STRİNGLER

immutable yapısı var karakterleri değiştirilemez.
indexingde daha büyük değer yazarsak hata alırız.
slicingde hata almaz olabilecek max değeri alır.

MANTIKSAL OPERATÖRLER

& and gibi çalışır ama short circuit kullanmaz.
| or gibi çalışır ama short circuit kullanmaz.

KOŞULU TEK SATIRDA YAZMA

```
if cevap == "y":
    x = 2
else:
    x = 0
x = 2 if cevap=="y" else 0
```

DÖNGÜLER FOR & WHİLE

```
s = "hey"
for c in s:
    print(c)
```

for'da test mekanizması yoktur for'u while kullanarak yazabiliriz, ama while'ı for kullanarak yazamayız.

```
n = len(s)
index = 0

while index < n:
        print(s[index])
        index += 1
```

while bir koşul bildirir ve kaç kere iterasyon yapacağımızı bilmeyiz.
break komutu gördüğü yerden döngüden çıkar ve **continue** komutu ile karşılaşıldığı zaman, döngünün bir sonraki iterasyonuna geçilir.

NON-SCALAR VERİ TİPLERİ

LİST

`list = [eleman1, eleman2, eleman3]`
farklı veri tipleri içerebilir.
indexing ve slicing yapılabilir.
mutable yapısı var elemanları değiştirilebilir.
elemanları değiştirken:
`l[0:3] = 30, 40, 60`
slicing yaptığım kısımla değiştireceğim yerin eleman sayısı eşit olmak zorunda değil.
`l[0:3] = [30]`
tek bir değer değiştireceksek direk tek olarak yazamayız. iterable olması gerekir liste formatında yazılmalı.

LİST FONSKİYONLARI

len() liste kaç elemanlı?
append() listenin sonuna eleman ekler.
extend() listenin sonuna birden çok eleman ekler.
insert() spesifik bir indexe eleman ekler.
remove() elemanı listeden siler (ilk gördüğünü siler).
pop() elemanı listeden siler ve döndürür.
count() elemanın listede kaç defa göründüğünü döndürür.
ındex() eleman ilk hangi indexte görünüyorsa onu döndürür.
reverse() listeyi tersine çevirir. (inplace)
`l[::-1]` slicingle de benzer bir işlem yapılabilir ama inplace yapmaz.
sort() listeyi sıralar. (inplace)
`l2 = sorted(l)` (inplace değil)
map(fonksiyon, iterable):
fonksiyon: listedeki her öğeye uygulanacak işlem. genellikle int, str, float gibi dönüşüm fonksiyonları veya kendi yazdığın bir fonksiyon olabilir.
iterable: liste, tuple, string gibi gezilebilir veri tipi.
map() her öğeye fonksiyonu uygular ve bir map objesi döner. bunu listeye çevirmek için `list(map(...))` yaparız.

ALIASING

`List2 = List` dediğimde Listte yapılan her değişiklik List2'yi etkiler.
`List2 = List.copy()` artık Listte yapılan değişiklik List2'yi etkilemez.

TUPLE

`tuple = (eleman1, eleman2, eleman3) ya da eleman1, eleman2, eleman3`
immutable yapısı vardır.
farklı veri tipleri içerebilir.
indexing ve slicing yapılabilir.
tupleda elemanların yerlerini değiştirmek için:
`(x, y) = (y, x)`

in keywordu eleman listede/tupleda var mı yok mu sorgular.

DİCTİONARY

`dictionary = {key1:value1, key2:value2, key3:value3}`
birden fazla listeyi bir yapıda tutmak için kullanılır.
keyler immutable yapıdadır.
valueler mutable ya da immutable olabilir ve farklı veri tipleri içerebilir.
diğer non-scalar veri tiplerindeki gibi elemana ulaşmak için köşeli parantez kullanılır:
`dictionary[key1]`
eleman eklemek için: `dictionary[key4] = value4`

boş dictionary oluşturabilmek için: `d = {}`
del() dictionaryden eleman siler.

SET
kümeler gibi düşünülebilir.
mutable yapısı vardır.
indexlenemez.
özgün değerler içerir aynı eleman bir sette birden fazla bulunmaz.
dictionaryden daha az yer kaplar.
boş set oluşturabilmek için: `s = set()`
add() sete eleman ekler.
remove() setten eleman siler.
discard() setten eleman siler ama yazılan eleman yoksa error vermez.
s1.difference(s2) s1-s2
U ya da **&** birleşim
n kesişim
s1.intersection(s2) s1 ile s2'nin kesişimi yapıp değeri s1'e atar.
s1.union(s2) s1 ile s2'nin birleşimini döndürür.
s1.isdisjoint(s2) s1 ile s2'nin kesişimi boş kümeyse False değilse True döndürür.
`len(s1.intersection(s2)) == 0`
s1.issubset(s2) s1'in s2'nin alt kümesiyse True değilse False döndürür.
s1.issuperset(s2) s1'in s2'nin üst kümesiyseTrue değilse False döndürür.

