İlkel ve İlkel Olmayan Veri Yapıları

Veri yapıları, verileri düzenlemenin ve saklamanın bir yoludur; veriler ve veriler üzerinde gerçekleştirilebilecek çeşitli mantıksal işlemler arasındaki ilişkiyi açıklar. Veri yapılarının sınıflandırılmasının birçok yolu vardır. Bir yol, onları ilkel ve ilkel olmayan veri türleri olarak kategorize etmektir.

Python, kayan noktalar (float), tamsayılar (integer), dizgiler (string) ve Boole'lar (Boolean) gibi ilkel (veya temel) veri yapılarına sahiptir. Python ayrıca listeler, demetler, sözlükler ve kümeler gibi ilkel olmayan veri yapılarına sahiptir. İlkel olmayan veri yapıları, tek bir değer yerine çeşitli biçimlerdeki bir değerler koleksiyonunu depolar. Bazıları, veri yapıları içerisinde başka veri yapıları tutabilir ve veri depolama yeteneklerinde derinlik ve karmaşıklık sağlayabilir

İlkel Olmayan Veri Yapıları

Veri yapıları, verilere erişilebilmesi ve verimli bir şekilde çalışılabilmesi için verileri düzenlemenin ve saklamanın bir yoludur.

İlerleyen günlerde göreceğimiz pandas ve NumPy gibi eklenti kütüphaneleri daha büyük veri kümeleri için gelişmiş hesaplama işlevleri eklese de, bu kütüphaneler, Python'un yerleşik (built-in) veri işleme araçlarıyla birlikte kullanılmak üzere tasarlanmıştır. İlk olarak aşağıdaki Python veri yapılarını göreceğiz:

1. Demetler (tuples),
2. Listeler (lists),
3. Sözlükler (dicts),
4. Kümeler (sets).

Demet (Tuple)

Bir demet (tuple), sabit uzunlukta (fixed-length), değiştirilemez (immutable) bir Python nesneleri sekansıdır. Bir demet oluşturmanın en kolay yolu, virgülle ayrılmış bir değerler sekansı girmektir. Bir dizi virgülle ayrılmış ifade, parantez verilmese bile bunlar tek bir demet olarak ele alınır.

```
In [1]: tup = 4, 5, 6
```

```
In [2]: type(tup)
```

```
Out[2]: tuple
```

Ancak, kimi zaman sekansın başına ve sonuna parantez konur:

In [3]:
```
tup = (4,5,6)
tup
```

Out[3]:
```
(4, 5, 6)
```

In [3]:
```
# boş bir demet oluşturma
s = ()
s= tuple() # tuple() fonksiyonu yerleşik fonksiyondur
```

Bir demetin elemanlarına, diğer birçok sekans türünde olduğu gibi köşeli parantez `[]` ile erişilebilir.

In [1]:
```
tup = tuple('string')
tup
```

Out[1]:
```
('s', 't', 'r', 'i', 'n', 'g')
```

In [2]:
```
tup[3]
```

Out[2]:
```
'i'
```

In [7]:
```
type(tup[4] * 4)
```

Out[7]:
```
str
```

NOT: C, C++, Java ve diğer birçok dilde olduğu gibi, Python'da indeksleme 0 ile başlar. Yani 0'ıncı indeksteki eleman, birinci öğedir:

In [8]:
```
tup[0]
```

Out[8]:
```
's'
```

Bir demet değiştirilemez (immutable) bir nesnedir. Bu nedenle, bir demet oluşturulduktan sonra, o demetteki nesneleri değiştiremezsiniz.

In [9]:
```
tup = ('foo', [1, 2], True)
```

In [10]:
```
tup[1] = 3
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/var/folders/z_/r0991b310gjd66vy5k26g4800000gn/T/ipykernel_15552/695778705.py in <mo
dule>
----> 1 tup[1] = 3

TypeError: 'tuple' object does not support item assignment
```

Bir demet içerisindeki bir elemanda, değiştirilebilir bir nesne bulunuyorsa (örneğin, bir liste), o elemandaki nesneyi değiştirebilirsiniz:

```
In [11]: tup= ('foo', [1, 2], True)
```

```
In [12]: tup[1].append(3)
```

```
In [13]: tup
```

```
Out[13]: ('foo', [1, 2, 3], True)
```

```
In [14]: tup[1] # bir liste
```

```
Out[14]: [1, 2, 3]
```

Tek elemanlı bir demet aşağıdaki gibi oluşturulur:

```
In [15]: demet1 = ('bar',)
```

```
In [16]: type(demet1)
```

```
Out[16]: tuple
```

```
In [4]: demet1 = ('bar')
```

```
In [5]: type(demet1)
```

```
Out[5]: str
```

Daha uzun demetler oluşturmak için `+` operatörünü kullanarak demetleri birleştirebilirsiniz:

```
In [17]: (4, None, 'foo') + (6, 0) + ('bar',)
```

```
Out[17]: (4, None, 'foo', 6, 0, 'bar')
```

Listelerde olduğu gibi bir demeti bir tamsayı ile çarpmak, demetin o kadar çok kopyasını bir araya getirme etkisine sahiptir:

```
In [18]: ('foo', 'bar') * 4
```

```
Out[18]: ('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```

Nesneleri açma (unpack)

Bir demet-benzeri bir değişken ifadesi atamaya çalışırsanız, Python eşittir işaretinin sağ tarafındaki değeri açmaya çalışır:

```
In [19]: tup = (4.5e-12, 5, 6)
```

```
In [20]: a, b, c = tup
```

```
In [22]: a
```

Out[22]: 4.5e-12

In [23]:
```
b
```

Out[23]: 5

In [24]:
```
c
```

Out[24]: 6

İç içe demetlere (nested tuples) sahip sekanslar bile açılabilir:

In [25]:
```
tup = (1, 2, 3, (4, 5), 6)
```

In [26]:
```
a, b, c, (d, e), f = tup
```

In [27]:
```
a
```

Out[27]: 1

In [28]:
```
b
```

Out[28]: 2

In [29]:
```
c
```

Out[29]: 3

In [30]:
```
d
```

Out[30]: 4

In [31]:
```
e
```

Out[31]: 5

Değişken açmanın yaygın bir kullanımı, demetlerin veya listelerin bir sekansı üzerinde yineleme (iteration) yapmaktır:

In [66]:
```
sequence =[(1, 2, 3), (4, 5, 6), (7, 8, 4)]
# list of tuples
```

In [33]:
```
for i in sequence:
    print(i)
```

```
(1, 2, 3)
(4, 5, 6)
(7, 8, 4)
```

Demet metotları

Bir demetin boyutu ve içeriği değiştirilemediğinden, bu yerleşik veri yapısının çok fazla metodu yoktur. Özellikle yararlı olan (listelerde de bulunur) bir değerin oluşum sayısını sayan `count` işlevidir:

```
In [34]: a = (1,2,2,2,3,4,2)

In [35]: a.count(2)

Out[35]: 4

In [36]: a.index(2)

Out[36]: 1
```

Liste (List)

Demet'lerin aksine, listeler değişken uzunluktadır (variable-length) ve içerikleri yerinde (in-place) değiştirilebilir (yani bu veri yapısı mutable yani değiştirilebilir veri yapısıdır). Bir listeyi, köşeli parantez `[]` kullanarak veya `list` fonksiyonu kullanarak tanımlayabilirsiniz:

```
In [37]: a_list = [2, 3, 7, None]
```

ya da bir demeti listeye çevirebilirsiniz:

```
In [38]: tup = ('foo', 'bar', 'baz')

In [39]: tup

Out[39]: ('foo', 'bar', 'baz')

In [40]: b_list = list(tup)
         b_list

Out[40]: ['foo', 'bar', 'baz']

In [41]: b_list[1] = 'peekaboo'

