# Veri Türleri

Python'da farklı türde verilerle çalışmak için çeşitli veri türleri bulunur. Her veri türü, farklı türde bilgileri saklamak ve işlemek için tasarlanmıştır. Python'daki temel veri türleri şu şekildedir.

1. **Tam Sayılar (int):** Tam sayıları ifade eder. Negatif, pozitif veya sıfır olabilirler. Örneğin: 10, -5, 0.
2. **Ondalıklı Sayılar (float):** Ondalıklı (kesirli) sayıları temsil eder. Virgülden sonra ondalık kısmı olan sayılardır. Örneğin: 3.14, 0.5, -10.75.
3. **Karakter Dizileri (str):** Karakterlerden oluşan metinleri temsil eder. Tek tırnak ya da çift tırnak arasında yazılır. Örneğin: "Merhaba", 'Python'.
4. **Mantıksal (boolean) Veri Türü (bool):** Yalnızca iki değere sahip olabilir: True (doğru) veya False (yanlış). Mantıksal ifadeleri ve koşulları kontrol etmek için kullanılır. Örneğin: 5 > 3 ifadesi True sonucunu döner.
5. **Listeler (list):** Birden fazla değeri bir arada tutabilen, sıralı ve değiştirilebilir veri yapılarıdır. Listeler köşeli parantezler `[ ]` ile tanımlanır ve içinde farklı veri türlerini barındırabilir. Örneğin: [1, 2, 3, "Python"].
6. **Demetler (tuple):** Listelere benzer ancak sabit (değiştirilemez) veri türüdür. Bir kez tanımlan-dıktan sonra içerikleri değiştirilemez. Parantez `( )` ile tanımlanır. Örneğin: (10, 20, 30).
7. **Sözlükler (dict):** Anahtar-değer (key-value) çiftleriyle çalışan veri yapısıdır. Bir anahtara karşılık gelen bir değer saklanır ve süslü parantez `{ }` ile tanımlanır. Örneğin: {"ad": "Ahmet", "yaş": 25}.
8. **Kümeler (set):** Eşsiz (tekrarsız) değerler kümesidir. Sırasızdır ve süslü parantez `{ }`ile tanımlanır. Aynı değeri birden fazla içermez. Örneğin: {1, 2, 3, 4}.

## Veri dönüşümleri

Python'da tamsayıları 10'luk sistem dışında ikilik, sekizlik veya onaltılık sistemde yazmak da mümkündür. İkilik sistemde bir tamsayı yazmak için sayının başına **0b** (0 ve binary kelimesinin ilk harfi) getirilir. Benzer şekilde sekizli sayma düzeni için sayının başına **0o**, 16'lı sayma düzeninde bir sayı yazmak için sayının başına **0x** karakterleri getirilir.

```
>>> 0b100    # İkilik düzende 100 sayısının onluk düzendeki karşılığı
4
>>> 0o100    # Sekizlik düzende 100 sayısının onluk düzendeki karşılığı
64
>>> 0x100    # Onaltılık düzende 100 sayısının onluk düzendeki karşılığı
256
```

Bir gerçek sayıyı tam sayıya çevirmek için **int()** fonksiyonu kullanılır. Bu fonksiyon, gerçek sayının ondalık bölümünü atar. Ayrıca **int()** fonksiyonu, metin olarak girilmiş bir sayıyı da normal sayıya çevirebilir. **int()** fonksiyonunu kullanırken, ikinci bir argüman olarak sayının hangi sayma sisteminde olduğunu da belirtebiliriz.

```
>>> int(-3.99)
-3
>>> int(4.3)
4
>>> int("1807")
1807
```

```
>>> int('1010', 2)       # İkili sayı
10
>>> int('A0', 16)        # Onaltılı sayı
160
>>> int('100', 4)        # Dörtlü sayı
16
```

Python ondalık kısmı olan her sayıyı float olarak kabul eder. Tamsayılardaki int() fonksiyonu gibi bir veriyi ondalıklı veri tipine çevirmek için **float()** fonksiyonu kullanılır.

```
>>> float(14)
14.0
```

```
>>> float('21.45')
21.45
```

