a : 3
merhaba
'Sami'

```

```

1 print(Person.a)
2 x=Person('Ali')
3 print(Person.a)
4 y=Person('Veli')
5 print(Person.a)

```

```

3
4
5

```

```
1 y # y=Person('Veli')
```

```
<__main__.Person at 0x1dc3d2078e0>
```

```
1 x.isim # x=Person('Ali')
```

```
'Ali'
```

```

1 x.soyisim="DENEME" # olmayan parametreyi oluşturduk
2 x.soyisim
3

```

```
'DENEME'
```

```
1 y.soyisim # hata verir → y nin soyisim değişkeni yoktur
```

```

-----
-
AttributeError                                Traceback (most recent call
last)
~\AppData\Local\Temp\ipykernel_9484\3175941016.py in <module>
----> 1 y.soyisim # hata verir → y nin soyisim değişkeni yoktur

AttributeError: 'Person' object has no attribute 'soyisim'

```

```
1 x.fonk() # Person sınıfının fonk fonksiyonu çağırır
```

```
merhaba
```

```

1 x.a=13
2 print(x.__dict__)
3 print(Deneme.__dict__)

```

```

{'isim': 'Ali', 'soyisim': 'DENEME', 'a': 13}
{'__module__': '__main__', '__dict__': <attribute '__dict__' of 'Deneme' objects>, '__weakref__': <attribute '__weakref__'

```

10.2.1 Class Account'ı Test Etmek

- Oluşturduğunuz her yeni sınıf yeni bir veri tipi haline gelir.

- Python, genişletilebilir bir dil'dir.
- Account sınıfının tanımına bakmadan önce, yeteneklerini gösterelim.

```
1 """
2 import account
3 dir(account)# Account sınıfı yok ??
4 """
5
6 # bunun yerine kendi accoun sınıfımızı oluşturalım

\nimport account\ndir(account)# Account sınıfı yok ?? \n'
```

▼ Create an **Account** Object with a Constructor Expression

- Bir nesne oluşturun ve onu oluşturan ve başlatan **initialize edecek** bir Oluşturucu **constructor** kullanın
- Oluşturucu ifadeler, parantezlerle belirtilen bağımsız değişken kullanarak yeni nesneler oluşturur ve verilerini başlatır.
- Sınıf adından sonra argüman olmasa bile parantezler gereklidir.

```
1 from decimal import Decimal
2 class Account:
3     """Account class for maintaining a bank account balance."""
4
5     def __init__(self, name, balance):
6         """Initialize an Account object."""
7
8         # if balance is less than 0.00, raise an exception
9         if balance < Decimal('0.00'):
10             raise ValueError('Initial balance must be ≥ to 0.00.')
11
12         self.name = name
13         self.balance = balance
14
15     def balance(self):
16         return self.balance
17
18     def deposit(self, amount):
19         #eğer miktar negatif ise hata mesajı verir
20         if amount < Decimal('0.00'):
21             raise ValueError('miktar pozitif olmalıdır')
22         self.balance += amount
```

Double-click (or enter) to edit

```
1 dir(Account)# özel metodlarına bakalım
```

```
['_class_',
 '_delattr_',
 '_dict_',
 '_dir_',
 '_doc_',
 '_eq_',
 '_format_',
 '_ge_',
 '_getattribute_',
 '_gt_',
 '_hash_',
 '_init_',
 '_init_subclass_',
 '_le_',
 '_lt_',
 '_module_',
 '_ne_',
 '_new_',
 '_reduce_',
 '_reduce_ex_',
 '_repr_',
 '_setattr_',
 '_sizeof_',
 '_str_',
 '_subclasshook_',
 '_weakref_',
 'balance',
 'deposit']
```

```
1
```

```
1 account1 = Account('John Green', Decimal('50.00'))
```

▼ Hesabın Adını ve Bakiyesini erişme (Getting an Account's Name and Balance)

- Hesap nesnesinin adı ve bakiye niteliklerine erişin (Access the `Account` object's `name` and `balance` attributes)

```
1 account1.name
```

```
'John Green'
```

```
1 account1.balance
```

```
Decimal('50.00')
```

▼ Hesaba Para Yatırma (Depositing Money into an Account)

- Bu yöntem, pozitif dolar miktarını alır ve bunu hesap bakiyesine ekler. (`deposit` method receives a positive dollar amount and adds it to the balance)

```
1 account1.deposit(Decimal('25.53'))
```

```
1 account1.balance
```

```
Decimal('75.53')
```

▼ Account sınıfının yöntemleri, argümanları doğrularlar. (Account Methods Perform Validation)

- Hesap yöntemleri bağımsız değişkenlerini doğrular (`Account`'s methods validate their arguments)

```
1 account1.deposit(Decimal('-123.45')) # sadece pozitif değer alır
```

```
-----
-
ValueError                                Traceback (most recent call
last)
~\AppData\Local\Temp\ipykernel_9484\170039444.py in <module>
----> 1 account1.deposit(Decimal('-123.45')) # sadece pozitif değer alır

~\AppData\Local\Temp\ipykernel_9484\4030065187.py in deposit(self, amount)
    18         #eğer miktar negatif ise hata mesajı verir
    19         if amount < Decimal('0.00'):
--> 20             raise ValueError('miktar pozitif olmalıdır')
    21         self.balance += amount

ValueError: miktar pozitif olmalıdır
```

kalın metin### Sınıf tanımlaması (cont.)

- Her sınıf genellikle açıklayıcı bir dokümantasyon dizesi sağlar.
- Bu dize, sınıf başlığını takip eden satır veya satırlarda yer almalıdır.
- Python'da herhangi bir sınıfın dokümantasyon dizesini görüntülemek için, sınıf adını yazıp bir soru işareti ekleyerek ve ardından Enter tuşuna basarak sorgulayabilirsiniz.
- Örnek: `Account?`

▼ Sınıf tanımlaması (cont.)

- `Account`, hem sınıfın adıdır hem de bir yapıcı ifadesinde kullanılan `Account` nesnesini oluşturmak ve sınıfın `__init__` yöntemini çağırmak için kullanılan addır.
- Python'ın yardım mekanizması, hem sınıfın dokümantasyon dizesini ("Docstring:") hem de `__init__` yönteminin dokümantasyon dizesini ("Init docstring:") gösterir.

Account Nesnelerini Başlatma: `__init__` Yöntemi

- Bir yapılandırıcı ifadesi yeni bir nesne oluşturur ve ardından sınıfın `__init__` yöntemini çağırarak verilerini başlatır.
- Her yeni sınıf, bir nesnenin veri özelliklerini nasıl başlatacağını belirten bir `__init__` yöntemi sağlayabilir.
- `__init__` yönteminden `None` dışında bir değer döndürmek, `TypeError` hatasına neden olur.
- `Account` sınıfının `__init__` yöntemi, bir `Account` nesnesinin `name` ve `balance` özelliklerini başlatır, ancak `balance` geçerli bir değer ise.

Account Nesnelerini Başlatma: `__init__` Yöntemi (cont.)

```
def __init__(self, name, balance):
    """Initialize an Account object."""
```

```
# if balance is less than 0.00, raise an exception
if balance < Decimal('0.00'):
    raise ValueError('Initial balance must be ≥ to 0.00.')

self.name = name
self.balance = balance
```

Account Nesnelerini Başlatma: **init** Yöntemi (cont.)

- Bir nesne için bir yöntemi çağırdığınızda, Python bu nesneye bir başvuruyu (referansı) o yöntemin ilk argümanı olarak örtük olarak geçer.
- Bu nedenle, bir sınıfın tüm yöntemleri en az bir parametre belirtmelidir.
- Geleneksel olarak, bir yöntemin ilk parametresine self adı verilir.
- Yöntemler, bu referans (self) kullanarak nesnenin özelliklerine ve diğer yöntemlerine erişmelidir.

Account Nesnelerini Başlatma: **init** Yöntemi (cont.)

- Bir nesne oluşturulduğunda henüz herhangi bir özelliği (attribute) bulunmaz.
- Özellikler, aşağıdaki gibi atamalar aracılığıyla dinamik olarak eklenir:

```
self.attribute_name '=' value
```

Account Nesnelerini Başlatma: **init** Yöntemi(cont.)

*Python sınıfları, **init** gibi birçok özel yöntemi tanımlayabilir, (<https://docs.python.org/3/reference/datamodel.html#special-method-names>)

- Her biri yöntem adında önde ve arkada çift alt çizgi (__) ile belirtilir.
- Sınıf object, tüm Python nesneleri için geçerli olan özel yöntemleri tanımlar.

'deposit' Yöntemi

- Pozitif bir amount değerini hesabın balance özelliğine ekler.
- amount, 0.00'dan küçükse ValueError hatası oluşturur.

```
def deposit(self, amount):
    """Deposit money to the account."""

    # if amount is less than 0.00, raise an exception
    if amount < Decimal('0.00'):
        raise ValueError('amount must be positive.')

    self.balance += amount
```

10.2.3 Kompozisyon: Sınıfların Üyeleri Olarak Nesne Referansları

- Bir Account'un bir name'i vardır ve bir Account'un bir balance'ı vardır.
- Bir nesnenin özellikleri, diğer sınıfların nesnelere referanslardır.
- Başka türdeki nesnelere referansları gömmek, bazen kompozisyon olarak adlandırılan ve bazen de "bir ilişkisi" olarak adlandırılan yazılım yeniden kullanılabilirliğin bir formudur.

10.3 Özelliklere Erişimi Kontrol Etme

- Önceki örnekte, name ve balance özellikleri yalnızca bu özelliklerin değerlerini almak için kullanıldı.
- Bu özellikleri, değerlerini değiştirmek için de kullanabilirsiniz.

```
1 #from account import Account # import etmek için package eklenmelidir onun yerine classı biz tanımladık şimdilik
2 from decimal import Decimal
```

```
1 account1 = Account('John Green', Decimal('50.00'))
2 account1.balance #get methodu
```

```
Decimal('50.00')
```

- Set the **balance** attribute to an *invalid* negative value, then display the **balance**

```
1 account1.balance = Decimal('-1000.00') #set methodu
```

```
1 account1.balance
    Decimal('-1000.00')
```

Kapsülleme

- Bir sınıfın istemci kodu, sınıfın nesnelerini kullanan herhangi bir koddur.
- Çoğu nesne yönelimli programlama dili, bir nesnenin verilerini istemci kodundan gizlemek veya kapsüllemek için olanak sağlar.
 - *private data*

▼ Önde Gelen Alt Çizgi (_) İsimlendirme Kuralı

- Python'da özel veri kavramı yoktur.
- Doğru kullanımı teşvik eden sınıflar tasarlamak için isimlendirme kuralı kullanılır.
- Geleneksel olarak, Python programcıları herhangi bir alt çizgi (_) ile başlayan bir özniteliğin sadece sınıfın iç kullanımı için olduğunu bilir.
- Alt çizgi (_) ile başlamayan öznitelikler ise istemci kodunda genel olarak erişilebilir olarak kabul edilir.

```
1 class Deneme:
2     def __init__(self, a, b):
3         self.a=a #public a
4         self._a=a #semi private a
5         self.__a=a #private a
```

▼ 10.4 Time Sınıfı, Veri Erişimi için Özelliklerle

- Özellikler, bir nesnenin verilerine erişim ve değiştirme şeklini kontrol edebilirler - **programcılar belirli kurallara uydukları sürece**
- Güçlü tarih ve saat manipülasyon yetenekleri için, Python'un datetime modülüne bakın. [datetime module](#)

```
1
```

▼ 10.4.1 Test-Driving Class

Before we look at class `Time`'s definition, let's demonstrate its capabilities

Time Sınıfı Test Edilmesi

`Time` sınıfının tanımına bakmadan önce, yeteneklerini gösterelim.

```
1 # from timewithproperties import Time # import etmek için eklenmelidir # kendimi oluşturacağız simdilik
```

```
1 class Time:
2     """Class Time with read-write properties."""
3
4     def __init__(self, hour=0, minute=0, second=0):
5         """Initialize each attribute."""
6         self.hour = hour # 0-23
7         self.minute = minute # 0-59
8         self.second = second # 0-59
9
10    @property
11    def hour(self):
12        """Return the hour."""
13        return self._hour
14
15    @hour.setter
16    def hour(self, hour):
17        """Set the hour."""
18        if not (0 ≤ hour < 24):
19            raise ValueError(f'saat ({hour}) 0-23 aralığında olmalıdır')
20
21        self._hour = hour
22
23    @property
24    def minute(self):
25        """Return the minute."""
26        return self._minute
27
28    @minute.setter
29    def minute(self, minute):
30        """Set the minute."""
```

```

31     if not (0 ≤ minute < 60):
32         raise ValueError(f'Dakika ({minute}) araliginda olmalidir')
33
34     self._minute = minute
35
36     @property
37     def second(self):
38         """Return the second."""
39         return self._second
40
41     @second.setter
42     def second(self, second):
43         """Set the second."""
44         if not (0 ≤ second < 60):
45             raise ValueError(f'Saniye ({second}) 0-59 araliginda olmalidir')
46
47         self._second = second
48
49     def set_time(self, hour=0, minute=0, second=0):
50         """Set values of hour, minute, and second."""
51         self.hour = hour
52         self.minute = minute
53         self.second = second
54
55     def __repr__(self):
56         """Return Time string for repr()."""
57         return (f'Time(hour={self.hour}, minute={self.minute}, ' +
58                 f'second={self.second})')
59
60     def __str__(self):
61         """Return Time string in 12-hour clock format."""
62         return (('12' if self.hour in (0, 12) else str(self.hour % 12)) +
63                f':{self.minute:0>2}:{self.second:0>2}' +
64                (' AM' if self.hour < 12 else ' PM'))

```

```
1 dir(Time)
```

```

['_class__',
 '_delattr__',
 '_dict__',
 '_dir__',
 '_doc__',
 '_eq__',
 '_format__',
 '_ge__',
 '_getattr__',
 '_gt__',
 '_hash__',
 '_init__',
 '_init_subclass__',
 '_le__',
 '_lt__',
 '_module__',
 '_ne__',
 '_new__',
 '_reduce__',
 '_reduce_ex__',
 '_repr__',
 '_setattr__',
 '_sizeof__',
 '_str__',
 '_subclasshook__',
 '_weakref__',
 'hour',
 'minute',
 'second',
 'set_time']

```

Zaman (Time) Nesnesi Oluşturma

- Bir Time nesnesi oluşturma
- Time sınıfının **init** metodu, varsayılan olarak 0 değerine sahip olan hour (saat), minute (dakika) ve second (saniye) parametrelerine sahiptir.

▼ Bir Time Nesnesini Gösterme

- Time sınıfı, Time nesnesinin dize temsillerini oluşturan iki yöntem tanımlar.
- Python'da bir değişkeni değerlendirdiğinizde, nesnenin dize temsilini oluşturmak için nesnenin **repr** özel yöntemini çağırır.

