Python 3.7

Makine Öğrenmesi Odaklı Python

Yazılım Mühendisliği Bölümü
Atılım Üniversitesi

Tolga Üstünkök

Ön Bilgilendirmeler

- Bazı İngilizce terimler oldukları gibi bırakılmışlardır. Bu terimleri slaytlar içerisinde *italic* yazılmış olarak görebilirsiniz.
- Bu sunumdaki konu akışı ve örnekler Luciano Ramalho'nun *Fluent Python* kitabından alınmıştır.

Agenda

- Prolog: Pythonism'in Önemi – Tutarlılık
- Sayısal Tipleri Taklit Etmek
- Python Veri Yapıları
- List Comprehensions
- Generator Expressions
- Tuple Açma[*Tuple Unpacking*]
- Dilimleme[Slicing]
- Fonksiyonlar Birinci Sınıf Objelerdir
- Fonksiyon Dekoratörleri
- Değişkenler Kutu Değildir!
- Operatör Aşırı Yükleme
- Epilog: Makine Öğrenmesi

Prolog: Pythonism'in Önemi - Tutarlılık

- Eğer başka object-oriented dilleri biliyorsanız

```
collection.len()
```

  yerine

```
len(collection)
```

  şeklinde bir kullanım garip gelebilir.
- Bu tarz bir kullanım aslında buz dağının görünen kısmıdır.
- Düzgün bir şekilde anlaşıldığında bu kullanım *Pythonic* adı verilen kod yazım tarzında Python scriptleri yazmanın anahtarıdır.

Prolog: Pythonism'in Önemi - Tutarlılık

- Python bir programlama dili olarak tanımlansa da bir framework gibi işler.
- Python'ı kullanırken de bir framework'te olduğu gibi framework tarafından çağrılan metodları implement edersiniz.
- Bu özel metodlar her zaman iki adet *underscore* (_) ile başlar ve iki adet *underscore* ile biter.

Prolog: Pythonism'in Önemi - Tutarlılık

- Örnek:
  - __getitem__
    - my_collection.__getitem__(key)

    my_collection adındaki bir *collection* objesinden *my_collection[key]* sözdizimini[syntax] kullanarak veri çekmek için tanımlanması gereken metoddur.
- Python dili bunun gibi onlarca özel metod tanımlamıştır.

Prolog: Pythonism'in Önemi - Tutarlılık

- Özel olarak tanımlanan bu metodlar aşağıda sıralanan dil yapılarını desteklemenize olanak sağlar.
  - Iteration[Tekrarlama]
  - Collections[Koleksiyonlar]
  - Attribute access[Öznitelik erişimi]
  - Operator overloading[Operatör aşırı yüklenmesi]
  - Function and method invocation[Fonksiyon ve metod çağırması]
  - Object creation and destruction[Obje yaratma ve yok etme]
  - String representation and formatting[Karakter dizisi tanımlama ve formatlama]
  - Managed contexts[Ortam yönetimi]

Prolog: Pythonism'in Önemi - Tutarlılık

- Luciano Ramalho, "Fluent Python" adlı kitabında bütün bu özellikleri bünyesinde barındıran "Pythonic Card Deck" adında bir örnek program tasarlamıştır.
- Kitapta ilerlendikçe bu program anlatılan yöntemler ile geliştirilmiştir.
- Biz de bu programa bir göz atalım.

Ne görüyorsunuz?

```
import collections

Card = collections.namedtuple(“Card”, [“rank”, “suit”])

class FrenchDeck:
    ranks = [str(n) for n in range(2, 11)] + list(“JQKA”)
    suits = “spades diamonds clubs hearts”.split()

    def __init__(self):
        self._cards = [Card(rank, suit) for suit in self.suits
                                        for rank in self.ranks]

    def __len__(self):
        return len(self._cards)

    def __getitem__(self, position):
        return self._cards[position]
```

Card adında bir sınıfı tanımlamak için *namedtuple* kullanılmıştır.