## Metin (String) Veri Tipi

Metin veri tipi bir veya daha fazla karakter dizisinden oluşur. Metin veri tipi Pyhon'da çift tırnak işareti (" ") veya tek tırnak işareti (' ') arasında yazılarak belirtilir. Alışkanlık olması açısından sadece bir tanesini kullanmakta fayda vardır. Metin veri tipinde tek ve çift tırnak işareti birlikte kullanılabilmektedir.

```
>>> "Python, 'Guido van Rossum' tarafından geliştirilmiştir."
```

Yukarıdaki metni aşağıdaki şekillerde yazarsanız hata mesajı ile karşılaşırsınız.

```
>>> "Python, "Guido van Rossum" tarafından geliştirilmiştir."
>>> 'Python, 'Guido van Rossum' tarafından geliştirilmiştir.'
"Python, "Guido van Rossum" tarafından geliştirilmiştir."
          ^
SyntaxError: invalid syntax
```

Bir metinde herhangi bir sıradaki karakteri çekmek için köşeli parantez `[ ]` içinde sıra numarası yazdır. Numaralama sol baştan sağa doğru 0,1,2,… şeklinde gider. ders = "Bilgisayar Bilimi" şeklinde verimiz olsun. İndeks numaralara şöyle olacaktır.

| B | i | l | g | i | s | a | y | a | r | | B | i | l | i | m | i |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |

```
>>> ders = "Bilgisayar Bilimi"
>>> ders[0]
'B'
>>> ders[3]
'g'
>>> ders[9]
'r'
```

```
>>> ders[11]
'B'
>>> ders[13]
'l'
>>> ders[15]
'm'
```

## Metin (String) Veri Tipi

Bir metinde herhangi bir sıradaki karakteri köşeli parantez `[ ]` ile çekebilmemize rağmen indeks numarası kullanarak mevcut metin değiştirilemez. Ayrıca farklı metin verileri toplam operatörü ile birleştirilebilir. Bir veriyi metne çevirmenin yolu **str()** fonksiyonunu kullanmaktır.

```
>>> ulke = "türkiye"
>>> ulke[1]; ulke[3]; ulke[5]
'ü'
'k'
'y'

>>> ulke[0] = "T"
Traceback (most recent cali last):
File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

```
>>> msj = "selam " + ulke; msj
'selam türkiye'

>>> x = 12345; type(x)
<class 'int'>

>>> y = str(x); y; type(y)
'12345'
<class 'str'>
```

Python'da metinleri birden fazla satırda yazmak da mümkündür. Bunun iki yolu vardır. İlk yolu, daha önce açıklama yazımında gördüğümüz gibi, metnin başında ve sonunda üçer adet çift tırnak ya da tek tırnak işareti kullanmaktır. Arada kalan metni farklı satırlara yazabilirsiniz.

```
>>> mesaj = """Python'da metinler
    birden fazla satıra
    yazılabilir."""

>>> mesaj; print(mesaj)
"Python'da metinler\nbirden fazla satıra\nyazılabilir."
Python'da metinler
birden fazla satıra
yazılabilir.
```

Görüldüğü üzeri direkt metin yazıldığında satır sonları **`\n`** sembolü ile gösterilmektedir. Ancak metin print ile yazıldığı kelimeler satır sonlarından bölünmektedir.

## Metin (String) Veri Tipi

Bir metin içine satırları bölmek istediği-miz yerlere `\n` sembolü konulur. Ayrıca metin içinde sekme (tab) sembolü `\t` konularak belirli boşluklar bırakılabilir. Metin içinde çift tırnak işareti veya tek tırnak işareti kullanmak için \" ve \' kullanılır.

```
>>> metin = "Bu metin \nbirden satıra\nyazılabilir."
>>> metin
'Bu metin \nbirden satıra\nyazılabilir.'
>>> print (metin)
Bu metin
birden satıra
yazılabilir.