In [42]: b_list

Out[42]: ['foo', 'peekaboo', 'baz']
```

Listeler ve demetler anlamsal olarak benzerdir (ancak, demetler değiştirilemez) ve birçok fonksiyonda birbirinin yerine kullanılabilir.

Bir listeye eleman ekleme ve eleman çıkarma

`append` metodu ile elemanlar bir listenin sonuna eklenebilir

In [43]:
```
b_list
```

Out[43]:
```
['foo', 'peekaboo', 'baz']
```

In [44]:
```
b_list.append('dwarf')
```

In [45]:
```
b_list
```

Out[45]:
```
['foo', 'peekaboo', 'baz', 'dwarf']
```

`insert` metodu kullanarak listede belirli bir konuma bir eleman ekleyebilirsiniz:

In [46]:
```
b_list.insert(1, 'red')
```

In [47]:
```
b_list
```

Out[47]:
```
['foo', 'red', 'peekaboo', 'baz', 'dwarf']
```

`insert` operasyonunun tersi ise, belirli bir indeksteki bir elemanı kaldıran ve o elemanı döndüren `pop` 'tur:

In [48]:
```
b_list
```

Out[48]:
```
['foo', 'red', 'peekaboo', 'baz', 'dwarf']
```

In [49]:
```
b_list.pop(2)
```

Out[49]:
```
'peekaboo'
```

In [50]:
```
b_list
```

Out[50]:
```
['foo', 'red', 'baz', 'dwarf']
```

Elemanın değeri verilerek kaldırmak isterseniz, `remove` metodunu kullanabilirsiniz:

In [51]:
```
b_list.remove('red')
```

DİKKAT! `remove` metodu ilk oluşumu (occurence) silecektir!

In [52]:
```
newList = ['foo', 'red', 'red', 'red', 'red', 'baz', 'dwarf']
```

In [53]:
```
newList
```

Out[53]:
```
['foo', 'red', 'red', 'red', 'red', 'baz', 'dwarf']
```

In [54]:
```
newList.remove('red')
```

In [55]:
```
newList
```

Out[55]:
```
['foo', 'red', 'red', 'red', 'baz', 'dwarf']
```

`red` değerine sahip tüm elemanlarını silmek için bir `for` döngüsü ve `if` koşulu kullanılabilir:

In [56]:
```
j = []
for i in newList:
    if i not in j:
        j.append(i)
print(j)
```

```
['foo', 'red', 'baz', 'dwarf']
```

Bunlara ek olarak `del` deyimi kullanılarak da listedeki bir eleman, silinecek elemanın konumu verilerek silinebilir:

In [1]:
```
myList = ['foo', 'red', 'peekaboo', 'baz', 'dwarf']
```

In [2]:
```
del myList[3]
```

In [3]:
```
myList
```

Out[3]:
```
['foo', 'red', 'peekaboo', 'dwarf']
```

Listeleri birleştirme (concatenating)

Demetlere benzer şekilde, `+` ile birlikte iki liste eklemek onları birleştirir:

In [57]:
```
[4, None, 'foo'] + [7, 8, (2, 3)]
```

Out[57]:
```
[4, None, 'foo', 7, 8, (2, 3)]
```

Halihazırda tanımlanmış bir listeniz varsa, `extend` metodunu kullanarak ona birden çok öğe ekleyebilirsiniz.

In [58]:
```
x = [4, None, 'foo']
```

In [59]:
```
x.extend([7, 8, (2, 3)])
```

In [60]:
```
x
```

Out[60]:
```
[4, None, 'foo', 7, 8, (2, 3)]
```

Sıralama (sorting)

`sort` fonksiyonunu çağırarak (yeni bir nesne oluşturmadan) bir listeyi yerinde sıralayabilirsiniz:

In [61]:
```
a=[7,2,5,1,3]
```

In [62]:
```
a.sort(reverse = False)
```

In [63]:
```
a
```

Out[63]:
```
[1, 2, 3, 5, 7]
```

Dilimleme (Slicing)

Çoğu sekans tipinin bellir bir kısmını, `[]` operatörüne iletilen `start:stop` 'tan oluşan dilim notasyonunu kullanarak seçebilirsiniz:

In [64]:
```
seq=[7,2,3,7,5,6,0,1]
```

In [65]:
```
seq[2]
```

Out[65]:
```
3
```

1'inci indeks ile 4'üncü indeks arasındaki tüm elemanları seç:

In [66]:
```
seq[1:5]
```

Out[66]:
```
[2, 3, 7, 5]
```

Dilimler ayrıca bir sekans ile atanabilir. Örneğin, `seq` listesindeki 3. ve 4. indeksteki elemanları değiştirelim:

In [67]:
```
seq
```

Out[67]:
```
[7, 2, 3, 7, 5, 6, 0, 1]
```

In [68]:
```
seq=[7,2,3,32,5,6,0,1]

seq[3:5] = [18, 12]
```

In [69]:
```
seq
```

Out[69]:
```
[7, 2, 3, 18, 12, 6, 0, 1]
```

In [70]:
```
seq[3]
```

Out[70]:
```
18
```

In [71]:
```
seq
```

Out[71]:
```
[7, 2, 3, 18, 12, 6, 0, 1]
```

NOT: Dilimleme yapılırken, "start" indeksindeki eleman dahil edilirken, "stop" indeksindeki eleman dahil edilmez,

`start` veya `stop` argümanları yazılmak zorunda değildir. "start" argümanı yazılmazsa, dilimleme listening en başından başlanır; `stop` argümanı yazılmazsa, dilimleme listening en

sonuna kadar devam eder:

```
In [72]: seq
```

```
Out[72]: [7, 2, 3, 18, 12, 6, 0, 1]
```

```
In [73]: seq[:5]
```

```
Out[73]: [7, 2, 3, 18, 12]
```

```
In [74]: seq[3:]
```

```
Out[74]: [18, 12, 6, 0, 1]
```

Negatif indeksler diziyi sona göre dilimler:

```
In [75]: seq
```

```
Out[75]: [7, 2, 3, 18, 12, 6, 0, 1]
```

```
In [76]: seq[-4:]
```

```
Out[76]: [12, 6, 0, 1]
```

```
In [77]: seq[-6:-2]
```

```
Out[77]: [3, 18, 12, 6]
```

İkinci bir iki nokta üst üste (`:`) işaretinden sonra, bir adım (step) da kullanılabilir, örneğin her iki elemanda bir elemanı almak:

```
In [78]: seq
```

```
Out[78]: [7, 2, 3, 18, 12, 6, 0, 1]
```

```
In [79]: seq[::2]
```

```
Out[79]: [7, 3, 12, 0]
```

Bunun akıllıca bir kullanımı, bir listeyi veya demeti tersine çevirmenk için `-1` 'i kullanmaktır:

```
In [80]: seq[::-1]
```

```
Out[80]: [1, 0, 6, 12, 18, 3, 2, 7]
```

Bir Listeyi Kopyalamak (Copying)

Öncelikle Python'un nesneleri nasıl yönettiğini anlamamız gerekiyor. Python'da C gibi değişkenler (variables) yoktur. Python'da her şey nesnedir ve nesneler başka nesnelerden referans alır.

Diyelim ki aşağıdaki gibi bir `a` listesine sahibiz:

In [2]:
```
a = [1, 2, 3]
```

Bu, `a`'nın az önce oluşturduğumuz `[1, 2, 3]` listesine işaret ettiği anlamına gelir.

Diyelim ki, `a`'nın bir kopyasını oluşturup, ismini `b` yapmak istiyoruz.

Python'da bir nesnenin kopyasını oluşturmak için `=` operatörünü kullanırız. Bunun yeni bir nesne oluşturduğunu düşünebilirsiniz; ancak durum öyle değil.

In [3]:
```
b = a
```

In [4]:
```
b
```

Out[4]:
```
[1, 2, 3]
```

Burada gerçekleşecek olan durum, hem `b` hem de `a`'nın aynı nesneyi refere etmesidir. Yani, `b`'de yapacağınız bir değişiklik, aynı şekilde `a`'da da gerçekleşecektir. Bu duruma **referans ile kopyalama (copy by reference)** denir.

`b = a` referans ataması (reference assignment) olarak da bilinir.

Şimdi `b` listesinde bir değişikliğe gidelim ve bu listenin üçüncü elemanını 11 yapalım:

In [5]:
```
b[2] = 11
```

In [6]:
```
b
```

Out[6]:
```
[1, 2, 11]
```

In [7]:
```
a
```

Out[7]:
```
[1, 2, 11]
```

Kolaylıkla görüleceği üzere, `b` listesindeki değişiklik, `a` listesinde de görülmüştür.

Bunun nedeni, yukarıdaki senaryoda, yarattığımız yeni değişkenin aslında `a`'nın aynı bellek adresine (memory address) bağlanmasıdır, bu, yeni bir `b` değişkeni yaratmış gibi görünsek de, gerçek bellek alanı (memory space) almadığı ve yeni değişkenin `a` listesine bir referans olduğu anlamına gelir.

In [8]:
```
id(a)
```

Out[8]:
```
140328885642048
```

In [9]:
```
id(b)
```

Out[9]: 140328885642048

Bu durum `b = a[:]` için farklıdır:

In [1]:
```
a = [1, 2, 3]