```
1 wake_up = Time(hour=6, minute=30)
```



```
1 wake_up #__repr__
```

```
Time(hour=6, minute=30, second=0)
```

- **str** özel yöntemi, bir nesnenin bir dizeye dönüştürüldüğünde çağrılır, örneğin nesneyi print ile çıktı olarak verdiğinizde

```
1 print(wake_up)
```

```
6:30:00 AM
```

▼ Bir Özelliği Kullanarak Bir Özelliği Alma

- Time sınıfı, hour, minute ve second özelliklerini sağlar
 - Bir nesnenin verilerini alma ve değiştirme kolaylığı sağlar
 - Yöntem olarak uygulandığı için ek mantık içerebilir

```
1 wake_up.hour
```

```
6
```

- hour özelliği, bir hour veri özniteliğinin değerini döndüren bir yonteme benzer şekilde görünmektedir.
- Aslında, _hour veri özniteliğinin değerini döndüren bir _hour yöntemi çağırısıdır.

▼ Time'ı Ayarlama

- Time sınıfının set_time yöntemi, varsayılan olarak her biri 0 olan hour, minute ve second parametrelerini sağlar.

```
1 wake_up.set_time(hour=7, minute=45)
2 wake_up
```

```
Time(hour=7, minute=45, second=0)
```

▼ Bir Özelliği Kullanarak Bir Özniteliği Ayarlama

- Time sınıfı, özellikleri aracılığıyla hour, minute ve second değerlerini ayrı ayrı ayarlamayı da destekler.

```
1 wake_up.hour = 6
2 wake_up
```

```
Time(hour=6, minute=45, second=0)
```

- Görünüşe göre sadece bir veri özniteliğine değer atıyor gibi görünüyor
- Aslında hour yöntemi çağırısıdır ve 6 değerini bir argüman olarak alır, değeri doğrular ve _hour adında ilgili bir veri özniteliğine atar.

▼ Geçersiz Bir Değer Atama Girişimi

Sınıf Time'ın özelliklerinin atandıkları değerleri doğruladığını kanıtlamak için, hour özelliğine geçersiz bir değer atamayı deneyelim ve böylece bir ValueError hatası alalım:

```
1 wake_up.hour = 100
```

```
-----
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_9484\238335704.py in <module>
----> 1 wake_up.hour = 100

~\AppData\Local\Temp\ipykernel_9484\3813918088.py in hour(self, hour)
    17     """Set the hour."""
    18     if not (0 ≤ hour < 24):
--> 19         raise ValueError(f'saat ({hour}) 0-23 araliginda olmalidir')
    20
    21     self._hour = hour

ValueError: saat (100) 0-23 araliginda olmalidir
```

SEARCH STACK OVERFLOW

10.4.2 Time Sınıfı Tanımı

Sınıf Time: Varsayılan Parametre Değerleriyle **init** Metodu

- hour, minute ve second parametrelerini belirtir, her birinin varsayılan argümanı 0 olarak ayarlanmıştır.
- self.hour, self.minute ve self.second içeren ifadeler, yeni Time nesnesi (self) için hour, minute ve second öznitelikleri oluşturur gibi görünmektedir.
- Bu ifadeler aslında, sınıfın hour, minute ve second özelliklerini uygulayan yöntemleri çağırır.
- Bu yöntemler, _hour, _minute ve _second adında öznitelikler oluşturur.

timewithproperties.py

```
"""Class Time with read-write properties."""
```

```
class Time: """Class Time with read-write properties."""
```

```
def __init__(self, hour=0, minute=0, second=0):
    """Initialize each attribute."""
    self.hour = hour # 0-23
    self.minute = minute # 0-59
    self.second = second # 0-59
```

Sınıf Time: hour Okunabilir-Yazılabilir Özelliği

- hour adlı yöntemler, _hour adlı bir veri özniteliğini değiştiren okunabilir-yazılabilir bir özellik olan hour adlı bir özelliği tanımlar.
- Tek başına çift alt çizgi (..) kullanma kuralı, müşteri kodunun _hour özniteliğine doğrudan erişmemesi gerektiğini gösterir.
- Özellikler, Time nesneleriyle çalışan programcılara veri öznitelikleri gibi görünür, ancak yöntemler olarak uygulanır.
- Her özellik, bir veri özniteliğinin değerini alırken (döndürme) kullanılan bir getter yöntemi tanımlar.
- Her özellik, bir veri özniteliğinin değerini ayarlamak için (set etmek) isteğe bağlı olarak bir setter yöntemi tanımlayabilir.

```
@property def hour(self): """Return the hour.""" return self._hour
```

```
@hour.setter
def hour(self, hour):
    """Set the hour."""
    if not (0 ≤ hour < 24):
        raise ValueError(f'Hour ({hour}) must be 0-23')

    self._hour = hour
```

1

Sınıf Time: hour Okunabilir-Yazılabilir Özelliği (cont.)

- @property dekoratörü, özelliğin getter yönteminden önce yer alır ve yalnızca bir self parametresi alır.
- Bir dekoratör, dekore edilen işleve kod ekler.
 - hour işlevini öznitelik sözdizimiyle çalışır hale getirir.
- getter yönteminin adı özelliğin adıdır.

Sınıf Time: hour Okunabilir-Yazılabilir Özelliği (cont.)

- @property_name.setter (@hour.setter) şeklindeki bir dekoratör, özelliğin setter yönteminden önce yer alır.
- Yöntem, self ve özniteliğe atanan değeri temsil eden bir parametre (hour) olmak üzere iki parametre alır.
- **init**, nesnenin *hour özniteliğini oluşturmadan ve başlatmadan önce _init'in* saat argümanını doğrulamak için bu setter'ı çağırdı.
- okunabilir-yazılabilir bir özellik, hem bir getter hem de bir setter'a sahiptir.
- sadece okunabilir bir özellik, yalnızca bir getter'a sahiptir. [bağlantı metni](#)

Class Time: minute ve second Okunabilir-Yazılabilir Özellikleri

- Aşağıdaki minute ve second adlı yöntemler, okunabilir-yazılabilir minute ve second özelliklerini tanımlar.
- Her özelliğin setter'ı, ikinci argümanın 0-59 aralığında olduğunu (dakika ve saniyelerin geçerli değer aralığı) sağlar.

```

@property
def minute(self):
    """Return the minute."""
    return self._minute

@minute.setter
def minute(self, minute):
    """Set the minute."""
    if not (0 ≤ minute < 60):
        raise ValueError(f'Minute ({minute}) must be 0-59')

    self._minute = minute

@property
def second(self):
    """Return the second."""
    return self._second

@second.setter
def second(self, second):
    """Set the second."""
    if not (0 ≤ second < 60):
        raise ValueError(f'Second ({second}) must be 0-59')

    self._second = second

```

Class Time: set_time Yöntemi

- set_time yöntemi, bir tek yöntem çağırısı ile üç özneliği değiştirir.
- Yöntem, yukarıda tanımlanan sınıfın özelliklerini kullanır.

```

def set_time(self, hour=0, minute=0, second=0):
    """Set values of hour, minute, and second."""
    self.hour = hour
    self.minute = minute
    self.second = second

```

Class Time: Özel Yöntem `__repr__`

- Bir nesneyi repr yerleşik fonksiyonuna geçirdiğinizde - ki bu, bir değişkeni bir IPython oturumunda değerlendirdiğinizde örtük olarak gerçekleşir - ilgili sınıfın **repr** özel yöntemi çağırılır ve nesnenin bir dize temsili alınır.

```

def __repr__(self):
    """Return Time string for repr()."""
    return (f'Time(hour={self.hour}, minute={self.minute}, ' +
            f'second={self.second})')

```

Class Time: Özel Yöntem `__repr__` (cont.)

- Python belgeleri, **repr**'in nesnenin "resmi" dize temsili döndürdüğünü belirtir.
- Oluşturur ve nesneyi başlatan bir yapıcı ifadesi gibi görünmelidir. (Should look like a constructor expression that creates and initializes the object)

▼ Class Time: Özel Yöntem `__str__`

- **str** özel yöntemi, aşağıdaki durumlarda örtük olarak çağırılır:
 - Bir nesneyi yerleşik str fonksiyonuyla bir dizeye dönüştürdüğünüzde
 - Bir nesneyi print komutuyla yazdırdığınızda

```
def __str__(self):
    """Print Time in 12-hour clock format."""
    return (('12' if self.hour in (0, 12) else str(self.hour % 12)) +
            f':{self.minute:0>2}:{self.second:0>2}' +
            (' AM' if self.hour < 12 else ' PM'))
```

10.4.3 Class `Time` Tanımı Tasarım Notları

Sınıfın Arayüzü (Interface of a Class)

- Sınıf `Time`'in özellikleri ve yöntemleri, sınıfın **public interface** tanımlar—
 - Bu, programcıların sınıfın nesneleriyle etkileşimde bulunmak için kullanması gereken özellikler ve yöntemlerdir.

Özellikler Her Zaman Erişilebilirdir

- Python, veri özelliklerini (`_hour`, `_minute` ve `_second`) doğrudan değiştirmenizi engellemez. Yani, Python doğrudan bu özelliklere erişerek değerlerini değiştirmenize izin verir.

```
1 # from timewithproperties import Time
```

```
1 wake_up = Time(hour=7, minute=45, second=30)
2 wake_up._hour
3
```

```
7
```

```
1 wake_up._hour = 100
2 wake_up
```

```
Time(hour=100, minute=45, second=30)
```

- [4] kod örneğinden sonra, `wake_up` nesnesi geçersiz veri içeriyor çünkü sınıfta veri doğrulaması uygulanmamıştır.
- Python'ın belgelerinde "Python'da veri gizlemeyi zorunlu kılan bir mekanizma bulunmamaktadır, her şey anlaşmaya dayalıdır" şeklinde bir ifade bulunmaktadır.

Dahili Veri Temsili

- Zamanı dahili olarak gece yarısından beri geçen saniye sayısı olarak temsil edebiliriz.
 - Bu durumda `hour`, `minute` ve `second` özelliklerini yeniden uygulamamız gerekecektir.
- Diğer programcılar, bu değişikliklerden haberdar olmadan aynı arayüzü kullanabilir ve aynı sonuçları elde edebilir.

Bir Sınıfın Uygulama Detaylarını Geliştirmek

- Başka programcılara sunulmadan önce bir sınıfın arayüzünü dikkatlice düşünmek önemlidir.
- Sınıfın arayüzünü, mevcut kodların kırılmadan sınıfın uygulama detaylarını güncellemesine izin verecek şekilde tasarlamak önemlidir.

Yardımcı Metodlar

- Tüm metodlar bir sınıfın arayüzünün bir parçası olmak zorunda değildir.
- Bazıları sadece sınıf içinde kullanılan yardımcı metodlar olarak hizmet eder ve istemci kodu tarafından sınıfın genel arayüzünün bir parçası olmayı amaçlamaz.
- Bu tür metodlar tek bir alt çizgi ile başlayan bir isme sahip olmalıdır. Bu, diğer programcıların bu metodların sınıfın genel kullanımına dahil olmadığını ve içsel kullanım amaçlı olduğunu anlamasına yardımcı olur.

`datetime` Modülü

- Zaman ve tarihleri temsil etmek için kendi sınıflarınızı oluşturmak yerine, Python Standart Kütüphanesi'nin `datetime` modülünün yeteneklerini kullanabilirsiniz.

<https://docs.python.org/3/library/datetime.html>

▼ 10.5 "Özel" Nitelikleri Taklit Etmek

- Python programcıları genellikle, bir sınıfın iç işleyişinde önemli olan ancak sınıfın genel erişim arayüzünün bir parçası olmayan veri veya yardımcı yöntemler için "özel" nitelikler kullanır.
- İki başlı çizgiyle başlayan (`__hour` gibi) bir niteliğin veya yöntemin "özel" olduğunu ve sınıfın istemcilerinin bunlara erişememesi gerektiğini belirtir.
- Python, bu tür tanımlayıcıları, nitelik adını `SınıfAdı_` şeklinde değiştirerek yeniden adlandırır.
 - buna ad değiştirme (name mangling) denir.

IPython Otomatik Tamamlama Yalnızca "Genel" Nitelikleri Gösterir

- IPython, bir ifadeyi otomatik tamamlamak için Tab tuşuna bastığınızda, tek veya çift başlı çizgiyle başlayan nitelikleri göstermez.

▼ Demonstrating "Private" Attributes

```
# private.py
"""Class with public and private attributes."""

class PrivateClass:
    """Class with public and private attributes."""

    def __init__(self):
        """Initialize the public and private attributes."""
        self.public_data = "public" # public attribute
        self.__private_data = "private" # private attribute
```

- "PrivateData" sınıfından bir nesne oluşturun