Bir *list comprehension* yapısı

Prolog: Pythonism'in Önemi - Tutarlılık

- Şimdi, az önceki örnek "French Deck" sınıfını kullanarak bir karo yedilisi oluşturalım.

```
>>> beer_card = Card("7", "diamonds")
>>> beer_card
Card(rank='7' , suit='diamonds')
```

- Şimdi de bütün bir 52 kartlı deste oluşturalım.

```
>>> deck = FrenchDeck()
>>> len(deck)
52
```

Prolog: Pythonism'in Önemi - Tutarlılık

- Az önce oluşturduğumuz desteden bir tane kart çekelim:

```
>>> deck[0]
Card(rank='2', suit='spades')
>>> deck[-1]
Card(rank='A', suit='hearts')
```

- Şimdi de rastgele bir kart çekelim:

```
>>> from random import choice
>>> choice(deck)
Card(rank='3', suit='hearts')
>>> choice(deck)
Card(rank='K', suit='spades')
```

Prolog: Pythonism'in Önemi - Tutarlılık

- *FrenchDeck* sınıfı *__getitem__* metodunu uyguladığı için artık *dilimlemeyi[slicing]* de desteklemektedir.

```
>>> deck[:3]
[Card(rank='2', suit='spades'), Card(rank='3', suit='spades'),
Card(rank='4', suit='spades')]
```

- Dolayısıyla *for* döngülerini de bu şekilde kullanabilirsiniz.

Prolog: Pythonism'in Önemi - Tutarlılık

- Python'da özel metodlar *interpreter* tarafından çağrılmak üzere tasarlanmış olmalarına rağmen programı yazan kişiler tarafından da çağrılabilirler. *(Bu durum kesinlikle önerilmemektedir.)*
- Örneğin *my_object.__len__()* çağırmak yerine *len(my_object)* çağırmalısınız.
- Bu tarz bir çağırma çoğu *builtin* metod için altta yatan *C* fonksiyonunu çağırır. Dolayısıyla hız artar.

Prolog: Pythonism'in Önemi - Tutarlılık

- Çoğu zaman özel metodların çağırıldığını görmezsiniz. Örneğin;

```
for i in x:
```

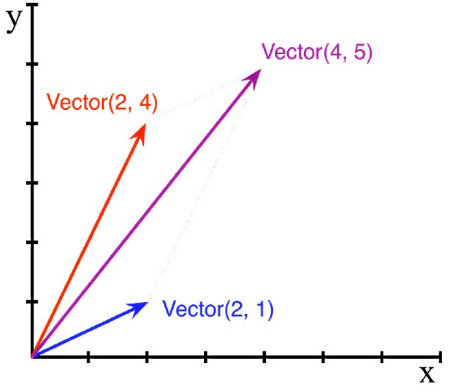
  arka planda *x* objesini *iter(x)* ile çağırır. Bu da *x* objesinin *__iter__()* fonksiyonunu çağırır.

- Bu özel metodları kullanmak Python veri modelini en verimli şekilde kullanmanın anahtarıdır.

Sayısal Tipleri Taklit Etmek

- Vektör işlemlerini kolaylıkla yapabileceğimiz bir sınıf hazırlayacağız.
- Bu sınıf çoğu karmaşık vektör işlemini sadece dört işlem sembolünü kullanarak yapabilmemize olanak sağlayacak. Örneğin;

```
>>> v1 = Vector(2, 4)
>>> v2 = Vector(2, 1)
>>> v1 + v2
Vector(4, 5)
>>> abs(v1)
4.472135
```



Sayısal Tipleri Taklit Etmek

```
from math import hypot

class Vector:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __repr__(self):
        return “Vector(%r, %r)” % (self.x, self.y)

    def __abs__(self):
        return hypot(self.x, self.y)

    def __bool__(self):
        return bool(abs(self))

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Vector(x, y)

    def __mul__(self, scalar):
        return Vector(self.x * scalar, self.y * scalar)
```

Sayısal Tipleri Taklit Etmek

Table 1-1. Special method names (operators excluded)