b = a[:]
```

In [2]:
```
b
```

Out[2]: [1, 2, 3]

In [3]:
```
id(a)
```

Out[3]: 4362623104

In [4]:
```
id(b)
```

Out[4]: 4362622848

Bu durumu engellemek için, `a` nesnesinin bir kopyasını `b = list(a)` veya `b = a.copy()` veya `b = a[:]` komutlarından birini kullanarak oluşturabilirsiniz.

`[:]` operatörü, bir dizinin bir dilimini döndürür. Listenin bir kısmını dilimleme: yeni bir liste oluşturun ve orijinal listenin bir kısmını bu yeni listeye kopyalayın. İlk indeksi yazmazsanız (örneğin `[:3]`), dilim listenin başında başlar; ikinci indeksi yazmazsanız (örneğin `[3:]`), listenin sonunda durur. `a[:]`'yı çağırarak, baştan başlayıp sonunda bitiren bir dilim alırsınız. Bu, `a`'nın tam bir kopyasıdır.

In [11]:
```
a = [1, 2, 3]
b = list(a)
b.append(11)
print('a:', a)
print('b:', b)

print()

print('id(a):', id(a))
print('id(b):', id(b))
```

```
a: [1, 2, 3]
b: [1, 2, 3, 11]

id(a): 140374560651520
id(b): 140374561226944
```

In [12]:
```
a = [1, 2, 3]
b = a.copy()
b.append(11)
print('a:', a)
```

```
print('b:', b)

print()

print('id(a):', id(a))
print('id(b):', id(b))
```

```
a: [1, 2, 3]
b: [1, 2, 3, 11]

id(a): 140374561205056
id(b): 140374559503424
```

In [13]:
```
a = [1, 2, 3]
b = a[:]
b.append(11)
print('a:', a)
print('b:', b)

print()

print('id(a):', id(a))
print('id(b):', id(b))
```

```
a: [1, 2, 3]
b: [1, 2, 3, 11]

id(a): 140374560834880
id(b): 140374560522688
```

Ancak, iç içe geçmiş listeler (nested lists) kullanırken daha dikkatli olmanız gerekmektedir.

Dış liste (outer list) farklı bir bellek adresinde bulunuyor olsa da, iç liste (inner list) halen `a` listesini referans alacaktır:

In [5]:
```
a = [4, 5, [1, 2]]
b = list(a)
b[2].append(11)
print('a:', a)
print('b:', b)

print()

print('id(a):', id(a))
print('id(b):', id(b))
print()
print('id(a):', id(a[2]))
print('id(b):', id(b[2]))
```

```
a: [4, 5, [1, 2, 11]]
b: [4, 5, [1, 2, 11]]

id(a): 140585562953920
id(b): 140585563640320

id(a): 140585562952064
id(b): 140585562952064
```

In [6]:
```python
a = [4, 5, [1, 2]]
b = a.copy()
b[2].append(11)
print('a:', a)
print('b:', b)

print()

print('id(a):', id(a))
print('id(b):', id(b))
print()
print('id(a):', id(a[2]))
print('id(b):', id(b[2]))
```

```
a: [4, 5, [1, 2, 11]]
b: [4, 5, [1, 2, 11]]

id(a): 140585563776192
id(b): 140585562951808

id(a): 140585563325888
id(b): 140585563325888
```

In [7]:
```python
a = [4, 5, [1, 2]]
b = a[:]
b[2].append(11)
print('a:', a)
print('b:', b)

print()

print('id(a):', id(a))
print('id(b):', id(b))
print()
print('id(a):', id(a[2]))
print('id(b):', id(b[2]))
```

```
a: [4, 5, [1, 2, 11]]
b: [4, 5, [1, 2, 11]]

id(a): 140585563388032
id(b): 140585562543104

id(a): 140585562953728
id(b): 140585562953728
```

`b = list(a)` veya `b = a.copy()` veya `b = a[:]` komutları `a` listesinin sığ kopyasını (shallow copy) oluşturacaktır.

Sığ kopyalamayla ilgili sorun, iç içe geçmiş bir nesneyi (a nested object) kopyalamamasıdır, yani, yukarıdaki örnek için orijinal listede iç içe olan `[1, 2]`'dir.

Tıpkı adı gibi, sığ kopya, özyinelemenin (recursion) derinliklerine inmek yerine yalnızca yüzeyi kopyalar, burada `b` listesinin üçüncü elemanı olan `[1, 2]` listesinin gerçek bir kopyasını oluşturmak yerine orijinal listenin referansını depolayacaktır. Yukarıda yaptığımız gibi bu elemanın bellek adresini yazdırarak doğrulama gerçekleştirebilirsiniz:

```
print('id(a):', id(a[2]))
print('id(b):', id(b[2]))
```

Bu durumu engellemek için `copy` modülünde bulunan `deepcopy()` metodu kullanılmalıdır. Derin bir kopya (a deep copy), bir nesneyi özyinelemeli olarak kopyalamaya gidecektir.

In [11]:
```
import copy

a = [4, 5, [1, 2]]
b = copy.deepcopy(a)
```

In [12]:
```
id(a)
```

Out[12]:
```
140699782538688
```

In [13]:
```
id(b)
```

Out[13]:
```
140699782003776
```

In [14]:
```
a
```

Out[14]:
```
[4, 5, [1, 2]]
```

In [15]:
```
b
```

Out[15]:
```
[4, 5, [1, 2]]
```

In [16]:
```
a[2].append(3)
```

In [17]:
```
a
```

Out[17]:
```
[4, 5, [1, 2, 3]]
```

In [18]:
```
b
```

Out[18]:
```
[4, 5, [1, 2]]
```

In [20]:
```
id(a[2])
```

Out[20]: 140699782010304

In [21]:
```
id(b[2])
```

Out[21]: 140699782198976

In [15]:
```
import copy

a = [4, 5, [1, 2]]
b = copy.deepcopy(a)
b[2].append(11)
print('a:', a)
print('b:', b)

print()

print('id(a):', id(a))
print('id(b):', id(b))
print()
print('id(a):', id(a[2]))
print('id(b):', id(b[2]))
```

```
a: [4, 5, [1, 2]]
b: [4, 5, [1, 2, 11]]

id(a): 140585562392384
id(b): 140585563016448

id(a): 140585563014208
id(b): 140585563014144
```

Burada `b` listesi üzerinde yapılan değişiklikler orijinal listeyi etkilemeyecek ve `a` ile `b`'nin bellek adresleri tamamen farklı olacaktır.

Peki ya demetleri (tuples) kopyalamak?

Bir demet değiştirilemez (immutable) veri yapısı olduğu için, onun tamamen aynı olan başka bir kopyasını yaratmanın gerçekten bir mantığı yoktur. Ancak, demetlerin içerisinde değiştirilebilir (mutable) elemanlar olabileceğinden ve copy/deepcopy/id'nin tahmin ettiğiniz gibi davrandığını unutmayınız.

In [16]:
```
import copy

tup = (1, 2, [])
put = copy.copy(tup)

tup[2].append('hello')

print(put)
#(1, 2, ['hello'])
print(tup)
```

```
#(1, 2, ['hello'])

print()
print('id(put):', id(put))
print('id(tup):', id(tup))
print()
print('id(put):', id(put[2]))
print('id(tup):', id(tup[2]))
```