```
1 class PrivateClass:
2     """Class with public and private attributes."""
3
4     def __init__(self):
5         """Initialize the public and private attributes."""
6         self.public_data = "public" # public attribute public
7         self._semi_private_data = "semi private" # semi private attribute internal
8         self.__private_data = "private" # private attribute
```

```
1 my_object = PrivateClass()
```

- `public_data` özneliğine doğrudan erişin

```
1 my_object.public_data
```

```
'public'
```

```
1 my_object._semi_private_data
```

```
'semi private'
```

- `__private_data` doğrudan erişme girişimi

```
1 my_object.__private_data
```

```
-----
AttributeError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_9484\3291977527.py in <module>
----> 1 my_object.__private_data

AttributeError: 'PrivateClass' object has no attribute '__private_data'
```

SEARCH STACK OVERFLOW

```
1 my_object._PrivateClass__private_data
```

```
'private'
```

```
1 my_object._PrivateClass__private_data = 'modified'
2 my_object._PrivateClass__private_data

'modified'
```

10.7 Kalıtım: Temel Sınıflar ve Alt Sınıflar

- Sıklıkla, bir sınıfın nesnesi aynı zamanda başka bir sınıfın nesnesidir
- CarLoan bir Loan'dır ve aynı şekilde HomeImprovementLoan'lar ve MortgageLoan'lar da birer Loan'dır
- CarLoan sınıfı, Loan sınıfından kalıtım aldığı söylenebilir.
- Bu bağlamda, Loan sınıfı bir temel sınıf olarak adlandırılırken, CarLoan sınıfı bir alt sınıftır.
- CarLoan, Loan'ın belirli bir türüdür, ancak her Loan'ın bir CarLoan olduğunu iddia etmek yanlıştır.

10.7 Kalıtım: Temel Sınıflar ve Alt Sınıflar (cont.)

- Temel sınıfların ve alt sınıfların tablosu - temel sınıflar genellikle "daha genel" ve alt sınıflar "daha spesifik" olma eğilimindedir:


Base class	Subclasses
Student	GraduateStudent, UndergraduateStudent
Shape	Circle, Triangle, Rectangle, Sphere, Cube
Loan	CarLoan, HomeImprovementLoan, MortgageLoan
Employee	Faculty, Staff
BankAccount	CheckingAccount, SavingsAccount

10.7 Kalıtım: Temel Sınıflar ve Alt Sınıflar (cont.)

- Her alt sınıf nesnesi, temel sınıfın bir nesnesidir.
- Bir temel sınıf tarafından temsil edilen nesneler kümesi, genellikle alt sınıflardan herhangi biri tarafından temsil edilen nesneler kümesinden daha büyüktür.


CommunityMember Kalıtım Hiyerarşisi

- Kalıtım ilişkileri, ağaç benzeri hiyerarşik yapılar oluşturur.
- Bir temel sınıf, alt sınıflarıyla hiyerarşik bir ilişkide bulunur.
- Tek kalıtım ile bir sınıf, bir temel sınıftan türetilir.
- Çoklu kalıtım ile bir alt sınıf, iki veya daha fazla temel sınıftan kalıtım alır.
- Üniversite topluluğu için örnek sınıf hiyerarşisi, aynı zamanda bir kalıtım hiyerarşisi olarak adlandırılır.

 Sample class hierarchy for a university community

- Hiyerarşideki her ok, bir ilişkiyi temsil eder.
 - "bir Employee, bir CommunityMember'dır"
 - "bir Teacher, bir Faculty üyesidir"
- CommunityMember, Employee, Student ve Alum'un doğrudan temel sınıfıdır ve diyagramdaki diğer sınıfların hepsinin dolaylı temel sınıfıdır. En alttan başlayarak okları takip ederek ve bir ilişkisini uygulayarak en üstteki en üst sınıfa kadar gidebilirsiniz.

Shape Kalıtım Hiyerarşisi

 Shape inheritance hierarchy

"is-a" ve "has-a" arasındaki fark şudur:

- Miras alma (inheritance) ile "is-a" ilişkisi oluşturulur. Bu durumda, bir alt sınıfın bir nesnesi aynı zamanda bir üst sınıfın bir nesnesi gibi davranabilir.
- "Has-a" ilişkisi (kompozisyon) ise bir sınıfın diğer sınıfların nesnelerine referanslar olarak sahip olduğu ilişkiyi ifade eder. Bu durumda, bir sınıf başka sınıfların nesnelerini üyeleri olarak içerir.

10.8 Bir Miras Hiyerarşisi Oluşturma; Polimorfizmi Tanıtma

- Bir şirketin maaş uygulamasında çalışan tiplerini içeren bir hiyerarşi
- Şirketin tüm çalışanları arasında birçok ortak nokta vardır
 - komisyonlu çalışanlar (bir üst sınıfın nesneleri olarak temsil edilecekler) satışlarının bir yüzdesini alır
 - sabit maaşlı komisyonlu çalışanlar (bir alt sınıfın nesneleri olarak temsil edilecekler) satışlarının bir yüzdesini artı bir sabit maaş alır

▼ 10.8.1 Temel Sınıf CommissionEmployee

Sınıf CommissionEmployee, aşağıdaki özellikleri sağlar:

- **init** metodu, `_first_name`, `_last_name` ve `_ssn` (Sosyal Güvenlik numarası) veri özelliklerini oluşturur ve `gross_sales` ve `commission_rate` özelliklerinin setter'larını kullanarak ilgili veri özelliklerini oluşturur.
- Salt okunur olan `first_name`, `last_name` ve `ssn` özellikleri, ilgili veri özelliklerini döndürür.
- Okuma ve yazma yetkisine sahip olan `gross_sales` ve `commission_rate` özellikleri, setter'larında veri doğrulaması yapar.
- `earnings` metodu, bir CommissionEmployee'nin kazançlarını hesaplar ve döndürür.
- **repr** metodu, bir CommissionEmployee'nin bir dize temsilini döndürür.

```

1
2 # commissionemployee.py
3 """CommissionEmployee base class."""
4 from decimal import Decimal
5
6 class CommissionEmployee:
7     """An employee who gets paid commission based on gross sales."""
8
9     def __init__(self, first_name, last_name, ssn,
10                  gross_sales, commission_rate):
11         """Initialize CommissionEmployee's attributes."""
12         self._first_name = first_name
13         self._last_name = last_name
14         self._ssn = ssn
15         self.gross_sales = gross_sales # validate via property
16         self.commission_rate = commission_rate # validate via property
17
18     @property
19     def first_name(self):
20         return self._first_name
21
22     @property
23     def last_name(self):
24         return self._last_name
25
26     @property
27     def ssn(self):
28         return self._ssn
29
30     @property
31     def gross_sales(self):
32         return self._gross_sales
33
34     @gross_sales.setter
35     def gross_sales(self, sales):
36         """Set gross sales or raise ValueError if invalid."""
37         if sales < Decimal('0.00'):
38             raise ValueError('Gross sales must be ≥ to 0')
39
40         self._gross_sales = sales
41
42     @property
43     def commission_rate(self):
44         return self._commission_rate
45
46     @commission_rate.setter
47     def commission_rate(self, rate):
48         """Set commission rate or raise ValueError if invalid."""
49         if not (Decimal('0.0') < rate < Decimal('1.0')):
50             raise ValueError(
51                 'Interest rate must be greater than 0 and less than 1')
52
53         self._commission_rate = rate
54
55     def earnings(self):
56         """Calculate earnings."""
57         return self.gross_sales * self.commission_rate
58
59     def __repr__(self):
60         """Return string representation for repr()."""
61         return ('CommissionEmployee: ' +

```

```

62         f'{self.first_name} {self.last_name}\n' +
63         f'social security number: {self.ssn}\n' +
64         f'gross sales: {self.gross_sales:.2f}\n' +
65         f'commission rate: {self.commission_rate:.2f}')
66
67

```

Tüm Sınıflar Dolaylı veya Dolaysız Olarak object Sınıfından Miras Alır

- Her Python sınıfı mevcut bir sınıftan miras alır.
- Yeni bir sınıf için açıkça temel sınıf belirtmediğinizde, Python sınıfın doğrudan object sınıfından miras aldığını varsayar.
- CommissionEmployee sınıfının başlığı şu şekilde yazılabilirdi:

```
class CommissionEmployee(object):
```

- CommissionEmployee sınıfının ardından gelen parantezler, mirası gösterir ve
 - tek miras için tek bir sınıfı veya
 - çoklu miras için virgülle ayrılmış bir temel sınıf listesini içerebilir.

Tüm Sınıflar Doğrudan veya Dolaylı Olarak "nesne" Sınıfından Miras Alır (cont.)

- CommissionEmployee, object sınıfının tüm yöntemlerini miras alır.
- object sınıfından miras alınan birçok yöntemden ikisi **repr** ve **str** yöntemleridir.
 - Bu nedenle, her sınıfın, üzerlerinde çağrıldıklarında nesnelerin dize temsillerini döndüren bu yöntemlere sahip olduğu söylenebilir.
- Türetilmiş bir sınıfta temel sınıfın yöntem uygulaması uygun değilse, bu yöntem uygun bir uygulama ile türetilmiş sınıfta geçersiz kılınabilir (yani yeniden tanımlanabilir).
 - **repr** yöntemi, object sınıfındaki varsayılan uygulamayı geçersiz kılar.

▼ Test Sınıfı CommissionEmployee

- CommissionEmployee özelliklerinden bazılarını test edin

```

1 c = CommissionEmployee('Sue', 'Jones', '333-33-3333',
2     Decimal('10000.00'), Decimal('0.06'))

```

```
1 c
```

```

CommissionEmployee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00
commission rate: 0.06

```

- calculate and display the CommissionEmployee's earnings

```
1 print(f'{c.earnings():.2f}')
```

```
600.00
```

```
1 c.last_name="deneme"
```

```

-----
AttributeError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_9484\2448427201.py in <module>
----> 1 c.last_name="deneme"

AttributeError: can't set attribute

```

SEARCH STACK OVERFLOW

- change the CommissionEmployee's gross sales and commission rate, then recalculate the earnings

```
1 c.gross_sales = Decimal('20000.00')
```

```
1 c.commission_rate = Decimal('0.1')
```



```
1 print(f'{c.earnings():.2f}')
```

```
2,000.00
```

10.8.2 Alt Sınıf SalariedCommissionEmployee

- Tek miras ile, alt sınıf temel sınıf ile aynı şekilde başlar
- Mirasın gerçek gücü, alt sınıfta temel sınıftan miras alınan özellikler için eklemeler, değişiklikler veya iyileştirmeler yapabilme yeteneğinden gelir.
- SalariedCommissionEmployee'nin birçok yeteneği, CommissionEmployee sınıfıninkilere benzer veya aynıdır
 - Her iki tür çalışanın da ad, soyad, T.C. kimlik numarası, brüt satış ve komisyon oranı veri öznitelikleri ve bu verileri manipüle etmek için özellikleri ve yöntemleri vardır
- Miras, kodu tekrarlamadan bir sınıfın özelliklerini "emsal" edebilmemizi sağlar

SalariedCommissionEmployee Sınıfının Tanımlanması

- Alt sınıf SalariedCommissionEmployee, çoğu yeteneğini CommissionEmployee sınıfından miras alır
- SalariedCommissionEmployee, bir CommissionEmployee'dir (çünkü miras, CommissionEmployee sınıfının yeteneklerini aktarır)
- SalariedCommissionEmployee sınıfı ayrıca aşağıdaki özelliklere sahiptir:
 - **init** yöntemi, CommissionEmployee sınıfından miras alınan tüm verileri başlatır, ardından base_salary özelliğinin setter'ını kullanarak _base_salary veri özniteliğini oluşturur
 - Okunabilir-yazılabilir base_salary özelliği, setter'ın veri doğrulaması yapmasını sağlar.
 - Özelleştirilmiş bir earnings yöntemi
 - Özelleştirilmiş bir **repr** yöntemi

```
1
```

```
1
2 # salariedcommissionemployee.py
3 """SalariedCommissionEmployee derived from CommissionEmployee."""
4
5 class SalariedCommissionEmployee(CommissionEmployee):
6     """An employee who gets paid a salary plus
7     commission based on gross sales."""
8
9     def __init__(self, first_name, last_name, ssn,
10                 gross_sales, commission_rate, base_salary):
11         """Initialize SalariedCommissionEmployee's attributes."""
12         super().__init__(first_name, last_name, ssn,
13                         gross_sales, commission_rate)
14         self.base_salary = base_salary # validate via property
15
16 @property
17 def base_salary(self):
18     return self._base_salary
19
20 @base_salary.setter
21 def base_salary(self, salary):
22     """Set base salary or raise ValueError if invalid."""
23     if salary < Decimal('0.00'):
24         raise ValueError('Base salary must be ≥ to 0')
25
26     self._base_salary = salary
27
28 def earnings(self):
29     """Calculate earnings."""
30     return super().earnings() + self.base_salary
31
32 def __repr__(self):
33     """Return string representation for repr()."""
34     return ('Salaried' + super().__repr__() +
35           f'\nbase salary: {self.base_salary:.2f}')
```

CommissionEmployee Sınıfından Miras Almak

```
class SalariedCommissionEmployee(CommissionEmployee):
```

- SalariedCommissionEmployee sınıfının CommissionEmployee sınıfından miras aldığını belirtir

- SalariedCommissionEmployee sınıfında CommissionEmployee sınıfının veri özniteliklerini, özelliklerini ve yöntemlerini görmüyorsunuz, ancak onlar mevcuttur

Method **init** ve Yerleşik Fonksiyon super

- Her alt sınıfın **_init** yöntemi, miras alınan veri özniteliklerini başlatmak için açıkça üst sınıfın **init** yöntemini çağırmalıdır
 - Bu çağrı, alt sınıfın **init** yöntemindeki ilk ifade olmalıdır
- **super().init** gösterimi, yerleşik super fonksiyonunu kullanarak üst sınıfın **init** yöntemini bulup çağırarak için kullanılır

Method earnings'i Geçersiz Kılma

- Sınıf SalariedCommissionEmployee'nin earnings yöntemi, bir SalariedCommissionEmployee'nin kazancını hesaplamak için sınıf CommissionEmployee'nin earnings yöntemini geçersiz kılar
 - Kazancın sadece komisyon kısmını elde etmek için **super().earnings()** ifadesi ile CommissionEmployee'nin earnings yöntemini çağırır

Method **repr**'i Geçersiz Kılma

- SalariedCommissionEmployee'nin **repr** yöntemi, bir SalariedCommissionEmployee için uygun olan bir String temsilini döndürmek için sınıf CommissionEmployee'nin **repr** yöntemini geçersiz kılar
- **super().repr()** ifadesi, CommissionEmployee'nin **repr** yöntemini çağırır

▼ Test Sınıfı SalariedCommissionEmployee

```
1 s = SalariedCommissionEmployee('Bob', 'Lewis', '444-44-4444',
2     Decimal('5000.00'), Decimal('0.04'), Decimal('300.00'))
```

```
1 print(s.first_name, s.last_name, s.ssn, s.gross_sales,
2     s.commission_rate, s.base_salary)
```

```
Bob Lewis 444-44-4444 5000.00 0.04 300.00
```

- SalariedCommissionEmployee nesnesi, CommissionEmployee ve SalariedCommissionEmployee sınıflarının tüm özelliklerine sahiptir.
- SalariedCommissionEmployee'nin kazançlarını hesaplayıp görüntüleyebilirsiniz.