NON-SCALAR VERİ TİPLERİNDE FOR

```
for <değişken> in <obje>
t = 0

for e in notlar:
    t += e

ortalama = t / len(notlar)

print(ortalama)
t = 0

for i in range(len(notlar)): #bize 0,1,2.. len(notlar)-1 sayılarını verecek
    t += notlar[i]

ortalama = t / len(notlar)

print(ortalama)
```

range() indexlerde iterasyon yapıp indexing ile değerlerine de ulaşabilir, yukardaki iki kodun mantığı da aynıdır.

listenin içindeki tüm elemanlarda değişiklik yapmak için:

```
for i in range(len(liste)):
        liste[i] += 10
```

dictionarylerde iterasyon keyler üzerinden gerçekleşir. valuelarına da erişebilmek için:

```
for k in d:
    v = d[k]
    print(v)
```

dictionarylerde direkt valuelar üzerinden iterasyon yapmak için:

```
for v in d.values():
    print(v)
```

dictionarylerde hem key hem de valuelarda iterasyon yapmak için:

```
for k,v in d.items():
    print("key değeri:", k, "value değeri:", v)
```

SPLİT VE JOİN

split() belirli bir bölme kriterine göre string'in alt parçalarını listenin elemanları olarak dönüştürür. parantezin içine neye göre böleceğimizi yazarız hiç bir şey yazmazsak default olarak boşluğa göre böler.

```
liste.split(pattern)
```

join() listenin elemanları arasına belirtilen yapıyı koyup string'e dönüştürür.

```
"patern".join(liste)
```

COMPREHENSİONS

LİST COMPREHENSİON

```
squares = []
for i in range(1,11):
    squares.append(i*i)

squares = [i * i for i in range(1,11)]

odd_squares = []
for e in squares:

    if e % 2 == 1:
        odd_squares.append(e)

odd_squares = [e for e in squares if e % 2 == 1]
```

SET COMPREHENSİON

```
set_numbers = {s for s in numbers if s in [1,2,3,4,5,6,1,2]}
```

DİCTİONARY COMPREHENSİON

```
square_dict = {e:e * e for e in range(1,11)}
```

NESTED LİST COMPREHENSİON

```
m = [[j for j in range(7)] for _ in range(5)]
```

matrixi list comprehension ile flat etmek için:

```
flatten_m = [e for l in m for e in l] #m'deki her alt liste (l) içindeki her elemanı (e) düz listeye ekler.
```

VARİABLE UNPACKİNG

bir kaç tane değişkeni aynı anda bir kaç tane değere atayabilmek için:

```
x, y, z = (1, 2, 3)
```

sağ ve soldakilerin sayıları farklıysa hata verir.

Bunu gidermek için * kullanılır.

x ve y integer olurken z liste olmuş olur. yıldız sembolünü sadece bir kez kullanabiliriz.

```
x, y, *z = (4, 7, 11, 4, 21) #ilk iki elemanı x ve y'ye eşitle, sonuna kadar kalan diğer tüm elemanları z'ye eşitle
```

ENUMERATE()

aynı anda hem indexlerde hem elemanlarda iterasyon yapmak için:

```
for i, e in enumerate(adlar):
    print(i, "indexindeki eleman:", e)
```

`enumerate(adlar, start = 2):` bu şekilde kaçıncı elemandan başlayacağını belirtebiliriz.

zip() farklı yapıların içinde paralel iterasyon yapabilmek için:

```
for i in range(len(ogrenciler)):
                s = ogrenciler[i]
                g = notlar[i]

                print(s,g)
#yukarıdaki ve aşağıdaki kod aynı işi yapar.
for s, g in zip(ogrenciler, notlar):
    print(s, g)
```

zip() ile dictionary yaratmak için:

```
for k, v in zip(keys, values): d[k] = v
```

FONKSİYONLAR

Fonksiyonlar kodda abstraction (soyutlama) ve decomposition (problemi küçük parçalara ayırma/modülerlik) yapmayı sağlar.

fonksiyon tanımlamak için: `def fonksiyonun_adı(input):`

return(): fonksiyonun sonucunu döndürür.

fonksiyonlar return'ü gördükten sonra altında kalan kodları çalıştırmaz.

void fonksiyonlar değer döndürmeyen fonksiyonlardır return yoktur.

predefined parameters kullanıcı özellikle bir değer belirtmedikçe önceden atanmış değeri baz alır.

first class function bir fonksiyonu başka bir fonksiyona argüman olarak verilebilir.

For - Function

```
def apply(l, f):
""" l bir liste, f listenin tüm elemanlarına uygulanacak fonksiyon sonunda
listenin orijinali elemanlarına fonksiyonun uygulanmış haliyle güncellenir """
```

Underscore Placeholders

uzun sayıların arasına alt çizgi koyarak okumayı kolaylaştırmak değeri değiştirmez. `num = 10_000_000_000`

F-Strings

```
"x'in değeri" + " " + str(x)
f "x'in değeri {x}"
```