```
(1, 2, ['hello'])
(1, 2, ['hello'])

id(put): 140328885702528
id(tup): 140328885702528

id(put): 140328885978880
id(tup): 140328885978880
```

NOT: `tup` ve `put` demetlerinin aynı bellek adresine sahip olduğuna dikkat ediniz! Listelerde bu durum farklıydı!! Çünkü demetler değiştirilemez, bu nedenle kopyalanması mantıklı değildir.

In [17]:
```
import copy

tup = (1, 2, [])
put = deepcopy(tup)

tup[2].append('hello')

print(put)
#(1, 2, [])

print(tup)
#(1, 2, ['hello'])

print()
print('id(put):', id(put))
print('id(tup):', id(tup))
print()
print('id(put):', id(put[2]))
print('id(tup):', id(tup[2]))
```

```
(1, 2, [])
(1, 2, ['hello'])

id(put): 140328885700800
id(tup): 140328885218624

id(put): 140328885980864
id(tup): 140328885978368
```

Bir Listeyi Temizlemek (Clear)

Bir listenin temizlenmesi, `list = []` 'ye benzer şekilde `list.clear()` tarafından yapılır.

`list.clear()` metodu Python 3.3 itibariyle Python'a eklendi. Bundan önce herhangi bir Python sürümü için kullanmayı denerseniz, anlamsal olarak eşdeğer olan ve önceki Python sürümleri için de çalışan `del list[:]` yöntemini kullanmalısınız.

In [1]:
```
l = [1,3,4,4]
print(l)

l.clear()
print(l)
```

```
[1, 3, 4, 4]
[]
```

In [2]:
```
l = [1,3,4,4]
print(l)

l = []
print(l)
```

```
[1, 3, 4, 4]
[]
```

In [3]:
```
l = [1,3,4,4]
print(l)

del l[:]
print(l)
```

```
[1, 3, 4, 4]
[]
```

Ancak, `list.clear()`, `list = []` ve `del list[:]` arasında fark olduğunu gözden kaçırmayınız.

In [6]:
```
list1 = [1, 2, 4, 5]
list2 = list1
print(list2)

print('id(list1):', id(list1))
print('id(list2):', id(list2))
```

```
[1, 2, 4, 5]
id(list1): 140336539088320
id(list2): 140336539088320
```

In [7]:
```
list1 = []
print('id(list1):', id(list1))
print()
print(list1)
print(list2)
```

```
id(list1): 140336539613184

[]
[1, 2, 4, 5]
```

`list1` listesini temizlememize rağmen, `list2` listesi silinmedi. Bunun nedeni `list1 = []` komutu yeni boş bir liste yaratır ve bu boş listeyi `list1` ismine atar (ki id'si görüldüğü üzere farklı). Bu, `liste2`'yi hala içindeki elemanlarla eski listeye işaret eder.

`list.clear()` nasıl davrandığını görelim:

In [8]:
```
list1 = [1, 2, 4, 5]
list2 = list1
print(list2)

print('id(list1):', id(list1))
print('id(list2):', id(list2))
```

```
[1, 2, 4, 5]
id(list1): 140336539346176
id(list2): 140336539346176
```

In [9]:
```
list1.clear()
print('id(list1):', id(list1))
print()
print(list1)
print(list2)
```

```
id(list1): 140336539346176

[]
[]
```

`list1.clear()` ise, `list1` ve `list2`'nin her ikisinin de işaret ettiği aynı listeyi temizler.

Aynısı `del list[:]` içinde geçerlidir. Çünkü `list[:]` bir listenin sığ kopyasını oluşturur.

In [11]:
```
list1 = [1, 2, 4, 5]
list2 = list1
print(list2)

print('id(list1):', id(list1))
print('id(list2):', id(list2))
```

```
[1, 2, 4, 5]
id(list1): 140336538667712
id(list2): 140336538667712
```

In [12]:
```
del list1[:]
print('id(list1):', id(list1))
print()
print(list1)
print(list2)
```

```
id(list1): 140336538667712

[]
[]
```

Yerleşik Sekans Fonksiyonları

Python'da aşina olmanız ve her fırsatta kullanmanız gereken bir dizi kullanışlı sekans fonksiyonları vardır.

enumerate

Bir sekans üzerinde yineleme (iteration) yaparken, geçerli elemanın indeksini takip etmek istemek oldukça yaygındır. En kolay yolu:

i=0 for value in collection: #value ile birşeyler yap i+=1

Bu durum çok yaygın olduğundan, Python'un yerleşik bir fonksiyonu vardır: `enumerate`. Bu fonksiyon `(i, value)` demetlerinin bir sekansını döndürür.

for i, value in enumerate(collection): # do something with value

In [81]:
```
myList = [18, 19, 20, 22]

for index, value in enumerate(myList):
    print(index)
    print(value)
    print()
```

```
0
18

1
19

2
20

3
22
```

sorted

`sorted` fonksiyonu, herhangi bir sekansın elemanlarından sıralanmış yeni bir liste döndürür:

In [82]:
```
myTuple = (14, 32, 54, 32)
sorted(myTuple)
```

Out[82]:
```
[14, 32, 32, 54]
```

In [83]:
```
sorted([7, 1, 2, 6, 0, 3, 2])
```

Out[83]:
```
[0, 1, 2, 2, 3, 6, 7]
```

In [84]:
```
sorted('horse race')
```

Out[84]:
```
[' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r', 's']
```

zip

`zip` fonksiyonu, bir demet listesi oluşturmak için bir dizi listenin, demetin veya diğer sekansların elemanlarını "eşleştirir":

In [85]:
```
seq1 = ['foo', 'bar', 'baz']
seq2 = ['one', 'two', 'three']
```

In [86]:
```
zipped = zip(seq1, seq2)
```

In [87]:
```
zipped
```

Out[87]:
```
<zip at 0x7fab17d19380>
```

Bu bir nesnedir. Burada verilen sayı ise bu nesnenin hafızada (memory) tuttuğu yerin lokasyonudur.

In [88]:
```
list(zipped)
```

Out[88]:
```
[('foo', 'one'), ('bar', 'two'), ('baz', 'three')]
```

`zip`, rastgele sayıda sekans alabilir ve ürettiği nesnenin eleman sayısı en kısa sekansa göre belirlenir:

In [89]:
```
seq1 = ['foo', 'bar', 'baz']
seq2 = ['one', 'two', 'three']
seq3 = [False, True]
```

In [90]:
```
list(zip(seq1, seq2, seq3))
```

Out[90]:
```
[('foo', 'one', False), ('bar', 'two', True)]
```

In [91]:
```
for i in zip(seq1, seq2, seq3):
    print(i)
```

```
('foo', 'one', False)
('bar', 'two', True)
```

`zip`'in çok yaygın bir kullanımı, aynı zamanda `enumerate` ile birleştirilmiş birden çok sekans üzerinde aynı anda yinelemektir (iteration):

In [92]:
```
seq1 = ['foo', 'bar', 'baz']
seq2 = ['one', 'two', 'three']
list(zip(seq1, seq2))
```

Out[92]:
```
[('foo', 'one'), ('bar', 'two'), ('baz', 'three')]
```

In [93]:
```
for i, (a, b) in enumerate(zip(seq1, seq2)):
    print(i)
    print(a)
```

```
    print(b)
    print()
```

```
0
foo
one

1
bar
two

2
baz
three
```

reversed

`reversed` fonksiyonu, ters bir sekansın elemanları üzerinde ters sırada yineleme gerçekleştirir:

In [94]:
```
list(reversed(range(10)))
```

Out[94]:
```
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

In [28]:
```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# returns True if number is even
def check_even(number):
    if number % 2 == 0:
          return True
    return False

# Extract elements from the numbers list for which check_even() returns True
even_numbers_iterator = filter(check_even, numbers)
```

In [29]:
```
even_numbers_iterator
```

Out[29]:
```
<filter at 0x7fc272d17910>
```

Bu bir filter nesnesidir. İçeriğini görmek için `list` veya `tuple` fonksiyonuna sarmayalabilirsiniz:

In [30]:
```
# converting to list
even_numbers = list(even_numbers_iterator)

print(even_numbers)
```

```
[2, 4, 6, 8, 10]
```

Bu işlemi gerçekleştirmek için bir lambda fonksiyonu da kullanılabilir:

In [32]:
```
numbers = [1, 2, 3, 4, 5, 6, 7]
```

```
# the lambda function returns True for even numbers
even_numbers_iterator = filter(lambda x: (x%2 == 0), numbers)

# converting to list
even_numbers = list(even_numbers_iterator)

print(even_numbers)
```

```
[2, 4, 6]
```

Bir başka örnek:

In [31]:
```
letters = ['a', 'b', 'd', 'e', 'i', 'j', 'o']

# a function that returns True if letter is vowel
def filter_vowels(letter):
    vowels = ['a', 'e', 'i', 'o', 'u']
    return True if letter in vowels else False

filtered_vowels = filter(filter_vowels, letters)

# converting to tuple
vowels = tuple(filtered_vowels)
print(vowels)
```

```
('a', 'e', 'i', 'o')
```

In [34]:
```
letters = ['a', 'b', 'd', 'e', 'i', 'j', 'o']

filtered_vowels = filter(lambda letter: (letter in ['a', 'e', 'i', 'o', 'u']), lett

# converting to tuple
vowels = tuple(filtered_vowels)
print(vowels)
```

```
('a', 'e', 'i', 'o')
```

map

Bazen, yeni bir yinelenebilir (iterable) oluşturmak için yinelenebilir bir girdinin tüm elemanlarında aynı operasyonu gerçekleştirmeniz gereken durumlarla karşılaşabilirsiniz. Bu soruna en hızlı ve en yaygın yaklaşım, bir Python for döngüsü kullanmaktır. Ancak, bu sorunu açık bir döngü olmadan da map() kullanarak çözebilirsiniz. Söz dizimi aşağıdaki gibidir:

```
map(function, iterable[, iterable1, iterable2,..., iterableN])
```

In [35]:
```
numbers = [1, 2, 3, 4, 5]
squared = []

for num in numbers:
    squared.append(num ** 2)


squared
```

Out[35]:
```
[1, 4, 9, 16, 25]
```

Bu döngüyü `numbers` listesi üzerinde çalıştırdığınızda, kare değerlerin bir listesini alırsınız. for döngüsü sayılar üzerinde yinelenir ve her değere bir güç işlemi uygular. Son olarak, elde edilen değerleri `squared` listesinde saklar.

Aynı sonucu, açık bir döngü kullanmadan map() kullanarak da elde edebilirsiniz. Yukarıdaki örneğin aşağıdaki yeniden uygulamasına bir göz atınız:

In [37]:
```
def square(number):
    return number ** 2

numbers = [1, 2, 3, 4, 5]

squared = map(square, numbers)
```

In [38]:
```
squared
```

Out[38]:
```
<map at 0x7fc272d30790>
```

In [39]:
```
list(squared)
```

Out[39]:
```
[1, 4, 9, 16, 25]
```

`map()` fonksiyonu, C ile yazıldığından ve yüksek düzeyde optimize edildiğinden, dahili döngüsü normal bir Python for döngüsünden daha verimli olabilir. Bu, map() kullanmanın bir avantajıdır.

`map()` kullanmanın ikinci bir avantajı, bellek tüketimiyle ilgilidir. Bir for döngüsü ile, tüm listeyi sisteminizin belleğinde saklamanız gerekir. map() ile, isteğe bağlı olarak öğeler alırsınız ve belirli bir zamanda sisteminizin belleğinde yalnızca bir öğe bulunur.

Peki, aşağıda dizgilerden oluşan bir listenin her elemanını tamsayıya çevirelim:

In [40]:
```
str_nums = ["4", "8", "6", "5", "3", "2", "8", "9", "2", "5"]
```

In [41]:
```
int(str_nums) # bu işe yaramaz.
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/var/folders/z_/r0991b310gjd66vy5k26g4800000gn/T/ipykernel_5479/3746212245.py in <mo
dule>
----> 1 int(str_nums)

TypeError: int() argument must be a string, a bytes-like object or a number, not 'li
st'
```

In [42]:
```
int_nums = map(int, str_nums)
int_nums
```

Out[42]:
```
<map at 0x7fc272d736a0>
```

In [43]:
```
list(int_nums)
```

Out[43]:
```
[4, 8, 6, 5, 3, 2, 8, 9, 2, 5]
```

Dizgi manipülasyonunda oldukça yaygın bir yaklaşım, belirli bir dizgiyi yeni bir dizgiye dönüştürmek için `str` sınıfının bazı metotlarını kullanmaktır.

Dizgilerin yinelebilirleriyle (iterables of strings) uğraşıyorsanız ve her dizgiye aynı dönüşümü (transformation) uygulamanız gerekiyorsa, çeşitli dizgi metotları ile birlikte `map()` fonksiyonunu kullanabilirsiniz:

In [45]:
```
string_it = ["processing", "strings", "with", "map"]

print(list(map(str.capitalize, string_it)))

print(list(map(str.upper, string_it)))

print(list(map(str.lower, string_it)))
```

```
['Processing', 'Strings', 'With', 'Map']
['PROCESSING', 'STRINGS', 'WITH', 'MAP']
['processing', 'strings', 'with', 'map']
```

In [46]:
```
with_spaces = ["processing ", "  strings", "with   ", " map   "]

list(map(str.strip, with_spaces))
```

Out[46]:
```
['processing', 'strings', 'with', 'map']
```

Tabii ki, lambda fonksiyonunu da kullanabilirsiniz. Örneğin, Celsius hava sıcaklığını Fahrenheit'a çevirelim:

In [50]:
```
celsius_temps = [100, 40, 80]

list(map(lambda C: 9 / 5 * C + 32, celsius_temps))
```

Out[50]:
```
[212.0, 104.0, 176.0]
```

In [51]:
```
fahrenheit_temps = [212, 104, 176]

list(map(lambda F: (F - 32) * 5 / 9, fahrenheit_temps))
```

Out[51]:
```
[100.0, 40.0, 80.0]
```

`map()` fonksiyonunu `filter` fonksiyonu ile birlikte de kullanabilirsiniz:

In [52]:
```
import math

math.sqrt(-16)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/var/folders/z_/r0991b310gjd66vy5k26g4800000gn/T/ipykernel_5479/3030156208.py in <mo
dule>
      1 import math
      2
----> 3 math.sqrt(-16)

ValueError: math domain error
```

In [53]:
```
numbers = [25, 9, 81, -16, 0]

list(map(math.sqrt, filter(lambda num: num>= 0, numbers)))
```

Out[53]:
```
[5.0, 3.0, 9.0, 0.0]
```

Sözlük (Dictionary)

Bir sözlük (`dict`) muhtemelen en önemli yerleşik Python veri yapısıdır. Anahtarlar (key) ve Değerlerden (values) oluşur. Bu anahtarlar ve değerler de Python nesneleridir.

Sözlüğe aynı zamanda anahtar/değer çiftleri koleksiyonu denir.

Sözlük oluşturmaya yönelik bir yaklaşım, küme parantezleri `{}` ve anahtarları ve değerleri ayırmak için iki nokta üst üste (`:`) kullanmaktır:

In [150…
```
empty_dict = {} # Bu bir boş sözlüktür.
empty_dict = dict()
empty_dict
```

Out[150…
```
{}
```

In [151…
```
type(empty_dict)
```

Out[151…
```
dict
```

In [152…
```
d1 = {'a' : 'some value', 'b' : [1, 2, 3, 4]}
d1
```

Out[152…
```
{'a': 'some value', 'b': [1, 2, 3, 4]}
```

NOT: Dikkat ederseniz burada değerler istenildiğinde tek bir sabit değer veya bir veri yapısı olabilir!!

Yukarıdaki sözlükte, `'a'` anahtarı bir dizgi (string), `'b'` anahtarı bir liste almıştır.

Bir liste veya demetin elemanlarına erişimle aynı sözdizimini kullanarak, bir sözlüğün elemanlarına erişebilir, eleman ekleyebilir veya elemanları ayarlayabilirsiniz:

In [153…
```
# anahtar 7, değer 'an integer'
d1[7] = 'murat'
```

In [154…
```
d1
```

Out[154…
```
{'a': 'some value', 'b': [1, 2, 3, 4], 7: 'murat'}
```

In [155…
```
d1[7]
```

Out[155…
```
'murat'
```

Sözlük anahtarları benzersiz (unique) olmalıdır, yani aynı değere sahip iki farklı anahtar ekleyemezsiniz!