```
1 print(f'{s.earnings():,.2f}')
```

```
500.00
```

- Modify the **gross_sales**, **commission_rate** and **base_salary** properties, then display the updated data via the SalariedCommissionEmployee's **__repr__** method

```
1 s.gross_sales = Decimal('10000.00')
```

```
1 s.commission_rate = Decimal('0.05')
```

```
1 s.base_salary = Decimal('1000.00')
```

```
1 print(s)
```

```
SalariedCommissionEmployee: Bob Lewis
social security number: 444-44-4444
gross sales: 10000.00
commission rate: 0.05
base salary: 1000.00
```

- SalariedCommissionEmployee'nin güncellenmiş kazançlarını hesaplayın ve görüntüleyin

```
1 print(f'{s.earnings():,.2f}')
```

```
1,500.00
```

▼ "is a" İlişisini Test Etme

`issubclass` ve `isinstance` işlevleri, "is a" ilişkilerini test etmek için kullanılır

- "issubclass", bir sınıfın diğerinden türetilip türetilmediğini belirler

```
1 issubclass(SalariedCommissionEmployee, CommissionEmployee)
```

```
True
```

- 'isinstance' fonksiyonu, bir nesnenin belirli bir türle "is a" ilişkisi olup olmadığını kontrol etmek için kullanılır.

```
1 isinstance(s, CommissionEmployee)
```

```
True
```

```
1 isinstance(c, SalariedCommissionEmployee)
```

```
False
```

```
1 isinstance(s, SalariedCommissionEmployee)
```

```
True
```

10.8.3 Polimorfik Olarak CommissionEmployee ve SalariedCommissionEmployee İşleme

- Miras ile, bir alt sınıfın her nesnesi aynı zamanda o alt sınıfın üst sınıfının bir nesnesi olarak da işlenebilir.
- Bu ilişkiden yararlanarak, miras yoluyla ilişkili nesneleri bir liste içine yerleştirebilir ve ardından listenin üzerinde döngü oluşturabilir ve her öğeyi bir üst sınıf nesnesi olarak işleyebilirsiniz.
 - Bu, çeşitli nesnelerin genel bir şekilde işlenmesine olanak sağlar.

```
1 employees = [c, s]
```

```
1 for employee in employees:
2     print(employee)
3     print(f'{employee.earnings():.2f}\n')
```

```
CommissionEmployee: Sue Jones
social security number: 333-33-3333
gross sales: 20000.00
commission rate: 0.10
2,000.00
```

```
SalariedCommissionEmployee: Bob Lewis
social security number: 444-44-4444
gross sales: 10000.00
commission rate: 0.05
base salary: 1000.00
1,500.00
```

- Her çalışan için doğru string temsili ve kazançlar gösterilir.
- Bu, nesne tabanlı programlamanın (OOP) temel bir yeteneği olan polimorfizm olarak adlandırılır.

10.9 Duck Typing and Polymorphism

- Çoğu nesne tabanlı programlama dili, polimorfik davranışı elde etmek için miras tabanlı "is a" ilişkilerini gerektirir.
- Python, aynı zamanda Python belgelerinde tanımlanan ördek yazmayı da destekler:

Bir nesnenin doğru arayüze sahip olup olmadığını belirlemek için nesnenin türüne bakmayan bir programlama stili; bunun yerine, yöntem veya özelliği sadece çağrılır veya kullanılır ("Eğer bir ördek gibi görünüyorsa ve ördek gibi ötüyorsa, o zaman bir ördek olmalıdır").

- Bir nesneyi çalışma zamanında işlerken, nesnenin türü önemli değildir.
- Nesne, erişmek istediğiniz veri özelliği, özellik veya yöntemi (uygun parametrelerle) içeriyorsa, kod çalışacaktır.

10.9 Duck Typing and Polymorphism (cont.)

- Reconsider the loop at the end of Section 10.8.3

```
for employee in employees:
    print(employee)
    print(f'{employee.earnings():.2f}\n')
```

- Çalışma doğru şekilde gerçekleşir, yeter ki employees sadece aşağıdaki özelliklere sahip nesneler içersin:
 - print ile görüntülenebilir (yani bir dize temsiliyeti vardır)
 - argumentsız çağrılabilen bir earnings yöntemine sahiptir

10.9 Duck Typing and Polymorphism (cont.)

- Tüm sınıflar doğrudan veya dolaylı olarak object'den miras alır, bu yüzden hepsi print'in görüntüleyebileceği dize temsilleri için varsayılan yöntemleri miras alır
- Bir sınıfın, argümentsız çağrılabilen bir earnings yöntemi varsa, nesnenin sınıfı CommissionEmployee ile "is a" ilişkisi olsa da olmasa da, o sınıfın nesnelerini employees listesine dahil edebiliriz
- WellPaidDuck sınıfını düşünelim:

```
1 class WellPaidDuck:
2     def __repr__(self):
3         return 'I am a well-paid duck'
4     def earnings(self):
5         return Decimal('1_000_000.00')
```

- Açıkça çalışanlar olmadığı belli
- Ancak, önceki döngü ile çalışacaktır

```
1 c = CommissionEmployee('Sue', 'Jones', '333-33-3333',
2                         Decimal('10000.00'), Decimal('0.06'))
```

```
1 s = SalariedCommissionEmployee('Bob', 'Lewis', '444-44-4444',
2                                Decimal('5000.00'), Decimal('0.04'), Decimal('300.00'))
```

```
1 d = WellPaidDuck()
```

```
1 employees = [c, s, d]
```

*Listedeki üç nesneyi de *polymorphically* işlemek için Duck typing kullanın

```
1 for employee in employees:
2     print(employee)
3     print(f'{employee.earnings():.2f}\n')
```

```
CommissionEmployee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00
commission rate: 0.06
600.00
```

```
SalariedCommissionEmployee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
500.00
```

```
I am a well-paid duck
1,000,000.00
```

10.10 Operator Overloading

- Python operatörlerinin kendi türünüzdeki nesneleri nasıl işlemesi gerektiğini tanımlamak için **operator overloading** kullanabilir
- çoğu operator overload edilebilir
- For every overloadable operator, class `object` defines a special method
 - e.g., `__add__` for addition (+) or `__mul__` for multiplication (*)
- Bu yöntemlerin Override edilmesi, belirli bir işlecin özel sınıfınızın nesneleri için nasıl çalıştığını tanımlamanıza olanak tanır
- Özel yöntemlerin tam listesi

<https://docs.python.org/3/reference/datamodel.html#special-method-names>

```
1 l=[]
2 dir(l)

['_add_',
 '_class_',
 '_class_getitem_',
 '_contains_',
 '_delattr_',
 '_delitem_',
 '_dir_',
 '_doc_',
 '_eq_',
 '_format_',
 '_ge_',
 '_getattribute_',
 '_getitem_',
 '_gt_',
 '_hash_',
 '_iadd_',
 '_imul_',
 '_init_',
 '_init_subclass_',
 '_iter_',
 '_le_',
 '_len_',
 '_lt_',
 '_mul_',
 '_ne_',
 '_new_',
 '_reduce_',
 '_reduce_ex_',
 '_repr_',
 '_reversed_',
 '_rmul_',
 '_setattr_',
 '_setitem_',
 '_sizeof_',
 '_str_',
 '_subclasshook_',
 'append',
 'clear',
 'copy',
 'count',
 'extend',
 'index',
 'insert',
 'pop',
 'remove',
 'reverse',
 'sort']
```

```
1 4+5 # toplama işlemi yapar
2 l1=[1,2]
3 l2=[3,1]
4 l1 + l2 # 2 listeyi concate yapacak şekilde overload yapılmıştır
5 "murtaza"+" hocamm"
```

'murtaza hocamm'

- object.__lt__(self, other)
- object.__le__(self, other)
- object.__eq__(self, other)
- object.__ne__(self, other)
- object.__gt__(self, other)
- object.__ge__(self, other)
- These are the so-called “rich comparison” methods.
- The correspondence between operator symbols and method names is as follows:
- $x < y$ calls `x.__lt__(y)`, $x \leq y$ calls `x.__le__(y)`, $x = y$ calls `x.__eq__(y)`,
- $x \neq y$ calls `x.__ne__(y)`, $x > y$ calls `x.__gt__(y)`, and $x \geq y$ calls `x.__ge__(y)`.

Operator Overloading kısıtlamaları

- öncelik overloading ile değiştirilemez
- Bir operatörün soldan sağa veya sağdan sola gruplaması değiştirilemez
- İster tekli ister ikili olsun, bir operatörün “Arity” değeri değiştirilemez
- yeni operatörler oluşturamaz

- Bir operatörün yerleşik türdeki nesneler üzerinde nasıl çalıştığı değiştirilemez
- Yalnızca özel sınıftaki nesnelerle veya özel sınıftaki bir nesne ile yerleşik türdeki bir nesnenin karışımıyla çalışır

Complex Numbers

- We'll define a class named `Complex` that represents complex numbers
- Complex numbers, like $-3 + 4i$ and $6.2 - 11.73i$, have the form

```
realPart '+' imaginaryPart '* i'
```

- `i` is the square root of -1
- We'll overload `+` and `+=`

10.10.1 Test-Driving Class `Complex`

```
1 try:
2     import complex
3     print("complex import edildi")
4 except:
5     s=input("Complex indirilsin mi e/h")
6     if "e"==s.lower():
7         print("!pip install Complex")
8         !pip install complex
9
10
11 from complex import Complex
```

```
complex import edildi
```

```
1 y=Complex(real=5,imaginary=1)
2 print(y)
3 y
```

```
5.00+1.00i
<complex.Complex at 0x1dc3c527310>
```

```
1 x=Complex(real=4,imaginary=-1)
2 print(x)
3 x
```

```
4.00-1.00i
<complex.Complex at 0x1dc3c527940>
```

```
1 x + y
```

```
<complex.Complex at 0x1dc3c527130>
```

```
1 print(x+y)
2 print(x)
3 print(y)
4
```

```
9.00+0.00i
4.00-1.00i
5.00+1.00i
```

- `x` ve `y` `Complex` nesnelerini `+` operatörü ile toplamak için `add` metodunu tanımlayabilirsiniz
- İki işlenenin gerçek kısımlarını ve iki işlenenin hayali kısımlarını toplar, ardından yeni bir "Karmaşık" nesnesi döndürür
- `+` does not modify either of its operands

```
1 print(x)
```

```
4.00-1.00i
```

- Use `+=` to add `y` to `x` and store the result in `x`
- `+=` operator *modifies* its left operand

```
1 x += y
```

```
1 print(x)
```

```
9.00+0.00i
```

```
1 dir(x)
```

```
[ '__add__',
  '__class__',
  '__delattr__',
  '__dict__',
  '__dir__',
  '__doc__',
  '__eq__',
  '__format__',
  '__ge__',
  '__getattr__',
  '__gt__',
  '__hash__',
  '__init__',
  '__init_subclass__',
  '__le__',
  '__lt__',
  '__module__',
  '__mul__',
  '__ne__',
  '__new__',
  '__reduce__',
  '__reduce_ex__',
  '__repr__',
  '__setattr__',
  '__sizeof__',
  '__str__',
  '__sub__',
  '__subclasshook__',
  '__truediv__',
  '__weakref__',
  'angle',
  'conjugate',
  'imaginary',
  'log',
  'mod',
  'real']
```

▼ Overloaded + Operator

- Python'da + operatörünü aşırı yüklemek için özel olarak tanımlanmış `__add__` metodunu kullanırsınız. Bu metod, + operatörünün nasıl davranacağını belirlemek için özelleştirilmiştir

```
def __add__(self, right):
    """Overrides the + operator."""
    return Complex(self.real + right.real,
                    self.imaginary + right.imaginary)
```

- İkili operatörleri aşırı yükleyen yöntemler iki parametre sağlamalıdır
 - İlk parametre (self), sol operandı temsil eder.
 - İkinci parametre (right), sağ operandı temsil eder.
- Bu metodlar, iki operandı alarak istenilen işlemi gerçekleştirir ve sonucu döndürür.
 - Orijinal operandların içeriğini değiştirmedikimize dikkat etmek önemlidir.

▼ Overloaded += Augmented Assignment

- Override special method `__iadd__` to define how `+=` adds two `Complex` objects