Category	Method names
String/bytes representation	`__repr__`, `__str__`, `__format__`, `__bytes__`
Conversion to number	`__abs__`, `__bool__`, `__complex__`, `__int__`, `__float__`, `__hash__`, `__index__`
Emulating collections	`__len__`, `__getitem__`, `__setitem__`, `__delitem__`, `__contains__`
Iteration	`__iter__`, `__reversed__`, `__next__`
Emulating callables	`__call__`
Context management	`__enter__`, `__exit__`
Instance creation and destruction	`__new__`, `__init__`, `__del__`
Attribute management	`__getattr__`, `__getattribute__`, `__setattr__`, `__delattr__`, `__dir__`
Attribute descriptors	`__get__`, `__set__`, `__delete__`
Class services	`__prepare__`, `__instancecheck__`, `__subclasscheck__`

Python Veri Yapıları

- Python, performans sebepleri ile birçok veri yapısını *C* dilinde gerçeklemiştir.
- Bunlar;
  - *Container Sequences (list, tuple, collections.deque)*
  - *Flat Sequences (str, bytes, bytearray, memoryview, array.array)*

  şeklinde ikiye ayrılırlar.

Python Veri Yapıları

- Burada dikkat edilmesi gereken önemli bir nokta; *container* tipindeki veri yapıları objelere ait referansları saklarken, *flat* tipindeki veri yapıları objelerin kendilerini saklar.
- Diziler için başka bir gruplandırma ise dizilerin değiştirilebilmelerine göre yapılabilir:
  - *Mutable (list, bytearray, array.array, collections.deque, and memoryview)*
  - *Immutable (tuple, str, bytes)*

Python Veri Yapıları

- *Mutable* veri tiplerine rahatlıkla yeni objeler ekleyip çıkarabilirsiniz.
- *Immutable* veri tipleri ise oluşturuldukları anda ne iseler hayat döngülerinin sonuna kadar o şekilde kalırlar.
- *Mutable* tiplere bir örnek olarak *list* verilebilir.
- *Immutable* tiplere örnek olarak ise *tuple* verilebilir.
- Hem *list* hem de *tuple* farklı tiplerde obje tutabilir.

List Comprehensions

- Python programlarının en önemli veri yapılarından biri olan *list* için birçok özel yapı ve kısa yol tanımlanmıştır.
- Hızlıca bir *list* yaratmanın *Pythonic* yolu *list comprehension (listcomp)*'dır.
- Şimdi bunun bir örneğine bakalım.

List Comprehensions

- Standart yol:

```
>>> symbols = '$¢£¥€'
>>> codes = []
>>> for symbol in symbols:
...     codes.append(ord(symbol))
...
>>> codes
[36, 162, 163, 165, 8364]
```

List Comprehensions

- *Pythonic* yol:

```
>>> symbols = '$¢£¥€'
>>> codes = [ord(symbol) for symbol in symbols]
>>> codes
[36, 162, 163, 165, 8364]
```

List Comprehensions

- Python tanım olarak *functional* bir programlama dili olmasa da, bu tip dillerin bütün özelliklerini taşır.
- Dolayısıyla *functional* programlama dillerinde bulunan *map*, *reduce*, *filter* gibi fonksiyonlar Python'da da bulunur.
- Ancak Python bunların kullanımını teşvik etmez. Çünkü *list comprehension* ve sonradan tartışacağımız bazı yapılar bu fonksiyonların bütün görevlerini üstlenir.

List Comprehensions

```
>>> symbols = '$¢£¥€¤'
>>> beyond_ascii = [ord(s) for s in symbols if ord(s) > 127]
>>> beyond_ascii
[162, 163, 165, 8364, 164]
>>> beyond_ascii = list(filter(lambda c: c > 127, map(ord, symbols)))
>>> beyond_ascii
[162, 163, 165, 8364, 164]
```

Generator Expressions

- *Tuple*, *array* ya da başka tip dizileri oluşturmak için *listcomp*'ları kullanabilirsiniz. Ancak *generator expression (genexp)*'ler çok daha az hafıza kullanırlar.
- Bunun sebebi, *genexp*'lerin bütün elemenları bir seferde oluşturup döndürmek yerine sadece sırası gelen elemanı oluşturup döndürmesidir.
- *Genexp*'lerin sözdizimi *listcomp*'lar ile aynıdır. Tek fark köşeli parantez ([]) yerine normal parantez kullanırlar.