In [158…
```
d1 = {'a' : 'some value', 'b' : [1, 2, 3, 4], 'a': 3}
```

In [159…
```
d1
```

Out[159…
```
{'a': 3, 'b': [1, 2, 3, 4]}
```

In [160…
```
d1['a']='murat'
```

In [161…
```
d1
```

Out[161…
```
{'a': 'murat', 'b': [1, 2, 3, 4]}
```

Kolaylıkla görülebileceği üzere, son eklenen `'a'` anahtarı, ilkinin üzerine yazmıştır.

NOT: Bir sözlük, tekrarlanmış (duplicated) anahtarlar oluşturmaya izin vermese de, aynı anahtarı birden çok kez kullanırsanız sizi uyarmaz. Bu nedenle, beklenmedik davranışlara neden olmamak için ekstra dikkatli olmanız gerekir.

Sözlükteki eğerleri `del` anahtar sözcüğünü (keyword) veya `pop` metodunu kullanarak silebilirsiniz:

In [163…
```
d1 = {'a' : 'some value', 'b' : [1, 2, 3, 4]}
d1
```

Out[163…
```
{'a': 'some value', 'b': [1, 2, 3, 4]}
```

In [164…
```
d1[7] = 'murat'
```

In [165…
```
d1
```

Out[165…
```
{'a': 'some value', 'b': [1, 2, 3, 4], 7: 'murat'}
```

In [166…
```
del d1[7]
```

In [167…
```
d1
```

Out[167…
```
{'a': 'some value', 'b': [1, 2, 3, 4]}
```

In [168…
```
d1[5] = 'some value'
```

In [169…
```
d1
```

Out[169…
```
{'a': 'some value', 'b': [1, 2, 3, 4], 5: 'some value'}
```

In [170…
```
d1['dummy'] = 'another value'
```

In [171…
```
d1
```

Out[171…
```
{'a': 'some value',
 'b': [1, 2, 3, 4],
 5: 'some value',
 'dummy': 'another value'}
```

In [172…
```
# sözlük içerisinde sözlük olabilir
d1 = {'a': 3, 'b': {'isim': 'murat', 'yas': 32, 'cinsiyet': 'erkek'} , 5: 'some val
d1
```

Out[172…
```
{'a': 3,
 'b': {'isim': 'murat', 'yas': 32, 'cinsiyet': 'erkek'},
 5: 'some value',
 'dummy': 'another value'}
```

`5` anahtar sözcüğüne sahip elemanı silelim:

In [173…
```
del d1[5]
```

In [174…
```
d1
```

Out[174…
```
{'a': 3,
 'b': {'isim': 'murat', 'yas': 32, 'cinsiyet': 'erkek'},
 'dummy': 'another value'}
```

`'dummy'` anahtar sözcüğüne sahip elemanı silelim:

In [175…
```
d1.pop('dummy')
```

Out[175…
```
'another value'
```

In [176…
```
d1
```

Out[176…
```
{'a': 3, 'b': {'isim': 'murat', 'yas': 32, 'cinsiyet': 'erkek'}}
```

Bir sözlük içerisindeki sözlüklere erişim köşeli parentezler ile yapılır:

In [177…
```
d1['b']['yas']
```

Out[177…
```
32
```

Sözlüğün bir anahtarının (key) değeri bir mutable veri yapısıyla, o veri yapısına nokta notasyonu (dot notation) ile müdahele edebilirsiniz:

In [179…
```
d1 = {'a': 3, 'b': [3, 5, 7, True] , 5: 'some value', 'dummy': 'another value'}
d1
```

Out[179…
```
{'a': 3, 'b': [3, 5, 7, True], 5: 'some value', 'dummy': 'another value'}
```

In [180…
```
d1['b'].append(19)
```

In [181…
```
d1
```

Out[181…
```
{'a': 3, 'b': [3, 5, 7, True, 19], 5: 'some value', 'dummy': 'another value'}
```

`keys` ve `values` metodları, sırasıyla bir sözlüğün anahtarlarının ve değerlerinin yineleyicilerini (iterators) verir.

In [23]:
```
d1
```

Out[23]:
```
{'a': 3, 'b': {'isim': 'murat', 'yas': 32, 'cinsiyet': 'erkek'}}
```

In [49]:
```
d1['b'].keys()
```

Out[49]:
```
dict_keys(['isim', 'yas', 'cinsiyet'])
```

In [25]:
```
d1.values()
```

Out[25]:
```
dict_values([3, {'isim': 'murat', 'yas': 32, 'cinsiyet': 'erkek'}])
```

In [26]:
```
d1.items()
```

Out[26]:
```
dict_items([('a', 3), ('b', {'isim': 'murat', 'yas': 32, 'cinsiyet': 'erkek'})])
```

`keys()` ve `values()` metotları sözlükler üzerinde döngü kurmak için kullanılabilir:

In [53]:
```
a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
for key in a_dict.keys():
    print(key)
```

```
color
fruit
pet
```

In [5]:
```
a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
for key in a_dict.values():
    print(key)
```

```
blue
apple
dog
```

`items()` metodunun demetlerin bir listesini (a list of tuples) dönderdiğine dikkat ediniz:

In [146…
```
a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
for i in a_dict.items():
    print(i)
    print()
```

```
('color', 'blue')

('fruit', 'apple')

('pet', 'dog')
```

In [148…
```
# Unpacking tuples konsepti kullanılırsa
a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
for key, value in a_dict.items():
    print(key)
    print(value)
    print()
```

```
color
blue

fruit
apple

pet
dog
```

ANCAK, sadece sözlük üzerinde döngü kurmak, anahtarları döndürür!

In [8]:
```
a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
for i in a_dict:
    print(i)
```

```
color
fruit
pet
```

`update` metodunu kullanarak bir sözlüğü diğeriyle birleştirebilirsiniz:

In [57]:
```
d1
```

Out[57]:
```
{'a': 3, 'b': [3, 5, 7, True, 19], 5: 'some value', 'dummy': 'another value'}
```

In [58]:
```
d1.update({'b' : 'foo', 'c' : 12})
```

In [59]:
```
d1
```

Out[59]:
```
{'a': 3, 'b': 'foo', 5: 'some value', 'dummy': 'another value', 'c': 12}
```

`update` metodu değişiklikleri yerinde (in-place) gerçekleştirir, bu nedenle güncellemeye iletilen verilerdeki mevcut anahtarların eski değerleri görmezden gelinir. Bunun sebebi sözlükteki her anahtar benzersiz (unique) olmalıdır. Yani, aynı değere sahip iki anahtarınız

olamaz. Aynı anahtarı tekrar kullanmaya çalışmak, saklanan önceki değerin üzerine yazacaktır.

```
In [118…  d1
Out[118…  {'a': 3, 'b': 'foo', 'c': 12}
In [119…  d1.update({'b' : 'murat'})
In [120…  d1
Out[120…  {'a': 3, 'b': 'murat', 'c': 12}
```

Listelerde ve demetlerde `sorted` sekans fonksiyonundan bahsetmiştik.