```
def __iadd__(self, right):
    """Overrides the += operator."""
    self.real += right.real
    self.imaginary += right.imaginary
    return self
```

10.10.2 Class `Complex` Definition

Method `__init__`

```
# complexnumber.py
"""Complex class with overloaded operators."""

class Complex:
    """Complex class that represents a complex number
    with real and imaginary parts."""

    def __init__(self, real, imaginary):
        """Initialize Complex class's attributes."""
        self.real = real
        self.imaginary = imaginary
```

Overloaded `+` Operator

```
def __add__(self, right):
    """Overrides the + operator."""
    return Complex(self.real + right.real,
                   self.imaginary + right.imaginary)
```

Overloaded `+=` Augmented Assignment

```
def __iadd__(self, right):
    """Overrides the += operator."""
    self.real += right.real
    self.imaginary += right.imaginary
    return self
```

Method `__repr__`

```
def __repr__(self):
    """Return string representation for repr()."""
    return (f'({self.real} ' +
            ('+' if self.imaginary ≥ 0 else '-') +
            f' {abs(self.imaginary)}i)')
```

```
1 class Complex:
2
3     # Constructor
4     def __init__(self, real, imag):
5         self.real = real
6         self.imag = imag
7
8     # For call to repr(). Prints object's information
9     def __repr__(self):
10         return 'Rational(%, %s)' % (self.real, self.imag)
11
12     # For call to str(). Prints readable form
13     def __str__(self):
14         return '%s + i%s' % (self.real, self.imag)
15
16
17     def __add__(self, right):
18         return str(Complex(self.real + right.real, self.imag + right.imag))
19
20     def __iadd__(self, right):
21         self.real += right.real
```



```

22     self.imag += right.imag
23     return self
24
25

```

```

1 y=Complex(real=5,imag=1)
2 print(y)
3 y

```

```

5 + i1
Rational(5, 1)

```

```

1 def __imult__(self, right):
2     """Overrides the *= operator."""
3     self.real *= right.real
4     self.imaginary *= right.imaginary
5     return self

```

```

1 # Driver program to test above
2 t = Complex(10, 20)
3 t2= Complex(3,-7)
4 # Same as "print t"
5 print (str(t))
6 print (repr(t))
7 t+t2

```

```

10 + i20
Rational(10, 20)
'13 + i13'

```

```
1 t * t2
```

```

-----
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_9484\1726468461.py in <module>
----> 1 t * t2

```

```
TypeError: unsupported operand type(s) for *: 'Complex' and 'Complex'
```

SEARCH STACK OVERFLOW

```
1 t>t2
```

```

-----
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_9484\4056519683.py in <module>
----> 1 t>t2

```

```
TypeError: '>' not supported between instances of 'Complex' and 'Complex'
```

SEARCH STACK OVERFLOW

```
1 t-t2
```

```

-----
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_9484\4053020568.py in <module>
----> 1 t-t2

```

```
TypeError: unsupported operand type(s) for -: 'Complex' and 'Complex'
```

SEARCH STACK OVERFLOW

```

1 class Complex:
2     def __init__(self, real, imaginary):
3         self.real = real
4         self.imaginary = imaginary
5
6
7
8     # For call to repr(). Prints object's information
9     def __repr__(self):
10         return 'Rational(%s, %s)' % (self.real, self.imaginary)
11
12     # For call to str(). Prints readable form
13     def __str__(self):
14         return '%s + i%s' % (self.real, self.imaginary)
15

```

```

16
17 def __add__(self, right):
18     return str(Complex(self.real + right.real, self.imaginary + right.imaginary))
19
20 def __iadd__(self, right):
21     self.real += right.real
22     self.imaginary += right.imaginary
23     return self
24
25 #x - y
26 def __sub__(self, right):
27     # __sub__ metodu, iki karmaşık sayıyı çıkarır ve sonucu yeni bir karmaşık sayı olarak döndürür
28     return Complex(self.real - right.real, self.imaginary - right.imaginary)
29
30 #x -= y
31 def __isub__(self, right):
32     # __isub__ metodu, sağ operand olan bir karmaşık sayıyı sol operand olan karmaşık sayıdan çıkarır ve sonucu sol op
33     self.real -= right.real
34     if(right.imaginary<0):
35         right.imaginary=abs(right.imaginary)
36     self.imaginary -= right.imaginary
37     return self
38
39 #x * y
40 def __mul__(self, right):
41     # __mul__ metodu, iki karmaşık sayının çarpımını hesaplar ve sonucu yeni bir karmaşık sayı olarak döndürür
42     return Complex((self.real * right.real) - (self.imaginary * right.imaginary),
43                    (self.imaginary * right.real) + (self.real * right.imaginary))
44
45 #x *= y
46 def __imul__(self, right):
47     new_real = (self.real * right.real) - (self.imaginary * right.imaginary)
48     new_imaginary = (self.imaginary * right.real) + (self.real * right.imaginary)
49     # değerler nesnenin değerleriyle değiştirilir ve nesne döndürülür
50     self.real = new_real
51     self.imaginary = new_imaginary
52     return self
52

```

```

1 # Driver program to test above
2 t = Complex(10, 20)
3 t2= Complex(3,-7)
4 # Same as "print t"
5 print (str(t))
6 print (repr(t))
7 print(t-t2)
8 print(t * t2)

```

```

10 + i20
Rational(10, 20)
7 + i27
170 + i-10

```

1

1

1

10.11 İstisna Sınıf Hiyerarşisi ve Özel İstisnalar

- Her istisna, Python'un istisna sınıf hiyerarşisinde yer alan bir sınıfın nesnesidir veya bu sınıflardan birine miras alan bir sınıfın nesnesidir.
- İstisna sınıfları, doğrudan veya dolaylı olarak temel sınıf `BaseException` 'dan miras alır ve `exceptions` modülünde tanımlanır.
- Dört temel `BaseException` alt sınıfı vardır:
 - `SystemExit`, program yürütmesini (veya etkileşimli bir oturumu) sonlandırır ve diğer istisna türleri gibi takip edilmediğinde bir izleme çıktısı üretmez.
 - `KeyboardInterrupt`, kullanıcı Ctrl + C (veya control + C) kesme komutunu yazdığında ortaya çıkar ve birçok sistemde hareketi durdurur.
 - `GeneratorExit`, bir üreteç kapatıldığında, normalde değerler üretmeyi bıraktığında veya `close` yöntemi açık bir şekilde çağrıldığında ortaya çıkar.
 - `Exception`, karşılaşılabileceğiniz çoğu yaygın istisna için temel sınıftır.

Temel Sınıf İstisnalarını Yakalama

- Bir `except` işleyicisi, belirli bir türdeki istisnaları yakalayabilir veya temel sınıf türünü kullanarak bu temel sınıf (base-class) istisnalarını ve tüm ilgili alt sınıf istisnalarını yakalayabilir.

Custom Exception Classes

- When you raise an exception from your code, you should generally use one of the existing exception classes from the Python Standard Library
- Can create your own custom exception classes that derive directly or indirectly from class `Exception`
- Before creating custom exception classes, look for an appropriate existing exception class in the Python exception hierarchy

Özel İstisna Sınıfları

- Kodunuzdan bir istisna yükselttiğinizde, genellikle Python Standart Kütüphanesindeki mevcut istisna sınıflarından birini kullanmalısınız.
- Kendi özel istisna sınıflarınızı, doğrudan veya dolaylı olarak `Exception` sınıfından türeyen sınıflar olarak oluşturabilirsiniz.
- Özel istisna sınıfları oluşturmadan önce, Python istisna hiyerarşisinde uygun bir mevcut istisna sınıfı arayın.

10.12 Named Tuples

- Tuples to aggregate several data attributes into a single object
- `collections` module provides `named tuples`
 - Enable you to reference a tuple's members by name rather than by index number
- Create a simple named tuple that might be used to represent a card in a deck of cards:

10.12 İsimlendirilmiş Demetler

- Birden çok veri niteliğini tek bir nesne olarak birleştirmek için tuple kullanılır.
- `collections` modülü , `named tuples` sağlar.
 - tuple elemanlarına indeks numarası yerine isim kullanmanızı sağlar.
- Bir kart destesi içindeki bir kartı temsil etmek için kullanılabilecek basit bir isimlendirilmiş tuple oluşturma:

```
1
```

```
1 from collections import namedtuple
```

- `namedtuple` fonksiyonu, yerleşik tuple türünün bir alt sınıfını oluşturur.
- İlk argüman, yeni **tipinizin adıdır**.
- İkinci argüman, yeni tipin elemanlarını belirlemek için kullanılan tanımlayıcıları temsil eden bir liste şeklindedir.

```
1 Card = namedtuple('Card', ['face', 'suit'])
```

- Now have a new tuple type named `Card`
- Create a `Card` object, access its members by name and display its string representation:

```
1 c1= Card(face='Ace', suit='Spades')
2 c2= Card(face='Ace', suit='Spades')
3 c3= Card(face='face', suit='suit')
4 c4= Card(face='face', suit='suit')
5
```

```
1 c1.face
```

```
'Ace'
```

```
1 c1.suit
```

```
'Spades'
```

```
1 type(c1)
```

```
__main__.Card
```

```
1 c1
```

```
Card(face='Ace', suit='Spades')
```

▼ Her adlandırılmış demet türü, ek yöntemlere sahiptir.

- `_make` sınıf yöntemi, bir değerler dizisi alır ve adlandırılmış demet türünden bir nesne döndürür.

```
1 values = ['Queen', 'Hearts']
```

```
1 c1 = Card._make(values)
```

```
1 c1
```

```
Card(face='Queen', suit='Hearts')
```

- bir CSV dosyasındaki kayıtları temsil eden bir adlandırılmış demet türünüz varsa faydalı olabilir
 - CSV kayıtlarını okurken ve parçalarken, onları adlandırılmış demet nesnelere dönüştürebilirsiniz
- Ayrıca, bir adlandırılmış demet nesnesinin üye isimlerini ve değerlerini içeren bir `OrderedDict` (Sıralanmış Sözlük) sözlük temsili alabilirsiniz.
 - Bu sözlük, key-değer çiftlerinin sözlüğe eklendiği sırayı hatırlar.

```
1 c1._asdict()
```

```
{'face': 'Queen', 'suit': 'Hearts'}
```

- Additional named tuple features:

<https://docs.python.org/3/library/collections.html#collections.namedtuple>

```
1 c1 == c2
```

```
False
```

```
1 c1 != c3
```

```
True
```

```
1 c3==c4
```

```
True
```

```
1 dir(c1)
```

```
[ '_add_',
  '_class_',
  '_class_getitem_',
  '_contains_',
  '_delattr_',
  '_dir_',
  '_doc_',
  '_eq_',
  '_format_',
  '_ge_',
  '_getattribute_',
  '_getitem_',
  '_getnewargs_',
  '_gt_',
  '_hash_',
  '_init_',
  '_init_subclass_',
  '_iter_',
  '_le_',
  '_len_',
  '_lt_',
  '_module_',
  '_mul_',
  '_ne_',
  '_new_',
  '_reduce_',
  '_reduce_ex_',
  '_repr_',
  '_rmul_',
  '_setattr_',
  '_sizeof_',
  '_slots_',
  '_str_',
  '_subclasshook_',
```

```
['_asdict',
 '_field_defaults',
 '_fields',
 '_make',
 '_replace',
 'count',
 'face',
 'index',
 'suit']
```

```
1 from collections import namedtuple
2 Time = namedtuple('Time', ['hour', 'minute', 'second'])
3 t = Time(13, 30, 40)
4 print(t.hour, t.minute, t.second)
5 t
6 #Time(hour=13, minute=30, second=45)
```

```
13 30 40
Time(hour=13, minute=30, second=40)
```

```
1 Account = namedtuple('Account', ['type', 'balance'], defaults=[0])
2 Account._field_defaults
3 #{'balance': 0}
4 Account('premium')
```

```
Account(type='premium', balance=0)
```

10.13 Python 3.7'nin Yeni Veri Sınıflarına Kısa Bir Giriş

- İsmlendirilmiş demetler üyelerine isimle erişme imkanı sağlasa da, hala sadece demetlerdir ve sınıflar değildir.
- İsmlendirilmiş demetlerin sağladığı bazı faydaların yanı sıra, geleneksel Python sınıflarının sağladığı yetenekleri de içeren Python 3.7'nin yeni veri sınıflarını Python Standart Kütüphanesi'ndeki `dataclasses` modülü ile kullanabilirsiniz.
- Daha özlü bir sözdizimi kullanarak ve çoğu sınıfta yaygın olan "boilerplate" kodunu otomatik olarak oluşturarak sınıfları daha hızlı oluşturmanıza yardımcı olurlar.

Veri Sınıfları Otomatik Kod Oluşturur

- Çoğu sınıf, bir nesnenin özniteliklerini oluşturmak ve başlatmak için bir `init` yöntemi ve bir nesnenin özel dize temsili için bir `repr` yöntemi sağlar.
- Veri sınıfları, veri özniteliklerini ve `init` ve `repr` yöntemlerini sizin için otomatik olarak oluştururlar.
- Veri sınıfları ayrıca bir alan adları listesinden (örneğin CSV dosyasındaki sütun adları gibi) dinamik olarak oluşturulabilir.
- Veri sınıfları, `=` operatörünü aşırı yükleyen `eq` yöntemini otomatik olarak oluştururlar.
- Herhangi bir sınıfın bir `eq` yöntemi varsa, aynı zamanda `!=` operatörünü de destekler.
- Veri sınıfları otomatik olarak `<`, `<=`, `>` ve `>=` karşılaştırma operatörleri için yöntemleri oluşturmazlar, ancak bunları destekleyebilirler.

10.13.1 Bir Card Veri Sınıfı Oluşturma

`dataclasses` ve `typing` Modüllerinden İçerme

- `dataclasses` modülü, veri sınıflarını uygulamak için dekoratörler ve fonksiyonlar tanımlar.
- `@dataclass` dekoratörü, yeni bir sınıfın bir veri sınıfı olduğunu belirtir ve çeşitli kodların sizin için yazılmasını sağlar.
- `typing` modülünden `ClassVar` ve `List`, `FACES` ve `SUITS`'in listelere başvuran sınıf değişkenleri olduğunu belirtecektir.