>>> metin = "Python \t ile \t programlama."; print (metin)
Python   ile   programlama.

>>> metin = "Python, \"Guido van Rossum\" tarafından geliştirilmiştir."
>>> metin
'Python, "Guido van Rossum" tarafından geliştirilmiştir.'

>>> print (metin)
Python, "Guido van Rossum" tarafından geliştirilmiştir.
```

Python'da "\" karakterinin özel bir kullanımı olduğundan dosya ve klasör adreslerini yazarken bu karakterin olduğu gibi kullanılması hata mesajı alınmasına yol açabilir. Bunun önüne geçmek için yazılacak metnin önüne **"r"** harfi getirilir. Bu harf "raw" yani ham kelimesinin kısaltmasıdır ve yazılan metnin olduğu gibi alınmasını sağlar.

```
>>> dir = r"C:\users\ilker\Desktop"; dir; print (dir)
'C:\\users\\ilker\\Desktop'
C:\users\ilker\Desktop
```

Nasıl ki metinlerle toplama işlemi yapmak mümkünse, çarpma işlemi de mümkündür. Çarpma işlemi de bir metni yan yana istenen sayıda yazmak için kullanılır.

```
>>> yazi1 = "Bilgisayar "
>>> yazi2 = "Bilimleri"
>>> yazi = yaz1 + yazi2; yazi
'Bilgisayar Bilimleri'

>>> dil = "Python"
>>> dil3 = dil * 3
>>> dil3
'PythonPythonPython'
```

## Metin (String) Veri Tipi

Python'da metinlerle işlem yapmak ve metinleri değiştirebilmek için geliştirilmiş birtakım metotlar bulunmaktadır. Bunlardan sıklıkla kullanılanları **.upper(), .lower(), .split(), .replace()** ve **.find()** olarak verilebilir.

```
>>> durum = "Python Programlama ve Uygulama"
>>> durum.upper(); durum.lower()
'PYTHON PROGRAMLAMA VE UYGULAMA'
'python programlama ve uygulama'

>>> yeni = durum.replace('Python', 'Java'); yeni
'Java Programla ve Uygulama'

>>> kelime = 'Programlama'
>>> kelime.find('r')
1
>>> kelime.find('f')
-1
```

Metinlerin belirli bir karakterle parçalara ayrılması için **.split()** fonksiyonu sıklıkla kullanılmaktadır. Bölünen kelimeler bir liste verisi olarak gösterilir. **.split()** fonksiyonu parametresiz kullanılırsa bölünme işleme boşluk karakterine göre yapılır. Fonksiyonda 2 parametre girişi vardır. 1. parametre bölünme yapılacağı karakter veya karakter grubudur, 2. parametre ise metinden toplam kaç parça alınacağının sayıdır.

```
>>> durum = "Python Programlama ve Uygulama"
>>> durum.split()
['Python', 'Programlama', 've', 'Uygulama']

>>> mtn = "merhaba, adım Ahmet'tir, 25 yaşındayım"
>>> mtn.split(',')
['merhaba', " adım Ahmet'tir", ' 25 yaşındayım']

>>> mtn = "merhaba, adım Ahmet'tir, 25 yaşındayım, İstanbul yaşıyorum"
>>> mtn.split(',', 2)
['merhaba', " adım Ahmet'tir", ' 25 yaşındayım, İstanbul yaşıyorum']

>>> mtn.split(',', 1)
['merhaba', " adım Ahmet'tir, 25 yaşındayım, İstanbul yaşıyorum"]
```

## Metin (String) Veri Tipi

Metinlerin belirli bir karakterle parçalara ayrılması için **.split()** fonksiyonu sıklıkla kullanılmaktadır. Bölünen kelimeler bir liste verisi olarak gösterilir. **.split()** fonksiyonu parametresiz kullanılırsa bölünme işleme boşluk karakterine göre yapılır. Fonksiyonda 2 parametre girişi vardır. 1. parametre bölünme yapılacağı karakter veya karakter grubudur, 2. parametre ise metinden toplam kaç parça alınacağının sayıdır.