Generator Expressions

- Şimdi bir örnekle *genexp* ile *tuple* oluşturalım.

```
>>> symbols = '$¢£¥€¤'
>>> tuple(ord(symbol) for symbol in symbols)
(36, 162, 163, 165, 8364, 164)
>>> import array
>>> array.array('I', (ord(symbol) for symbol in symbols))
array('I', [36, 162, 163, 165, 8364, 164])
```

- Eğer *genexp* bir fonksiyonun tek argümanıysa parantez kullanımına gerek yoktur.

Tuple Açma[*Tuple Unpacking*]

- Python ile program yazarken *tuple*'lara yalnızca *immutable list*'ler olarak bakmak doğru değildir.
- *Tuple*'ları veritabanı kayıtları olarak da görmek gerekir. Örneğin (şehir, yıl, nüfus, nüfus değişimi (%), alan) özniteliklerinden oluşan bir veritabanı tablosu düşünün.
- Bu tablonun bir kaydını ifade etmek için aşağıdaki gibi bir *tuple* yeterli olacaktır.

```
record = ('Tokyo', 2003, 32450, 0.66, 8014)
```

Tuple Açma[*Tuple Unpacking*]

- Python'ın çok güzel bir özelliği ise bir *tuple*'ın elemanlarını tek tek değişkenlere atmanın çok kolay olmasıdır. İşte bu işleme ***tuple unpacking*** denir. Örnek:

```
city, year, pop, chg, area = ('Tokyo', 2003, 32450, 0.66, 8014)
```

- Başka bir örnek ise tek bir *tuple* değişkenden birçok değişkene açmaktır. Örnek:

```
>>> divmod(20, 8)
(2, 4)
>>> t = (20, 8)
>>> divmod(*t)
(2, 4)
```

Tuple Açma[*Tuple Unpacking*]

- Bir fonksiyondan dönen değerlerin yalnızca bir kısmını tek tek değişkenlere atmak istiyorsanız yine * kullanabilirsiniz. Örneğin;

```
>>> a, b, *rest = range(5)
>>> a, b, rest
(0, 1, [2, 3, 4])
```

Dilimleme[Slicing]

- Python’da bütün dizi tipleri dilimlemeyi destekler. Dilimleme Python ile program yazmanın önemli bir kısmını oluşturur.
- **Not:** *range* fonksiyonu ile dilimleme her zaman son indisi hariç tutarken*[exclusive]* ilk indisi dahil[*inclusive*] eder.
- Şimdi birkaç örnek ile dilimleyi inceliyelim.

Dilimleme[Slicing]

```
>>> l = [10, 20, 30, 40, 50, 60]
>>> l[:2]
[10, 20]
>>> l[2:]
[30, 40, 50, 60]
>>> l[:3]
[10, 20, 30]
>>> l[3:]
[40, 50, 60]
```

Dilimleme[Slicing]

```
>>> s = ‘bicycle’
>>> s[::3]
‘bye’
>>> s[::-1]
‘elcycib’
>>> s[::-2]
‘eccb’
```

Dizilerle + ve * Kullanımı

- Python dizileri + ve * operatörlerini desktekler. Örneğin:

```
>>> l = [1, 2, 3]
>>> l * 5
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> 5 * ‘abcd’
‘abcdabcdabcdabcdabcd’
```

- Bu operatörlerin çalışabilmesi için her iki dizinin tipinin aynı olması gereklidir.
- Yaptığınız programları görselleştirmek için: http://pythontutor.com/visualize.html#mode=edit

Fonksiyonlar Birinci Sınıf Objelerdir

- Python’da fonksiyonlar birinci sınıf*[first-class]* objelerdir.
- Bir objenin birinci sınıf olabilmesi için:
  - Çalışma zamanında*[runtime]* yaratılabilmelidir.
  - Bir veri yapısı içerisinde bir değişkene veya elemana atanabilmelidir.
  - Bir fonksiyona argüman olarak gönderilebilmelidir.
  - Bir fonksiyonun çıktısı olarak döndürülebilmelidir.