```
# carddataclass.py
"""Card data class with class attributes, data attributes,
autogenerated methods and explicitly defined methods."""
from dataclasses import dataclass
from typing import ClassVar, List
```

`@dataclass` Dekoratörünü Kullanma

- Bir sınıfın bir veri sınıfı olduğunu belirtmek için tanımından önce `@dataclass` dekoratörünü kullanın:

```
@dataclass
class Card:
```

- `@dataclass(order=True)` dekoratörü, veri sınıfının `<`, `<=`, `>`, `>=` için aşırı yüklenmiş karşılaştırma operatörü yöntemlerini otomatik olarak oluşturmasını sağlar
 - Veri sınıfı nesnelerinizi karşılaştırmak veya sıralamak gerektiğinde faydalıdır

▼ Değişken İşaretleme: Sınıf Öznelikleri

- Veri sınıfları, sınıf özneliklerini ve veri özneliklerini sınıfın içine, ancak sınıfın yöntemlerinin dışına bildirir.
- Veri sınıfları, sınıf özneliklerini veri özneliklerinden ayırt etmek için ek bilgi veya ipuçları gerektirir.
 - Ayrıca, otomatik olarak oluşturulan yöntemlerin uygulama ayrıntılarını etkiler.

```
FACES: ClassVar[List[str]] = ['Ace', '2', '3', '4', '5', '6', '7',
                              '8', '9', '10', 'Jack', 'Queen', 'King']
SUITS: ClassVar[List[str]] = ['Hearts', 'Diamonds', 'Clubs', 'Spades']
```

- Değişken işaretleme : `ClassVar[List[str]]` (bazen bir tip işareti olarak adlandırılır), `FACES`'in bir sınıf özneliği (`ClassVar`) olduğunu ve stringlerden oluşan bir liste (`List[str]`) olduğunu belirtir.
- `SUITS` da bir sınıf özneliğidir ve stringlerden oluşan bir liste olarak belirtilir.
- Sınıf değişkenleri tanımlamada başlatılır ve sınıfa özgüdür, sınıfın bireysel nesnelere değil.
- Bununla birlikte, **init**, **repr** ve **eq** yöntemleri sadece sınıfın nesneleriyle kullanılır.
 - Bir veri sınıfı bu yöntemleri oluşturduğunda, tüm değişken işaretleme yöntemlerini inceleyerek, yalnızca veri özneliklerini yöntem uygulamalarına dahil eder.

▼ Variable Annotations: Data Attributes

- Normalde, bir nesnenin veri özneliklerini sınıfın **init** yöntemi içinde oluştururuz.
- Sınıfın içine veri özneliği adlarını koymak, `NameError` hatası üretir, örneğin:

```
1 from dataclasses import dataclass
```

```
1 @dataclass(eq=False) #default False gelir
2 class Demo:
3     x:int # attempting to create a data attribute x
4 D1=Demo(10)
5 D2=Demo(20)
6 print(D1 == D2)
```

False

```
1 @dataclass(eq=False) #default True gelir
2 class Demo:
3     x:int # attempting to create a data attribute x
4 D1=Demo(10)
5 D2=Demo(10)
6 print(D1 == D2) # doğru olduğu halde hata vermez onun yerine sonuç ne olursa olsun varsayılan olarak False döner
```

False

```
1 @dataclass(eq=True) #default True gelir
2 class Demo:
3     x:int # attempting to create a data attribute x
4 D1=Demo(10)
5 D2=Demo(10)
6 print(D1 == D2) # doğru olduğu halde hata vermez onun yerine sonuç ne olursa olsun varsayılan olarak False döner
```

True

▼ `@dataclass(init=True, repr=True, eq=True, order=False, unsafe_hash=False, frozen=False, match_args=True, kw_only=False, slots=False, weakref_slot=False)`

```
1 * Like class attributes, each data attribute must be declared with a variable annotation
2 * The variable annotation `": str"` indicates `face` and `suit` should refer to string objects
```

```
File "C:\Users\memati\AppData\Local\Temp\ipykernel_9484\928300985.py", line 1
* Like class attributes, each data attribute must be declared with a variable annotation
^
SyntaxError: invalid syntax
```

```
1 ### Defining a Property and Other Methods
2 * Data classes are classes, so they may contain properties and methods and participate in class hierarchies
```

```
File "C:\Users\memati\AppData\Local\Temp\ipykernel_9484\2805280249.py", line 2
* Data classes are classes, so they may contain properties and methods and participate in class hierarchies
^
SyntaxError: invalid syntax
```

SEARCH STACK OVERFLOW

```
@property
def image_name(self):
    """Return the Card's image file name."""
    return str(self).replace(' ', '_') + '.png'

def __str__(self):
    """Return string representation for str()."""
    return f'{self.face} of {self.suit}'

def __format__(self, format):
    """Return formatted string representation."""
    return f'{str(self):{format}}'
```

Değişken Bildirim Notları

- Tür bildirimleri olsa bile Python hâlâ dinamik tipli bir dil olarak kalır.
- Bu nedenle, tür bildirimleri çalışma zamanında zorunlu kılınmaz.

10.13.2 Using the Card Data Class

- Create a Card`:

10.13.3 Veri Sınıfının Adlandırılmış Demetlere Göre Avantajları

- Farklı adlandırılmış demet türlerine ait nesneler, aynı sayıda üyeye ve bu üyelerin aynı değerlerine sahipse eşit olarak kabul edilebilir.
 - Farklı veri sınıfı nesnelerini karşılaştırmak her zaman False döndürür, aynı şekilde bir veri sınıfı nesnesini bir demet nesnesiyle karşılaştırmak da False döndürür.
- Bir demeti açan kodunuz varsa, bu demete daha fazla üye eklemek, açma kodunu bozar.
 - Veri sınıfı nesneleri açamaz.
 - Varolan kodu bozmadan bir veri sınıfına daha fazla veri özelliği ekleyebilirsiniz.
- Veri sınıfı, bir kalıtım hiyerarşisinde bir temel sınıf veya alt sınıf olabilir.

▼ 10.13.4 Veri Sınıflarının Geleneksel Sınıflara Göre Avantajları

- Bir veri sınıfı, **init**, **repr** ve **eq** metodlarını otomatik olarak oluşturarak size zaman kazandırır.
- Bir veri sınıfı, **<**, **<=**, **>**, **>=** karşılaştırma operatörlerini aşırı yükleyen özel metotları otomatik olarak oluşturabilir.
- Bir veri sınıfında tanımlanan veri özelliklerini değiştirdiğinizde, otomatik olarak oluşturulan kod güncellenir.
 - Daha az kodu yönetmeniz ve hata ayıklamanız gerekmektedir.
- Gerekli değişken açıklamaları, statik kod analizi araçlarından faydalanmanızı sağlar.
 - Yürütme zamanında ortaya çıkmadan önce ek hataları önleyebilirsiniz.
 - Bazı statik kod analizi araçları ve IDE'ler, değişken açıklamalarını inceleyebilir ve yanlış türü kullanan kodlarda uyarı verebilir.

```
1 from dataclasses import dataclass
2
3 @dataclass
4 class InventoryItem:
5     name: str
6     unit_price: float
7     quantity_on_hand: int = 0
```

```

8
9     def total_cost(self) → float:
10         return self.unit_price * self.quantity_on_hand

```

```

1 #without dataclasses
2 class Deneme:          # değişkenlerin tipi
3     def __init__(self, ad:str, soyad:str):
4         self.ad=ad
5         self.soyad=soyad
6         # döndürülecek değer tipi
7 def main() → None:
8     d1=Deneme(ad="Ali", soyad="Bursalı")
9     print(d1)
10
11 if __name__=="__main__":
12     main()

```

<__main__.Deneme object at 0x0000001DC3C5395B0>

```

1 #without dataclasses
2 class Deneme:
3     def __init__(self, ad:str, soyad:str):
4         self.ad=ad
5         self.soyad=soyad
6
7     #with __repr__
8     def __repr__(self) → str:
9         return f"{self.ad}, {self.soyad}"
10
11 def main() → None:
12     d1=Deneme(ad="Ali", soyad="Bursalı")
13     print(d1)
14
15 if __name__=="__main__":
16     main()

```

Ali, Bursalı

1 # Dataclass kullanarak

```

1 from dataclasses import dataclass
2
3 @dataclass
4 class Deneme:
5     ad:str
6     soyad:str
7
8 def main() → None:
9     d1=Deneme(ad="Ali", soyad="Bursalı")
10    print(d1)
11
12 if __name__=="__main__":
13    main()

```

Deneme(ad='Ali', soyad='Bursalı')

```

1 @dataclass
2 class Deneme:
3     ad:str
4     soyad:str
5     durum:bool = True #default değer
6
7 def main() → None:
8     d1=Deneme(ad="Ali", soyad="Bursalı", durum=False)
9     print(d1)
10
11 if __name__=="__main__":
12     main()

```

Deneme(ad='Ali', soyad='Bursalı', durum=False)

```

1 from dataclasses import dataclass, field
2 import random
3 @dataclass
4 class Deneme:
5     ad:str
6     soyad:str
7     durum:bool = True #default value

```



```

8 #liste: list[str] = [] #problematic for every new instance new list
9 liste: list[str] = field(default_factory=list)
10
11 def main() → None:
12     d1=Deneme(ad="Ali", soyad="Bursalı", durum=False)
13     print(d1)
14
15 if __name__=="__main__":
16     main()

```

```
Deneme(ad='Ali', soyad='Bursalı', durum=False, liste=[])
```

```

1 from dataclasses import dataclass, field
2 import random
3 @dataclass
4 class Deneme:
5     ad:str
6     soyad:str
7     durum:bool = True #default value
8     #listeler: list[str] = [] problematic for every new instance new list
9     listeler: list[str] = field(default_factory=list)
10    sayac: int = field(default_factory=random.random)
11
12 def main() → None:
13     d1=Deneme(ad="Ali", soyad="Bursalı")
14     print(d1)
15     d2=Deneme(ad="Ayşe", soyad="Bolulu")
16     print(d2)
17
18 if __name__=="__main__":
19     main()

```

```
Deneme(ad='Ali', soyad='Bursalı', durum=True, listeler=[], sayac=0.3888101903029909)
Deneme(ad='Ayşe', soyad='Bolulu', durum=True, listeler=[], sayac=0.9934819393247375)
```

```

1 from dataclasses import dataclass, field
2 import random
3 @dataclass
4 class Deneme:
5     ad:str
6     soyad:str
7     durum:bool = True #default value
8     #listeler: list[str] = [] problematic for every new instance new list
9     listeler: list[str] = field(default_factory=list)
10    names : list = field(default_factory=lambda : [])
11
12 def main() → None:
13     d1=Deneme(ad="Ali", soyad="Bursalı")
14     print(d1)
15     d2=Deneme(ad="Ayşe", soyad="Bolulu")
16     print(d2)
17     d1.names.append('Apple')
18     d1.names.append('Banana')
19     print(d1) # Fruits(names=['Apple', 'Banana'])
20     #d3 = Deneme(['Peach', 'Pear'])
21     #print(d3)
22
23 if __name__=="__main__":
24     main()

```

```
Deneme(ad='Ali', soyad='Bursalı', durum=True, listeler=[], names=[])
Deneme(ad='Ayşe', soyad='Bolulu', durum=True, listeler=[], names=[])
Deneme(ad='Ali', soyad='Bursalı', durum=True, listeler=[], names=['Apple', 'Banana'])
```

```

1 from dataclasses import dataclass, field
2 import random
3 @dataclass
4 class Deneme:
5     ad:str
6     soyad:str
7     durum:bool = True #default value
8     #listeler: list[str] = [] problematic for every new instance new list
9     listeler: list[str] = field(default_factory=list)
10    sayac: int = field(init=True, default_factory=random.random)#init default değeri True, sayaca varsayılan olarak random
11
12 def main() → None:
13     d1=Deneme(ad="Ali", soyad="Bursalı", sayac="123") #init=False → hata verir
14     print(d1)
15     d2=Deneme(ad="Ayşe", soyad="Bolulu")
16     print(d2)
17

```

```
18 if __name__=="__main__":
19     main()
```

```
Deneme(ad='Ali', soyad='Bursalı', durum=True, listeler=[], sayac='123')
Deneme(ad='Ayşe', soyad='Bolulu', durum=True, listeler=[], sayac=0.8117431361444702)
```

- `_ara: str = field(init=False)` satırı, `Deneme` sınıfı için `_ara` isimli bir özellik tanımlar. Bu özellik, ad ve soyad özelliklerinin birleşimini tutacak bir özelliktir. *
- *`ara` özelliği, `init=False` argümanı ile tanımlanmıştır. `init` argümanı, özelliğin `_init` metodunda atanıp atanmayacağını belirler. `init=False` olarak ayarlandığında, `ara` özelliği, sınıfın `_init` metodunda atanmaz. Bunun yerine, *`ara` değeri*, `_ post_init __` adlı bir özel metotta atanır.*
-
- `field` dekoratörü, `dataclass` sınıfının özelliklerini tanımlamak için kullanılır. Bu dekoratör, özelliklere varsayılan değerler atamak, özelliklerin varsayılan değerlerini değiştirmek, özelliklerin okunabilirliğini veya yazılabilirliğini değiştirmek gibi özelliklere sahiptir. `_ara: str = field(init=False)` satırındaki `_` (alt çizgi) sembolü, Python'da genellikle ismi önemsiz olan bir değişkeni ifade etmek için kullanılır. `_` sembolü, `_ara` özelliğinin isminin önemsiz olduğunu gösterir.
-
- `field` ise, `dataclass` sınıfının özelliklerini tanımlamak için kullanılan bir dekoratördür. `field` dekoratörü, özelliklerin varsayılan değerlerini belirlemek, yazılabilirliğini veya okunabilirliğini değiştirmek, özellikleri sınıfın dışında tanımlamak gibi özelliklere sahiptir.
-
- Bu örnekte *`ara` özelliği, `init=False` parametresi ile tanımlanmıştır. Bu, `ara` özelliğinin sınıfın `_init` metodunda atanmayacağı anlamına gelir. Bunun yerine, `ara` özelliği `_ post_init __` adlı bir özel metotta atanır.*