Bu veri yapılarında benzer şekilde `max` ve `min` fonksiyonları da kullanılır

```
In [141…  a = (8, 9, 23, 1, 5, 7)
In [142…  sorted(a)
Out[142…  [1, 5, 7, 8, 9, 23]
In [143…  sorted(a, reverse = True)
Out[143…  [23, 9, 8, 7, 5, 1]
In [144…  max(a)
Out[144…  23
In [145…  min(a)
Out[145…  1
```

Bir Python sözlüğüne `max` uygulamak size alfanümerik olarak en büyük anahtarı verecektir. Benzer durum `min` fonksiyonu için de geçerlidir.

```
In [4]:  birth_year = {"Ben": 1945, "Alex": 2000, "Oliver": 1995}
In [5]:  max(birth_year)
Out[5]:  'Oliver'
In [6]:  min(birth_year)
Out[6]:  'Alex'
```

Bir sözlükteki en büyük değere sahip anahtarı bulmak için, `max` metodunun `key` parametresini (`sort` metodunda olduğu gibi) lambda fonksiyonuyla birlikte kullanınız:

In [7]:
```
max(birth_year, key= lambda key_: birth_year[key_])
```

Out[7]:
```
'Alex'
```

In [8]:
```
min(birth_year, key= lambda key_: birth_year[key_])
```

Out[8]:
```
'Ben'
```

Bir sözlüğü sıralamak

In [133…
```
people = {3: "Jim", 2: "Jack", 4: "Jane", 1: "Jill"}
people
```

Out[133…
```
{3: 'Jim', 2: 'Jack', 4: 'Jane', 1: 'Jill'}
```

In [134…
```
people.items()
```

Out[134…
```
dict_items([(3, 'Jim'), (2, 'Jack'), (4, 'Jane'), (1, 'Jill')])
```

In [135…
```
# Sort by key
sorted(people.items())
```

Out[135…
```
[(1, 'Jill'), (2, 'Jack'), (3, 'Jim'), (4, 'Jane')]
```

In [136…
```
dict(sorted(people.items()))
```

Out[136…
```
{1: 'Jill', 2: 'Jack', 3: 'Jim', 4: 'Jane'}
```

In [137…
```
sorted(people.items(), reverse = True)
```

Out[137…
```
[(4, 'Jane'), (3, 'Jim'), (2, 'Jack'), (1, 'Jill')]
```

In [138…
```
# Sort by value
sorted(people.items(), key=lambda item: item[1])
```

Out[138…
```
[(2, 'Jack'), (4, 'Jane'), (1, 'Jill'), (3, 'Jim')]
```

In [139…
```
dict(sorted(people.items(), key=lambda item: item[1], reverse = False))
```

Out[139…
```
{2: 'Jack', 4: 'Jane', 1: 'Jill', 3: 'Jim'}
```

In [140…
```
# Sort by value in descending order
dict(sorted(people.items(), key=lambda item: item[1], reverse = True))
```

Out[140…
```
{3: 'Jim', 1: 'Jill', 4: 'Jane', 2: 'Jack'}
```

get metodu

`get()` metodu, belirtilen anahtara sahip öğenin değerini döndürür.

```
dictionary.get(keyname, value)
```

- `keyname` (required): The keyname of the item you want to return the value from
- `value` (optional): A value to return if the specified key does not exist. Default value `None`.

In [88]:
```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

car.get("model")
```

Out[88]:
```
'Mustang'
```

In [89]:
```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

car.get("price")
```

Yukarıdaki kod çalıştırıldığında JupyterLab hiç bir çıktı döndermeyecektir. Yani `type`'ı `None` olacaktır.

Ancak `get` metodunun `value` argümanını kullanarak var olmayan anahtarları (keys) bir yerlerde toplayabiliriz:

In [90]:
```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

car.get("price", "Does not exist")
```

Out[90]:
```
'Does not exist'
```

Sözlük anahtarı üzerindeki kısıtlamalar

Python'da hemen hemen her tür değer bir sözlük anahtarı olarak kullanılabilir. Tamsayı, kayan-nokta ve Boole nesneleri anahtar olarak kullanılabilir:

In [5]:
```
foo = {42: 'aaa', 2.78: 'bbb', True: 'ccc'}
foo
```

Out[5]:
```
{42: 'aaa', 2.78: 'bbb', True: 'ccc'}
```

Veri tipleri (datatypes) ve fonksiyonlar (functions) gibi yerleşik nesneleri bile kullanabilirsiniz:

In [4]:
```
d = {int: 1, float: 2, bool: 3}
d
```

Out[4]:
```
{int: 1, float: 2, bool: 3}
```

Ancak, sözlük anahtarlarının uyması gereken birkaç kısıtlama vardır.

İlk olarak, belirli bir anahtarın sözlükte yalnızca bir kez görülebileceğinden yukarıda bahsetmistik.

İkinci olarak, bir sözlük anahtarı değiştirilemez (immutable) bir tipte olmalıdır. Tamsayı (integer), kayan-nokta (float), dizgi (string) ve Boole (Boolean) gibi aşina olduğunuz değiştirilemez veri tiplerinin birçoğunun sözlük anahtarları olarak işlev gördüğü örnekler vardır.

Bir demet aynı zamanda bir sözlük anahtarı olabilir, çünkü demetler değiştirilemezdir:

In [7]:
```
d = {(1, 1): 'a', (1, 2): 'b', (2, 1): 'c', (2, 2): 'd'}
d
```

Out[7]:
```
{(1, 1): 'a', (1, 2): 'b', (2, 1): 'c', (2, 2): 'd'}
```

In [8]:
```
d[(1,1)]
```

Out[8]:
```
'a'
```

In [9]:
```
d[(2,1)]
```

Out[9]:
```
'c'
```

Ancak, ne liste ne de başka bir sözlük sözlük anahtarı işlevi göremez, çünkü listeler ve sözlükler değiştirebilirdir (mutable):

In [11]:
```
d = {[1, 1]: 'a', [1, 2]: 'b', [2, 1]: 'c', [2, 2]: 'd'}
d
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/var/folders/z_/r0991b310gjd66vy5k26g4800000gn/T/ipykernel_6343/184572516.py in <mod
ule>
----> 1 d = {[1, 1]: 'a', [1, 2]: 'b', [2, 1]: 'c', [2, 2]: 'd'}
      2 d

TypeError: unhashable type: 'list'
```

Teknik Not: Hata mesajı neden "**hash**lenemez (unhashable)" diyor?

Teknik olarak, bir nesnenin sözlük anahtarı olarak kullanılabilmesi için değiştirilemez (immutable) olması gerektiğini söylemek pek doğru değildir.

Daha doğrusu, bir nesne **hash**lenebilir (hashable) olmalıdır, bu da bu nesnenin bir hash fonksiyonuna gönderilebileceği anlamına gelir.

Bir **hash** fonksiyonu (hash function), rastgele boyuttaki verileri alır ve bunları, tablo arama (lookup) ve karşılaştırma (comparison) için kullanılan, **hash** değeri (hash value) (veya sadece **hash** de denir) adı verilen nispeten daha basit sabit-boyutlu (fixed-size) bir değere eşler (mapping).

Python'un yerleşik `hash()` fonksiyonu, bir nesnenin hash değerini döndürür ve aşağıdaki gibi **hash**lenebilir (hashable) olmayan bir nesne için bir istisna (exception) oluşturur:

In [132…
```
hash('foo')
```

Out[132…
```
8028166870896770072
```

`'foo'` nesnesinin hash değeri elde edilebilir çünkü dizgiler (string) değiştirilemez (immutable) veri yapısıdır.

In [13]:
```
hash([1, 2, 3])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/var/folders/z_/r0991b310gjd66vy5k26g4800000gn/T/ipykernel_6343/2492717709.py in <mo
dule>
----> 1 hash([1, 2, 3])

TypeError: unhashable type: 'list'
```

Sözlüklerde kopyalamaya dikkat edilmelidir:

In [12]:
```
mydict1 = {'value': 11}

mydict2 = mydict1
```

In [13]:
```
print(id(mydict1))
```

```
4362459264
```

In [14]:
```
print(id(mydict2))
```

```
4362459264
```

Diyelim ki `dict2` sözlüğünde `'value'` anahtarının değerini değiştirelim ve `22` yapalım:

In [15]:
```
mydict2['value'] = 22
```

In [16]:
```
mydict2
```

Out[16]:
```
{'value': 22}
```

In [17]:
```
mydict1
```

```
Out[17]: {'value': 22}
```

Görüldüğü üzere `mydict1` ve `mydict2` aynı nesneye point ettiği için `mydict1` de değişmiştir. Diyelim ki `mydict3` isminde diğer bir sözlüğümüz olsun:

```
In [18]: mydict3 = {'value': 33}

In [19]: mydict2 = mydict3

In [20]: mydict2
Out[20]: {'value': 33}

In [21]: mydict3
Out[21]: {'value': 33}

In [22]: mydict1
Out[22]: {'value': 22}
```

Artık `mydict2` sözlüğü yeni bir sözlüğe point etmektedir. Şimdi de diyelim ki `mydict1 = mydict2` olsun

```
In [23]: mydict1 = mydict2

In [24]: mydict1
Out[24]: {'value': 33}

In [25]: mydict2
Out[25]: {'value': 33}

In [26]: mydict3
Out[26]: {'value': 33}
```

Peki bu durumda `{'value': 22}` sözlüğüne ne olur? Bu durumda, bu sözlüğe herhangi bir şey değişken point etmemektedir. Bu durumda bu sözlüğe erişilemez.