```
>>> durum = "Python Programlama ve Uygulama"
>>> durum.split()
['Python', 'Programlama', 've', 'Uygulama']

>>> mtn = "merhaba, adım Ahmet'tir, 25 yaşındayım"
>>> mtn.split(',')
['merhaba', " adım Ahmet'tir", ' 25 yaşındayım']

>>> mtn = "merhaba, adım Ahmet'tir, 25 yaşındayım, İstanbul yaşıyorum"
>>> mtn.split(',', 2)
['merhaba', " adım Ahmet'tir", ' 25 yaşındayım, İstanbul yaşıyorum']

>>> mtn.split(',', 1)
['merhaba', " adım Ahmet'tir, 25 yaşındayım, İstanbul yaşıyorum"]
```

Ayrıca **.title(), .capitalize(), .rstrip(), .lstrip(), .strip(), .startswith() ve .endswith()** metotları da bulunmakta-dır. Ayrıca, bir metnin belirtilen formatta olup olmadığını kontrol etmek için kullanılan **.isupper(), .istitle(), .isspace(), .isnumeric(), .islower(), .isdigit(), .isdecimal()** yöntemleri de bulunmaktadır.

```
>>> mtn = "bilgisayar"
>>> mtn.islower()
True

>>> mtn.isnumeric()
False
```

```
>>> mtn = "12345"
>>> mtn. isnumeric()
>>> mtn. isdigit()
True

>>> mtn.isupper()
False
```

## List (Liste) Veri Yapısı

Liste, isminden de anlaşılacağı üzere sıralı nesnelerden oluşan liste ya da dizilerdir. Listelerde yer alan veriler köşeli parantez `[ ]` içinde ve virgülle ayrılarak gösterilir. Listeler veriyi eklenme sırasına göre tutan ve içeriği sonradan değiştirilebilen veri yapılarıdır. Bir listeyi tanımladıktan sonra listenin bir veya daha fazla elemanını güncellemek mümkündür.

Listenin elemanları önceki bölümde gördüğümüz veri tiplerinden herhangi birisi olabilir. Bir listenin elemanları farklı veri tipinde de olabilir. Hatta, bir listenin elemanları farklı listeler ve değişken de olabilir.

```
>>> notlar1 = ['Eren',100, 'İlhan',99, 'Ali',98, 'Erman',97, 'Mehmet',96]
>>> notlar2 = [['Eren', 100],
               ['Ilhan',99],
               ['Ali',98],
               ['Erman',97],
               ['Mehmet',96]]
>>> notlar3 = [['Eren', 'İlhan', 'Ali', 'Erman', 'Mehmet'],
               [100, 99, 98, 97, 96]]
```

## Tuple (Demet) Veri Yapısı

Bir başka veri yapısı olan **tuple**, listelere çok benzer. Demetlerde yer alan veriler yuvarlak normal parantez `( )` içinde ve virgülle ayrılarak gösterilir. Listeler gibi verileri aldıkları sırada saklar ve saklanan veriye indeks numarası ile erişmek mümkündür. Ancak listelerden farklı olarak bir defa tanımlandıktan sonra demet veri yapısındaki değerleri değiştirmek mümkün değildir. Bu nedenle, oluşturulan bir verinin kesinlikle değiştirilmemesi gereken zamanlarda demet veri yapısı daha güvenilirdir. Bu özellik, veri güvenliği ve korunmasının gerekli olduğu işlerde çok faydalıdır.