```
1 from dataclasses import dataclass, field
2 import random
3
4 @dataclass
5 class Deneme:
6     ad: str
7     soyad: str
8     durum: bool = True # Varsayılan değer
9     listeler: list[str] = field(default_factory=list) # Boş bir liste oluşturulur
10    sayac: int = field(init=False, default_factory=random.random) # Rasgele bir sayı atanır
11    _ara: str = field(init=False) # "ad" ve "soyad" özelliklerinin birleşimini tutacak bir özellik
12
13    def __post_init__(self) → None:
14        self._ara = f"{self.ad} {self.soyad}" # "_ara" özelliği "ad" ve "soyad" özelliklerinin birleşiminden oluşturulur
15
16 def main() → None:
17     # "Deneme" sınıfından bir örnek oluşturulur
18     d1 = Deneme(ad="Ali", soyad="Bursalı")
19
20     # Oluşturulan örnek ekrana yazdırılır
21     print(d1)
22
23 if __name__ == "__main__":
24     main()
```

```
Deneme(ad='Ali', soyad='Bursalı', durum=True, listeler=[], sayac=0.31101655520285154, _ara='Ali Bursalı')
```

```
1 from dataclasses import dataclass, field
2 import random
3 @dataclass
4 class Deneme:
5     ad:str
6     soyad:str
7     durum:bool = True #default value
8     #listeler: list[str] = [] problematic for every new instance new list
9     listeler: list[str] = field(default_factory=list)
10    sayac: int = field(init=False, default_factory=random.random)
11    _ara: str = field(init=False, repr=False) #internal usage
12
13    def __post_init__(self)→None:
14        self._ara=f"{self.ad} {self.soyad}"
15
16 def main() → None:
17     d1=Deneme(ad="Ali", soyad="Bursalı")
18     print(d1)
19
20 if __name__=="__main__":
21     main()
```

```
Deneme(ad='Ali', soyad='Bursalı', durum=True, listeler=[], sayac=0.0019055482659819933)
```

```
1 from dataclasses import dataclass, field
2 import random
```

```

3 @dataclass (frozen=True)
4 class Deneme:
5     ad:str
6     soyad:str
7     durum:bool = True #default value
8     #listeler: list[str] = [] problematic for every new instance new list
9     listeler: list[str] = field(default_factory=list)
10    sayac: int = field(init=False, default_factory=random.random)
11    _ara: str = field(init=False, repr=False) #internal usage
12
13    def __post_init__(self)→None:
14        #self._ara=f"{self.ad} {self.soyad}"
15        pass
16 def main() → None:
17     d1=Deneme(ad="Ali", soyad="Bursalı")
18     print(d1)
19
20 if __name__=="__main__":
21     main()

```

Deneme(ad='Ali', soyad='Bursalı', durum=True, listeler=[], sayac=0.8996440586690976)

10.14 Docstring ve doctest ile Birim Testleri

**** Kolaylık sağlaması için bu notebook, accountdoctest.py dosyasının tamamını içermektedir, böylece onu değiştirebilir ve testleri yeniden çalıştırabilirsiniz. accountdoctest.py içeren hücreyi çalıştırdığınızda, doctestler çalışacaktır. ****

- Yazılım geliştirmenin önemli bir yönü, kodunuzun doğru çalıştığından emin olmak için onu test etmektir.
- Kapsamlı bir test yapmış olsanız bile, kod hala hatalar içerebilir.
- Ünlü Hollandalı bilgisayar bilimcisi Edsger Dijkstra'ya göre, "Test etmek hataların yokluğunu değil, varlığını gösterir."

Modül doctest ve testmod Fonksiyonu

- doctest modülü, kodunuzu test etmenize ve değişiklikler yaptıktan sonra kolayca yeniden test etmenize yardımcı olur.
- doctest modülünün testmod fonksiyonu çalıştırıldığında, fonksiyonlarınızın, metodlarınızın ve sınıflarınızın docstring'lerini inceler ve her biri >>> ile başlayan örnek Python ifadelerini arar. Her bir ifade bir sonraki satırda beklenen çıktıyı (varsa) belirtir.
- testmod, bu ifadeleri çalıştırır ve beklenen çıktıyı üretip üretmediğini doğrular.
- Eğer doğru çıktı üretilmezse, testmod hataları rapor eder ve hangi testlerin başarısız olduğunu belirtir, böylece kodunuzdaki sorunları bulup düzeltebilirsiniz.
- Bir docstring içinde tanımladığınız her bir test, bir fonksiyon, bir metod veya bir sınıf gibi belirli bir kod birimini test eder.
- Bu tür testlere birim testleri denir.

Değiştirilmiş Account Sınıfı

accountdoctest.py dosyası, bu bölümün ilk örneğindeki Account sınıfını içerir.

- **init** metodunun docstring'i dört test içerecek şekilde değiştirildi ve bu testler, metodun doğru çalıştığını sağlamak için kullanılabilir:
 - İlk test, account1 adında bir örnek Account nesnesi oluşturur
 - Bu ifade herhangi bir çıktı üretmez
 - İkinci test, ilk test başarıyla gerçekleştirildiyse account1'in name özelliğinin değeri ne olması gerektiğini gösterir
 - Üçüncü test, ilk test başarıyla gerçekleştirildiyse account1'in balance özelliğinin değeri ne olması gerektiğini gösterir
 - Son test, geçersiz bir başlangıç bakiyesiyle bir Account nesnesi oluşturur
 - Örnek çıktı, bu durumda bir ValueError istisnasının meydana gelmesi gerektiğini gösterir
 - İstisnalar için, doctest modülünün belgeleri sadece takip zincirinin ilk ve son satırlarını göstermeyi önerir
- Testlerinizi açıklayıcı metinlerle birleştirebilirsiniz.

```

1 # accountdoctest.py
2 """Account class definition."""
3 from decimal import Decimal
4
5 class Account:
6     """Account class for demonstrating doctest."""
7
8     def __init__(self, name, balance):
9         """Initialize an Account object.
10
11         >>> account1 = Account('John Green', Decimal('50.00'))
12         >>> account1.name

```

```

13     'John Green'
14     >>> account1.balance
15     Decimal('50.00')
16
17     The balance argument must be greater than or equal to 0.
18     >>> account2 = Account('John Green', Decimal('-50.00'))
19     Traceback (most recent call last):
20     ...
21     ValueError: Initial balance must be ≥ to 0.00.
22     """
23
24     # if balance is less than 0.00, raise an exception
25     if balance < Decimal('0.00'):
26         raise ValueError('Initial balance must be ≥ to 0.00.')
27
28     self.name = name
29     self.balance = balance
30
31     def deposit(self, amount):
32         """Deposit money to the account."""
33
34         # if amount is less than 0.00, raise an exception
35         if amount < Decimal('0.00'):
36             raise ValueError('amount must be positive.')
37
38         self.balance += amount
39
40 if __name__ == '__main__':
41     import doctest
42     doctest.testmod(verbose=True)
43

```

```

Trying:
    account1 = Account('John Green', Decimal('50.00'))
Expecting nothing
ok
Trying:
    account1.name
Expecting:
    'John Green'
ok
Trying:
    account1.balance
Expecting:
    Decimal('50.00')
ok
Trying:
    account2 = Account('John Green', Decimal('-50.00'))
Expecting:
    Traceback (most recent call last):
    ...
    ValueError: Initial balance must be ≥ to 0.00.
ok
71 items had no tests:
    __main__
    __main__.Account
    __main__.Account.deposit
    __main__.Card
    __main__.CommissionEmployee
    __main__.CommissionEmployee.__init__
    __main__.CommissionEmployee.__repr__
    __main__.CommissionEmployee.commission_rate
    __main__.CommissionEmployee.earnings
    __main__.CommissionEmployee.first_name
    __main__.CommissionEmployee.gross_sales
    __main__.CommissionEmployee.last_name
    __main__.CommissionEmployee.ssn
    __main__.Complex
    __main__.Complex.__add__
    __main__.Complex.__iadd__
    __main__.Complex.__imul__
    __main__.Complex.__init__
    __main__.Complex.__isub__
    __main__.Complex.__mul__
    __main__.Complex.__repr__
    __main__.Complex.__str__
    __main__.Complex.__sub__
    __main__.Demo
    __main__.Demo.__eq__
    __main__.Demo.__init__
    __main__.Demo.__repr__
    __main__.Deneme
    __main__.Deneme.__delattr__
    __main__.Deneme.__eq__
    __main__.Deneme.__hash__
    __main__.Deneme.__init__
    __main__.Deneme.__post_init__

```

```
__main__.Deneme.__repr__
__main__.Deneme.__setattr__
__main__.InventoryItem
```

Module `__main__`

- Herhangi bir modülü yüklediğinizde, Python modülün adını içeren bir dizeyi, modülün **name** adlı global bir özneliğine atar.
- Bir Python kaynak dosyasını bir betik olarak çalıştırdığınızda, Python, modülün adı olarak **'main'** dizesini kullanır.
- name**'i bir if ifadesinde kullanarak, kaynak dosyasının yalnızca bir betik olarak çalıştırıldığında çalışması gereken kodu belirtebilirsiniz.
 - doctest modülünü içe aktarır ve modülün testmod fonksiyonunu çağırarak docstring birim testlerini yürütür.

Running Tests

- Testleri yürütmek için `accountdoctest.py` dosyasını bir betik olarak çalıştırın.
- Eğer `testmod`'u hiçbir argüman vermeden çağırırsanız, başarılı test sonuçlarını göstermez.
- Bu örnekte, `testmod`'u `verbose=True` anahtar kelime argümanı ile çağırarak her testin sonuçlarını gösteririz.
- Bir başarısız testi göstermek için, `accountdoctest.py` dosyasındaki 25-26. satırları başına `#` işareti ekleyerek yorum satırı yapın, ardından `accountdoctest.py` dosyasını bir betik olarak çalıştırın.

IPython `%doctest_mode` Magic

- Varolan kod için doctest oluşturmanın pratik bir yolu, kodunuzu test etmek için IPython etkileşimli bir oturum kullanmak ve bu oturumu bir `docstring`'e kopyalamak ve yapıştırmaktır.
- IPython'in `In []` ve `Out []` işaretleri doctest ile uyumlu değildir, bu yüzden IPython, doğru doctest formatında işaretleri görüntülemek için `%doctest_mode` büyüsünü sağlar.
 - İki işaret stili arasında geçiş yapar.

10.15 İsim Alanları ve Kapsamlar

- Her tanımlayıcının, programınızda nerede kullanabileceğinizi belirleyen bir kapsamı vardır ve yerel ve global kapsamları tanıttık.
- Kapsamlar, tanımlayıcıları nesnelerle ilişkilendiren ve sözlük olarak "arka planda" uygulanan isim alanları tarafından belirlenir.
- İsim alanları birbirinden bağımsızdır.
 - Aynı tanımlayıcı birden çok isim alanında görünebilir.
- Üç temel isim alanı vardır: yerel, global ve yerleşik.

Yerel İsim Alanı

- Her fonksiyonun ve metodun bir yerel isim alanı vardır ve yerel tanımlayıcıları (parametreler ve yerel değişkenler) nesnelerle ilişkilendirir.
- Yerel isim alanı, fonksiyon veya metod çağrıldığı anda oluşur ve sonlandığı ana kadar var olur, yalnızca o fonksiyon veya metoda erişilebilir.
- Bir fonksiyonun veya metodun bloğunda, mevcut olmayan bir değişkene atama yapmak, yerel bir değişken oluşturur ve yerel isim alanına ekler.
- Yerel isim alanındaki tanımlayıcılar, onları tanımladığınız noktadan fonksiyon veya metodun sonlandığı ana kadar geçerli kapsamdadır.

Global İsim Alanı (Global Namespace)

- Her modülün bir global isim alanı vardır ve bir modülün global tanımlayıcılarını (global değişkenler, fonksiyon adları ve sınıf adları gibi) nesnelerle ilişkilendirir.
- Python, modülü yüklediğinde modülün global isim alanını oluşturur.
- Modülün global isim alanı ve tanımlayıcıları, programın (veya etkileşimli oturumun) sona erdiği ana kadar o modüldeki kod için geçerli kapsamdadır.
- Bir IPython oturumu, bu oturumda oluşturduğunuz tüm tanımlayıcılar için kendi global isim alanına sahiptir.
- Her modülün global isim alanı ayrıca modülün adını içeren bir tanımlayıcı olan **name** adlı bir tanımlayıcı içerir. Örneğin, `math` modülü için `'math'` veya `random` modülü için `'random'` gibi.

Yerleşik İsim Alanı (Built-In Namespace)

- Python'un yerleşik işlevlerini (örneğin, `input` ve `range`) ve tiplerini (örneğin, `int`, `float` ve `str`) tanımlayan nesneleri ilişkilendiren tanımlayıcıları içerir.
- Python yorumlayıcısı çalışmaya başladığında yerleşik isim alanını oluşturur.
- Yerleşik isim alanının tanımlayıcıları, programın (veya etkileşimli oturumun) sona erdiği ana kadar tüm kod için geçerli kapsamda kalır.

İsim Alanlarında Tanımlayıcıları Bulma (Finding Identifiers in Namespaces)

- Bir tanımlayıcı kullandığınızda, Python o tanımlayıcıyı şu anda erişilebilir olan isim alanlarında arar ve sırasıyla yerel, global ve yerleşik olarak ilerler.