Bu durumda Python, bu sözlüğü "Garbage Collection" denilen bir süreç ile silecektir.

Küme (Set)

Bir küme, sıralanmamış (unordered) benzersiz (unique) elemanlar topluluğudur. Kümeleri sözlükler gibi düşünebilirsiniz, ancak yalnızca anahtarlar vardır, değerler yoktur.

Bir küme iki şekilde oluşturulabilir: `set` fonksiyonu aracılığıyla veya küme parantezleri içeren bir küme değişmezi (set literal) aracılığıyla:

In [92]:
```
set([2, 2, 2, 1, 3, 3])
# DİKKAT! Tekrar eden elemanlar var
# ama küme benzersiz elemanlardan oluşur.
```

Out[92]:
```
{1, 2, 3}
```

In [93]:
```
a={9, 10, 2, 2, 2, 1, 3, 3, 105 }
a
```

Out[93]:
```
{1, 2, 3, 9, 10, 105}
```

In [123…
```
{2,2,2,1,3,3}
```

Out[123…
```
{1, 2, 3}
```

Kümeler, birleşim, kesişim, fark ve simetrik fark gibi matematiksel küme işlemlerini destekler. Bu iki örnek kümesini göz önünde bulundurun:

In [124…
```
a={1,2,3,4,5}
a
```

Out[124…
```
{1, 2, 3, 4, 5}
```

In [125…
```
b={3,4,5,6,7,8}
b
```

Out[125…
```
{3, 4, 5, 6, 7, 8}
```

Bu iki kümenin birleşimi, her iki kümede de meydana gelen farklı elemanların kümesidir. Bu, `union` metoduyla veya `|` ikili operatörü ile gerçekleştirilir:

In [126…
```
a.union(b)
```

Out[126…
```
{1, 2, 3, 4, 5, 6, 7, 8}
```

In [127…
```
a | b
```

Out[127…
```
{1, 2, 3, 4, 5, 6, 7, 8}
```

Kesişme, her iki kümede de meydana gelen elemanları içerir. `&` operatörü veya `intersection` metoduyla kullanılabilir:

In [128…
```
a.intersection(b)
```

Out[128…
```
{3, 4, 5}
```

In [129…
```
a & b
```

Out[129…
```
{3, 4, 5}
```

Diğer küme yöntemlerini aşağıda görebilirsiniz:

List, Set, and Dict Comprehensions

"List comprehension", en sevilen Python dili özelliklerinden biridir. Bir koleksiyonun elemanlarını filtreleyerek, filtreden geçen öğeleri tek bir kısa ifadede dönüştürerek kısa ve öz bir şekilde yeni bir liste oluşturmanıza olanak tanırlar. Aldığı en basit form aşağıdaki gibidir:

```
[expr for val in collection if condition]
```
Bu, aşağıdaki for döngüsüne eşdeğerdir:

```
result = []
for val in collection:
    if condition:
        result.append(expr)
```
Burada, if deyimi ile filtreleme yapılma zorunluluğu yoktur

Örneğin, bir dizgi (string) listesi verildiğinde, uzunluğu 2 veya daha fazla olan dizgileri filtreleyebilir ve filtrelenen dizgileri şu şekilde büyük harfe dönüştürebiliriz:

In [9]:
```
strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
strings
```

Out[9]:
```
['a', 'as', 'bat', 'car', 'dove', 'python']
```

In [12]:
```
[x.upper() for x in strings if len(x) > 2]
```

Out[12]:
```
['BAT', 'CAR', 'DOVE', 'PYTHON']
```

In [13]:
```
capital_ = []
for x in strings:
    if len(x) >2:
        x_big = x.upper()
        capital_.append(x_big)

capital_
```

Out[13]:
```
['BAT', 'CAR', 'DOVE', 'PYTHON']
```

"Set (küme) comprehension" ve "Sözlük (dict) comprehension", bunun doğal bir uzantıdır, bir liste yerine, bir küme veya sözlük döndürür. Bir "dict comprehension" aşağıdaki gibi yazılır:

```
dict_comp = {key-expr : value-expr for value in collection if condition}
```
Köşeli parantez yerine, küme parantezleri kullanırsak, "Set (küme) comprehension" elde ederiz:

```
set_comp = {expr for value in collection if condition}
```

Örneğin, aşağıdaki dizgilerin bir listesindeki elemanların uzunluklarını (length) bir küme (set) içerisinde yazdıralım:

In [14]:
```
strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
strings
```

Out[14]:
```
['a', 'as', 'bat', 'car', 'dove', 'python']
```

In [15]:
```
unique_lengths = {len(x) for x in strings}
unique_lengths
```

Out[15]:
```
{1, 2, 3, 4, 6}
```

DİKKAT: Kümelerin özelliği nedeniyle tekrarlı öğeler silinmiştir.

Benzer şekilde, bunu bir "Sözlük (dict) comprehension" olarak yazabiliriz:

In [20]:
```
loc_mapping = {val.upper() : len(val) for _, val in enumerate(strings)}
loc_mapping
```

Out[20]:
```
{'A': 1, 'AS': 2, 'BAT': 3, 'CAR': 3, 'DOVE': 4, 'PYTHON': 6}
```

Burada `enumerate` fonksiyonunun kullanımını hatırlayalım.

EKSTRA NOT: Burada, `index` argümanını kullanmadık. Kullanmadığımız argüman için `_` kullanabiliriz:

In [18]:
```
{val : len(val) for _, val in enumerate(strings)}
```

Out[18]:
```
{'a': 1, 'as': 2, 'bat': 3, 'car': 3, 'dove': 4, 'python': 6}
```

Sözlükleri Bir Fonksiyon çıktısı olarak kullanmak

In [118…
```
def hesap_makine(a, b):
    return a+b, a-b, a/b, a*b
```

In [119…
```
hesap_makine(a = 3, b = 2)
```

Out[119…
```
(5, 1, 1.5, 6)
```

In [120…
```
a, b, c, d  = hesap_makine(a = 3, b = 2)
```

In [121…
```
a
```

Out[121…
```
5
```

In [122…
```
b
```

Out[122…
```
1
```

In [123…
```
c
```

Out[123…
```
1.5
```

In [124…
```
d
```

Out[124…
```
6
```

In [125…
```
def hesap_makine(a, b):
    return {'toplam': a+b, 'fark': a-b, 'bölüm': a/b, 'çarpım':a*b}
```

In [126…
```
res_ = hesap_makine(a = 3, b = 2)
```

In [127…
```
res_['toplam']
```

Out[127…
```
5
```

In [128…
```
res_['fark']
```

Out[128…
```
1
```

In [129…
```
res_['bölüm']
```

Out[129…
```
1.5
```

In [ ]:
```
res_['çarpım']
```

Hangisini kullanmalıyım?

- Oluşturulacak sekansta sıralamayı takip etmeniz gerekiyorsa, bir liste (list) veya demet (tuple) kullanınız.
- Yalnızca benzersiz değerleri (unique values) takip etmek istiyorsanız ve sırayı (order) umursamıyorsanız, bir küme (set) kullanınız.
- Nesnenizi tanımladıktan sonra değişiklik yapmanız gerekmiyorsa, yerden tasarruf etmek için bir demet (tuple) kullanın ve hiçbir şeyin verilerinizin üzerine yazmayacağından emin olunuz.
- Anahtar/değer (key/value) çiftlerinde yapılandırılmış (structured) verileri izlemeniz ve değiştirmeniz gerekiyorsa, bir sözlük (dictionary) kullanınız.

Değiştirilebilir (Mutable) / Değiştirilemez (Immutable) Veri Yapıları - ÖZET