Fonksiyonlar Birinci Sınıf Objelerdir

```
>>> def factorial(n):
...     '''returns n!'''
...     return 1 if n < 2 else n * factorial(n – 1)
...
>>> factorial(42)
1405006117752879898543142606244511569936384000000000
>>> factorial.__doc__
'returns n!'
>>> type(factorial)
<class 'function'>
```

Fonksiyonlar Birinci Sınıf Objelerdir

```
>>> fact = factorial
>>> fact
<function factorial at 0x...>
>>> fact(5)
120
>>> map(factorial, range(11))
<map object at 0x...>
>>> list(map(fact, range(11)))
[1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]
```

Fonksiyonlar Birinci Sınıf Objelerdir

- Bir fonksiyonu parametre olarak alan başka fonksiyonlara *higher-order functions*[yüksek mertebeden fonksiyonlar] denir.
- *map* fonksiyonu bunun bir örneğidir.
- Bazı *higher-order* fonksiyonlar *functools* modülünde toplanmıştır. (i.e. *reduce*)

Fonksiyonlar Birinci Sınıf Objelerdir

- Python'da 7 çeşit çağrılabilir[*callable*] obje vardır. Bunlar:
  - Kullanıcı tarafından tanımlanan fonksiyonlar
  - Yerleşik fonksiyonlar (i.e. len)
  - Yerleşik metodlar (i.e. dict.get)
  - Metodlar (sınıflarda tanımlanan fonksiyonlar)
  - Sınıflar
  - Sınıf örnekleri
  - *Generator* fonksiyonlar *(yield* kullanan fonksiyonlar*)*

Fonksiyon Dekoratörleri

- Bir fonksiyonu parametre olarak (dekore edilmiş fonksiyon) alan çağrılabilirlere dekoratör denir.
- Dekoratör parametre olarak aldığı fonksiyon üzerinde bazı işlemler yapıp o fonksiyonu döndürebileceği gibi tamamen yeni bir fonksiyon da döndürebilir.
- Bir örnek ile Python'da nasıl dekoratör yapılabileceğini görelim.

Fonksiyon Dekoratörleri

- Bu tip bir yazım:

```
@decorate
def target():
    print(“running target()”)
```

- Bunun ile aynı etkiye sahiptir:

```
def target():
    print(“running target()”)

target = decorate(target)
```

Fonksiyon Dekoratörleri

- Bir fonksiyon dekoratörü genel olarak bir dekore ettiği fonksiyonu bir yenisiyle değiştir. Örneğin:

```
>>> def deco(func):
...     def inner():
...         print(“running inner()”)
...     return inner
...
>>> @deco
... def target():
...     print(“running target()”)
...
>>> target()
running inner()
>>> target
<function deco.<locals>.inner at 0x...>
```

Fonksiyon Dekoratörleri

- Dekoratörler hakkında bilmeniz gereken önemli birşey; dekoratörler dekore ettikleri fonksiyon tanımlanır tanımlanmaz çağrılırlar.
- Bu özellikleri sayesinde bazı yazılım örüntülerinin uygulanmasında oldukça kullanışlıdırlardır. (Örneğin; strategy pattern)

Değişkenler Kutu Değildir!

- Python’da da Java’da olduğu gibi değişkenler objelere verilen referanslardır.
- Yani objenin kendisinden ziyade onlara verilen etiketlerden ibaretlerdir.
- Sonraki slayttaki örnek bunu kanıtlar niteliktedir.

Değişkenler Kutu Değildir!

```
>>> a = [1, 2, 3]
>>> b = a
>>> a.append(4)
>>> b
[1, 2, 3, 4]
```

Değişkenler Kutu Değildir!

- == operatörü değişkenlerin değerlerini karşılaştırır.
- *is* operatörü ise değişkenlerin kimliklerini karşılaştırır.
- Örneğin bir değişkenin *None* olup olmadığının kontrolü şu şekilde yapılabilir:

```
x is None
x is not None
```

Operatör Aşırı Yükleme

- Operatör aşırı yükleme özelliği suistimal edilmeye çok açık bir özelliktir.
- Java'da olmamasının en önemli sebebi James Gosling'in (Java'nın yaratıcısı), C++ dilinde çok fazla insanın operatör aşırı yüklemeyi suistimal ettiğini görmesidir.
- Python operatör aşırı yüklemeyi destekler.