```
1 z = 'global z'
```

```
1 del z
```

```
1
2 def print_variables():
3     y = 'local y in print_variables'
4     print(y)
5     print(z)
```

```
1 print_variables()
```

```
local y in print_variables
global z
```

- Snippet [3] print_variables çağırıldığında, Python aşağıdaki şekilde yerel, global ve yerleşik isim alanlarında arama yapar:
 - Snippet [3] bir fonksiyon veya metodun içinde değil, bu yüzden oturumun global isim alanı ve yerleşik isim alanı şu anda erişilebilir durumdadır
 - Python önce oturumun global isim alanında print_variables'ı arar
 - print_variables, kapsam içinde olduğu için, Python ilgili nesneyi kullanarak print_variables'ı çağırır
 - print_variables çalışmaya başladığında, Python fonksiyonun yerel isim alanını oluşturur
 - print_variables fonksiyonu, yerel değişken y'yi tanımladığında, Python y'yi fonksiyonun yerel isim alanına ekler
 - Değişken y artık fonksiyon sona erene kadar kapsam içinde kalır.
 - Ardından, print_variables, y'yi argüman olarak geçirerek yerleşik fonksiyon print'i çağırır
 - Bu çağırışı gerçekleştirmek için, Python y ve print tanımlayıcılarını çözmeli.
 - y tanımlayıcısı yerel isim alanında tanımlanmış olduğundan kapsam içindedir ve Python print'in argümanı olarak ilgili nesneyi kullanacak
 - Fonksiyonu çağırmak için, Python print'in ilgili nesnesini bulmalı
 - Python, yerel isim alanında print'i bulamaz
 - Sonra Python, oturumun global isim alanına bakar ve print'i bulamaz
 - Son olarak, Python, yerleşik isim alanına bakar ve print'i bulur
 - print, kapsam içinde olduğu için, Python ilgili nesneyi kullanarak print'i çağırır
 - Ardından, print_variables, yerleşik fonksiyon print'i tekrar z argümanı ile çağırır, ancak z yerel isim alanında tanımlanmamıştır
 - Python, global isim alanına bakar
 - z argümanı, global isim alanında tanımlı olduğundan, z kapsam içindedir ve Python print'in argümanı olarak ilgili nesneyi kullanır
- Bu noktada, print_variables fonksiyonunun suite'inin sonuna ulaşıyoruz, bu yüzden fonksiyon sona erer ve yerel isim alanı artık mevcut değildir, bu da yerel değeri

```
1 y
```

```
-----
NameError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_9484\3563912222.py in <module>
----> 1 y

NameError: name 'y' is not defined
```

SEARCH STACK OVERFLOW

- Bir yerel isim alanı olmadığı için Python, y'yi oturumun global isim alanında arar
- Tanımlayıcı y orada tanımlanmamıştır, bu yüzden Python, y'yi yerleşik isim alanında arar
- Tekrar Python, y'yi bulamaz
- Python, y tanımlı değil hatası (NameError) döner.
- print_variables ve z tanımlayıcıları hala oturumun global isim alanında var olduğu için bunları kullanmaya devam edebiliriz.

İç İç Fonksiyonlar (Nested Functions)

- Ayrıca bir ****** kapsayan isim alanı ****** da bulunur. (**enclosing namespace**)
- Python size diğer fonksiyonların veya metodların içinde iç içe fonksiyonlar tanımlamanıza izin verir.
- İç içe bir fonksiyon içinde bir tanımlayıcıya eriştiğinizde, Python önce iç içe fonksiyonun yerel isim alanını, ardından kapsayan fonksiyonun isim alanını, sonra global isim alanını ve son olarak yerleşik isim alanını arar.
- Bu bazen LEGB (local, enclosing, global, built-in) kuralı olarak adlandırılır.

Sınıf İsim Alanı (Class Namespace)

- Bir sınıfın, sınıf niteliklerinin depolandığı bir isim alanı vardır.
- Bir sınıf niteliğine eriştiğinizde, Python önce o niteliği sınıfın isim alanında arar, ardından temel sınıfın isim alanında ve bu şekilde devam eder, ya niteliği bulur ya da object sınıfına ulaşır.
- Nitelik bulunamazsa, bir NameError hatası oluşur.

Nesne İsim Alanı (Object Namespace)

- Her nesnenin, nesnenin yöntemlerini ve veri niteliklerini içeren kendi bir isim alanı vardır.
- Sınıfın **init** yöntemi boş bir nesne (self) ile başlar ve her niteliği nesnenin isim alanına ekler.
- Bir nesnenin isim alanında bir niteliği tanımladıktan sonra, nesneyi kullanan istemciler niteliğin değerine erişebilir.

10.16 Intro to Data Science: Time Series and Simple Linear Regression

- Zaman serisi: Zaman noktalarıyla ilişkilendirilmiş değerlerin (gözlemler) dizileridir.
 - Günlük kapanış hisse senedi fiyatları
 - Saatlik sıcaklık ölçümleri
 - Uçak uçuşundaki konum değişiklikleri
 - Yıllık ürün verimleri
 - Dönemsel şirket karları
 - Dünya çapındaki Twitter kullanıcılarının zaman damgalı tweetleri
- Basit doğrusal regresyon kullanarak zaman serisi verilerinden tahminler yapacağız.

Zaman Serisi (Time Series)

- Tek değişkenli zaman serisi: Her bir zaman noktası için bir gözlem içerir.
- Çok değişkenli zaman serisi: Her bir zaman noktası için iki veya daha fazla gözlem içerir. Zaman serisiyle genellikle yapılan iki işlem şunlardır:
 - Zaman serisi analizi, mevcut zaman serisi verilerine desenler (örneğin mevsimsellik) bakarak veri analistlerinin veriyi anlamalarına yardımcı olur.
 - Zaman serisi tahmini, geçmiş verileri kullanarak geleceği tahmin etmek için yapılır.
- Biz zaman serisi tahmini gerçekleştireceğiz.

Basit Doğrusal Regresyon (Simple Linear Regression)

- Bağımsız bir değişkeni (ay/yıl kombinasyonunu) ve bağımlı bir değişkeni (o ay/yıl için ortalama yüksek sıcaklık) temsil eden bir değer koleksiyonu verildiğinde, basit doğrusal regresyon bu değişkenler arasındaki ilişkiyi bir doğru çizgisi olarak tanımlar, bu doğruya regresyon doğrusu denir.

Linear Relationships

- Given a Fahrenheit temperature, we can calculate the corresponding Celsius temperature using:

$$c = 5 / 9 * (f - 32)$$

- **f** (the Fahrenheit temperature) is the *independent variable*
- **c** (the Celsius temperature) is the *dependent variable*
- Each value of **c** *depends on* the value of **f** used in the calculation

Linear Relationships (cont.)

- Plotting Fahrenheit temperatures and their corresponding Celsius temperatures produces a straight line

enable high-res images in notebook

```
%config InlineBackend.figure_format = 'retina' %matplotlib inline c = lambda f: 5 / 9 * (f - 32)
```

```
1 temps = [(f, c(f)) for f in range(0, 101, 10)]
```

```
-----
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_9484\1286481937.py in <module>
----> 1 temps = [(f, c(f)) for f in range(0, 101, 10)]

~\AppData\Local\Temp\ipykernel_9484\1286481937.py in <listcomp>(.0)
----> 1 temps = [(f, c(f)) for f in range(0, 101, 10)]

TypeError: 'CommissionEmployee' object is not callable
```

SEARCH STACK OVERFLOW

```
1 temps
```

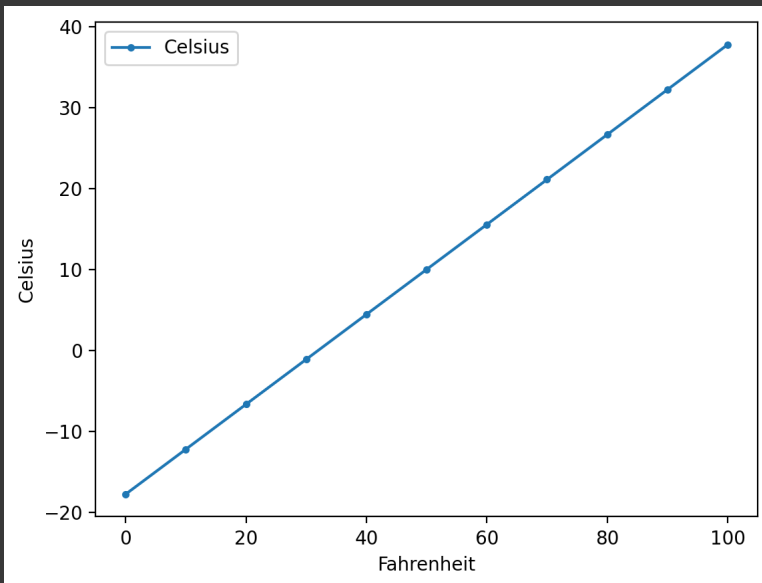
```
[(0, -17.77777777777778),
 (10, -12.222222222222223),
 (20, -6.666666666666667),
 (30, -1.1111111111111112),
 (40, 4.444444444444445),
 (50, 10.0),
 (60, 15.555555555555557),
 (70, 21.11111111111111),
 (80, 26.666666666666668),
 (90, 32.22222222222222),
 (100, 37.77777777777778)]
```

- Verileri bir DataFrame içine yerleştirin, ardından plot yöntemini kullanarak sıcaklıklar arasındaki doğrusal ilişkiyi gösterin
- style kelime argümanı verilerin görünümünü kontrol eder
 - '.', her noktanın nokta olarak görünmesi ve noktaların birleşik çizgilerle birleştirilmesi gerektiğini gösterir

```
1 import pandas as pd
```

```
1 temps_df = pd.DataFrame(temps, columns=['Fahrenheit', 'Celsius'])
```

```
1 axes = temps_df.plot(x='Fahrenheit', y='Celsius', style='.-')
2 y_label = axes.set_ylabel('Celsius')
```



Basit Doğrusal Regresyon Denklemi Bileşenleri

Herhangi bir düz çizginin noktaları aşağıdaki şekilde hesaplanabilir:

$$y = m x + b$$

- m, çizginin eğimidir,
- b, çizginin y-ekseniyle olan kesişimidir (x = 0'da),

- x, bağımsız değişkenidir (bu örnekte tarih),
- y, bağımlı değişkenidir (bu örnekte sıcaklık),
- Basit doğrusal regresyonda, verilen bir x için y tahmin edilen değerdir.

SciPy'nin stats Modülündeki linregress Fonksiyonu

- Basit doğrusal regresyon, verilerinize en iyi uyan bir doğru çizgide eğimi (m) ve kesişimi (b) belirler
- Aşağıdaki diyagram, bu bölümde işleyeceğimiz zaman serisi verilerinin birkaç noktasını ve buna karşılık gelen regresyon doğrusunu göstermektedir
 - Her bir veri noktasının regresyon doğrusundan olan uzaklığını göstermek için dikey çizgiler ekledik

SciPy'nin stats Modülündeki linregress Fonksiyonu (cont.)

- Basit doğrusal regresyon algoritması, eğimi ve kesişimi iteratif olarak ayarlar ve her bir ayarlama, her bir noktanın doğrudan olan uzaklığının karesini hesaplar
- "En iyi uyum" gerçekleşirken, eğim ve kesişim değerleri bu kareli uzaklıkların toplamını en aza indirir
 - en küçük kareler yöntemi hesaplaması
- SciPy (Scientific Python), Python'da mühendislik, bilim ve matematik için yaygın olarak kullanılan bir kütüphanedir
 - linregress fonksiyonu (scipy.stats modülünden) basit doğrusal regresyonu sizin için gerçekleştirir

Getting Weather Data from NOAA

- The National Oceanic and Atmospheric Administration (NOAA) offers public historical data including time series for average high temperatures in specific cities over various time intervals
- Obtained the January average high temperatures for New York City from 1895 through 2018 from NOAA's "Climate at a Glance" time series at:

<https://www.ncdc.noaa.gov/cag/>

- `ave_hi_nyc_jan_1895-2018.csv` in the `ch10` examples folder
- Three columns per observation:
 - `Date` —A value of the form 'YYYYMM' (such as '201801'). MM is always 01 because we downloaded data for only January of each year.
 - `Value` —A floating-point Fahrenheit temperature.
 - `Anomaly` —The difference between the value for the given date and average values for all dates (not used in this example)

NOAA'dan Hava Verisi Almak

- National Oceanic and Atmospheric Administration (NOAA), belirli şehirler için farklı zaman aralıklarında ortalama yüksek sıcaklık zaman serilerini içeren halka açık tarihsel veriler sunar.
- NOAA'nın "Climate at a Glance" zaman serisinden 1895'ten 2018'e kadar olan Ocak ayı ortalama yüksek sıcaklıkları için New York City verilerini aldık.
- <https://www.ncdc.noaa.gov/cag/>
- `ave_hi_nyc_jan_1895-2018.csv` adlı dosya, `ch10` örnekler klasöründe bulunmaktadır.
- Her bir gözlem için üç sütun bulunur:
 - `Date` - 'YYYYMM' formatında bir değer (örneğin '201801'). MM, yalnızca Ocak ayı için verileri indirdiğimiz için her zaman 01'dir.
 - `Value` - Ondalık noktalı Fahrenheit sıcaklık.
 - `Anomaly` - Verilen tarih için değerin tüm tarihler için ortalama değerlerden farkı (bu örnekte kullanılmamaktadır).

▼ Loading the Average High Temperatures into a DataFrame

1