```
>>> demet1 = (3.14, 2.78, 5.67, 2.18)
>>> demet2 = ("İstanbul", "Ankara", "İzmir", "Bursa", "Adana")
>>> demet3 = ("Fizik", "Matematik", "Kimya", 100, 99, 97)
```

## Dictionary (Sözlük) Veri Yapısı

Dictionary ya da sözlük veri yapısı, adından da anlaşılacağı üzere anahtar kelimelere değerlerin atandığı sözlük benzeri yapılardır. Nasıl ki bir sözlükte her sözcüğün karşısında bir açıklama yazıyorsa **dict** veri yapısında da her bir anahtar kelimenin karşısında yazan bir değer vardır. Diğer bir deyişle, **dictionary**, her bir değere bağlı bir anahtar kelimenin olduğu veri türüdür. Anahtarlar harf ve rakamlardan oluşan alfa nümerik türde veriler iken değerler herhangi bir veri tipinde olabilir. Hatta anahtar:değer ikilisindeki değer bir başka sözlük de olabilir.

Sözlükler ise küme parantezleri `{ }` kullanılarak oluşturulur. Küme parantezi içinde, anahtar kelime tırnak işareti içinde yazılıp iki noktadan sonra anahtar kelimenin aldığı değer yazılır. Farklı anahtar - değer İkilileri de birbirlerinden virgülle ayrılır. Örneğin belirli firmaların hisse senedi fiyatlarının olduğu liste olsun.

```
>>> hisseler = {'firma1':17.70, 'firma3':28.70, 'firma5':7.05, 'firma7':13.56}
>>> print(hisseler)
{'firma1':17.70, 'firma3':28.70, 'firma5':7.05, 'firma7':13.56}
```

## Set (Küme) Veri Yapısı

Veri nesnelerini herhangi bir sıralama olmaksızın ve her nesneden sadece bir tane olacak şekilde kaydetmek istersek **set (küme)** veri yapısını kullanabiliriz. Set veri yapısı adını matematikteki küme tanımından almıştır. Tıpkı matematikteki küme tanımı gibi bu veri yapılarında her veriden sadece bir tane bulunur ve verilerin sırası önemli değildir. Setler de listeler gibi sonradan değiştirilebilir (mutable) verilerdir. Set (küme) veri yapısı oluşturmak için **set()** fonksiyonu kullanılır. Örneğin, bir alışveriş listesini bir set veri yapısı olarak kaydedildiğini varsayalım.

```
>>> alisveris_listesi = ['ekmek', 'süt', 'yoğurt', 'pirinç', 'bal', 'meyve', 'ekmek', 'süt']
>>> alisveris = set(alisveris_listesi)

>>> print(alisveris)
{'süt', 'pirinç', 'bal', 'meyve', 'yoğurt', 'ekmek'}
```

# Giriş/Çıkış İşlemleri

Python'da giriş (input) ve çıkış (output) işlemleri, kullanıcının veri girişi yapması ve bu verilerin ekrana veya dosyaya yazdırılması gibi işlemleri ifade eder. Giriş işlemleri için Python'da en yaygın kullanılan fonksiyon **"input()"** fonksiyonudur. Bu fonksiyon, kullanıcıdan klavye aracılığıyla veri girmesini sağlar. Örneğin, bir sayı veya metin girmesini isteyebilirsiniz. **"input()"** fonksiyonu her zaman bir string (metin) döndürür, bu nedenle girilen veriyi farklı veri türlerine dönüştürmek gerekiyorsa uygun tür dönüşümü yapılmalıdır. Örneğin, sayısal bir giriş istiyorsanız **"int()"** veya **"float()"** ile dönüştürme yapabilirsiniz:

```
isim = input("Adınızı girin: ")
yas = int(input("Yaşınızı girin: "))
```

**"input()"** fonksiyonu kullanıldığı zaman parametre olarak verilen metin ekrana yazılır ve verinin girilip **"enter"** tuşuna basılmasına kadar beklemede kalınır. Bu özelliği sebebiyle çoğu zaman ekranda bir bilginin gösterilip beklenmesi için de kullanılmaktadır.

```
bilgi = """
Açıklamaları dikkatli okuyun
işlemleri uygun sırada ve
doğru şekilde yapınız.
"""
print(bilgi)
input()
```

Bu sayede program kullanıcıdan bir girişi bekleyecek ve **"enter"** tuşuna basıncaya kadar da kapanmayacaktır.

## input fonksiyon örneği

Bir kullanıcıdan ismini soran ve ona selam mesajı yazan bir durumu dikkate alalım. Aşağıdaki kodu IDLE editöründe dosya modunda yazıp **"adsor.py"** ismi ile kaydedelim.