Operatör Aşırı Yükleme

- Ancak Python operatör aşırı yüklemeye bazı limitler getirmiştir. Bunlar:
  - Yerleşik tipler için operatörler aşırı yüklenemez.
  - Yeni operatörler yaratılamaz. Sadece olanlar aşırı yüklenebilir.
  - Bazı operatörler aşırı yüklenemez: *is, and, or, not*
- Her aşırı yüklenebilen operatörün kendine ait bir özel fonksiyonu vardır.

Operatör Aşırı Yükleme

- Daha önce yaratmaya başladığımızı *Vector* sınıfını hatırlayalım.
- Orada __*add*__ ve __*mul*__ şeklinde iki adet fonksiyon tanımlamıştık.
- Bu fonksiyonlar sırasıyla + ve * operatörlerini aşırı yüklemişlerdir ve *Vector(1, 1) + Vector(2, 5) ve ya Vector(1, 2) * 3* şeklinde bir yazımı olanaklı kılmışlardır.

Sayısal Tipleri Taklit Etmek

```
from math import hypot

class Vector:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __repr__(self):
        return “Vector(%r, %r)” % (self.x, self.y)

    def __abs__(self):
        return hypot(self.x, self.y)

    def __bool__(self):
        return bool(abs(self))

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Vector(x, y)

    def __mul__(self, scalar):
        return Vector(self.x * scalar, self.y * scalar)
```

Epilog: Makine Öğrenmesi

- Eğer makine öğrenmesi çalışmaya Python'da başlamaya (ve ya devam etmeye) karar verdiyseniz, şu ana kadar konuştuğumuz bütün başlıkların (ve belki daha fazlasının) büyük önemi vardır.
- Makine öğrenmesini destekleyen hemen hemen bütün API'lar burada konuştuğumuz konuları yoğun biçimde kullanır.

Epilog: Makine Öğrenmesi

- Bunun en güzel örneği *Pandas* kütüphanesidir.
- Şu ana kadar konuştuğumuz bütün konuları (*Numpy* ve *Scipy*'ın da desteği ile) canlı olarak görebileceğiniz ve bizim de yoğun bir şekilde kullanacağımız bir kütüphanedir.
- Dolayısıyla yaptığımız şeylerin anlaşılması için burada konuştuğumuz konuların azami ölçüde anlaşılması önemlidir.

Epilog: Makine Öğrenmesi

- Makine öğrenmesine girmeden önce kısaca veri manipülasyonu hakkında bir bilgilendirme olacak.
- Bu kısa bilgilendirmenin devamı esas konuların içerisine yedirililmiş bir şekilde gelecek.

Epilog: Makine Öğrenmesi

- Makine öğrenmesine iki farklı yoldan giriş yapacağız. Bunlar:
  - Makine öğrenmesindeki standart yöntemler *(Linear Regression, Decision Tree, KNN, SVM, etc.)*
  - Derin öğrenme[*Deep Learning*]
    - *Convolutional Neural Networks (CNN)*
    - *Recurrent Neural Networks (RNN)*
    - *Generative Adversarial Networks (GAN)*

Epilog: Makine Öğrenmesi

- Bu yöntemleri anlatırken bazı kütüphanelerden faydalanacağız.
- Kütüphaneler arası etkileşimi en aza indirmek için iki adet *virtual environment* kullanacağız.
- *Virtual environment* yaratmayı ve *package* yönetimini kolaylaştırmak için *conda package manager*'ı kullanacağız.

Epilog: Makine Öğrenmesi

- *Conda* iki farklı şekilde dağıtılmaktadır:
  - Anaconda
    - Anaconda içerisinde bir miktar sık kullanılan paketi içeren bir dağıtımdır.
  - Miniconda
    - Miniconda içerisinde önceden hiçbir şey yüklü gelmez. Siz ihtiyaçlarınıza göre yüklersiniz.
- Benim kişisel tercihim *Miniconda* yönündedir.

Epilog: Makine Öğrenmesi

- Konu dağılımı şu şekilde planlanmıştır:
  - İkinci gün standart makine öğrenmesi yöntemleri
  - Üçüncü gün derin öğrenme yöntemleri
- Bu konuların rahat bir şekilde anlaşılabilmesi için bir miktar lineer cebir[*linear algebra*] ve istatistik bilgisi gereklidir.
- Sonraki oturumda görüşmek üzere...