```
isim = input("İsminiz Nedir?: ")
print("Merhaba ", isim, end="!\n")
```

Doğal olarak sorulan soruya verilen cevaba göre ekrana yazılan mesaj farklı olacaktır. Örneğin eğer bu soruya "Semih" cevabı verilmişse mesaj "Merhaba Semih!" şeklinde olacaktır.

Görüldüğü gibi **"input()"** fonksiyonunda parantez içine verilen parametre, kullanıcıdan veri alınırken kullanıcıya sorulacak soruyu göstermektedir. Şimdi bu kodu biraz daha geliştirelim ve aşağıdaki kodlar önceki programa ekleyip tekrar kaydedelim.

```
yas = input("Yaşınız: ")

print ("Demek ", yas, " yaşındasınız.")
print ("Genç mi yoksa yaşlı mı olduğuna karar veremedim.")
```

Program tekrar çalıştırıldığında isim girildikten sonra bu sefer kullanıcının yaşının girilmesi bekleyecek ve giriş yapıldıktan sonra diğer mesajlar yazılacaktır.

```
İsminiz Nedir?: Canan
Merhaba Canan!
Yaşınız: 35
Demek 35 yaşındasınız.
Genç mi yoksa yaşlı mı olduğuna karar veremedim.
```

## İnput() ile sayısal değerler

**"input()"** fonksiyonu ile alınan veriler metin veridir. Bu nedenle sayısal veriler bu fonksiyon ile alınırken veri tipi dönüşümleri uygulanmalıdır. Bu durumu aşağıdaki kod ile test edelim.

```
veri = input("Sayısal ya da nümerik bir veri giriniz: ")

tip = type(veri)

print("Girilen veri tipi :", tip)
```

Program çalıştırıldığında ister alfabetik ister nümerik veriler girilsin, girilen verinin türü **"str"** olacaktır. Bu bakımdan eğer sayısal veriler girdi olarak kullanılacaksa muhakkak **"float()"** ya da **"int()"** ile veri dönüşümü yapılmalıdır.

Bir dairenin çapını isteyen ve dairenin alanını hesaplayan aşağıdaki kodu IDLE editöründe dosya modunda yazıp **"daire.py"** ismi ile kaydedelim ve çalıştıralım. Elbette verilen daire çapına göre farklı çıktılar çıkacaktır.

```
cap = input("Dairenin çapı: ")

yaricap = int(cap) / 2
pi = 3.1415
alan = pi * yaricap**2

print("Çapı ", cap, "cm olan dairenin alanı: ", alan, "cm2'dir")
```

```
Dairenin çapı: 12
Çapı 12cm olan dairenin alanı: 113.094cm2'dir
```

## Çıkış işlemleri

Çıkış işlemleri, verilerin bir ekrana veya başka bir çıktı ortamına yazdırılmasıdır. Python'da bu işlem için en yaygın kullanılan fonksiyon **"print()"** fonksiyonudur. **"print()"** fonksiyonu, ekrana veri yazdırmak için kullanılır ve birden fazla değeri virgül ile ayırarak yazdırabilir. Ayrıca **"print()"** ile dosyaya yazdırma işlemi de yapılabilir. Dosya işlemleri için Python'da **"open()"** fonksiyonu kullanılır. Dosya açıldığında, **"write()"** fonksiyonu ile dosyaya veri yazılabilir, **"read()"** ile dosyadan veri okunabilir. Çıkış işlemlerini dosyaya yönlendirmek, verileri saklamak ve daha sonra işlemek için önemlidir. Sıklıkla kullanılan dosyaya yazım formatı şöyledir.

```
with open("dosya.txt", "w") as dosya:

    dosya.write("Merhaba, Python!")
```

Ayrıca **"print()"** fonksiyonuyla da dosyaya veri yazmak mümkündür. Hatırlamak gerekirse print() fonksiyonun **"file"** parametresi ile istenilen veriler **"open"** fonksiyonu ile belirtilen dosyaya yazılabilmektedir. Aşağıdaki kodu IDLE editöründe dosya modunda yazıp **"dosyakyt.py"** ismi ile kaydedelim ve çalıştıralım.

```
f = open("bilgiler.txt", "w")

print ("1. bilgi satırı\n", file=f)
print ("2. bilgi satırı\n", file=f)
print ("3. bilgi satırı\n", file=f)

f.close()

print ("Kayıt işlemi tamamlandı")
```

Program ekrana "Kayıt işlemi tamamlandı" mesajını yazdığında, programın çalıştığı klasörde "bilgiler.txt" dosya oluşturulmuş ve içine belirtilen bilgiler yazılmış olacaktır.

# open() fonksiyonu kullanımı

Python'da **"open()"** fonksiyonu, dosya okuma ve yazma işlemleri yapmak için kullanılır. Bu fonksiyon, belirtilen dosyayı açar ve üzerinde okuma, yazma, ekleme gibi işlemler yapılmasını sağlar. **"open()"** fonksiyonu, dosya açarken iki temel parametre alır: dosya adı ve dosya modu. Dosya modları, dosyanın nasıl açılacağını (okuma, yazma vb.) belirler.

**"open()"** fonksiyonunda kullanılan dosya modları:

- **r** : Sadece okuma modudur. Varsayılan olarak kullanılır. Dosya varsa açılır, yoksa hata verir.
- **w** : Yazma modudur. Dosya varsa içeriği silinir ve üzerine yazılır. Dosya yoksa yeni bir dosya oluşturur.
- **a** : Ekleme modudur. Dosya varsa sonuna veri ekler, yoksa yeni bir dosya oluşturur.
- **r+** : Hem okuma hem yazma modudur. Dosya yoksa hata verir.
- **b** : İkili (binary) dosyalar için kullanılır. Örneğin resim, video, ses dosyaları.
- **t** : Metin dosyaları için kullanılır (varsayılan mod).

### Dosya Yazma İşlemi

Bir dosyaya yazma işlemi yapmak için genellikle **"w" (yazma)** veya **"a" (ekleme)** modu kullanılır. "w" modu dosyanın var olan içeriğini silip üzerine yazarken, "a" modu mevcut verilerin sonuna ekleme yapar. Dosya işlemlerini bitirdikten sonra **"close()"** fonksiyonuyla dosya kapatılmalıdır.

Yazma işlemi

```
with open("dosya.txt", "w") as dosya:

    dosya.write("Merhaba, Python!")
```

Ekleme işlemi

```
with open("ornek.txt", "a") as dosya:

    dosya.write("\nYeni bir satır ekledim.")
```

Bu örneklerde **"open()"** ile dosya açılır, **"write()"** fonksiyonu ile içerik dosyaya yazılır. **"with"** yapısı, dosya işlemi bittiğinde otomatik olarak dosyayı kapatır.

# open() fonksiyonu

### Dosya Okuma İşlemi

Dosyayı okumak için "r" (okuma) modu kullanılır. Dosya var olduğu sürece okuma işlemi yapılabilir. Okuma işlemleri için Python'da üç ana yöntem vardır:

1. **read()** : Dosyanın tamamını okur.
2. **readline()** : Dosyadan bir satır okur.
3. **readlines()** : Dosyadaki tüm satırları bir liste olarak okur.

Okuma işlemi

```
with open("ornek.txt", "r") as dosya:
    icerik = dosya.read()
    print(icerik)
```

Satır satır okuma

```
with open("ornek.txt", "r") as dosya:
    satir = dosya.readline()
    while satir:
        # .strip() boşlukları ve satır sonunu kaldırır
        print(satir.strip())
        satir = dosya.readline()
```

### Dosya Kapama

Dosya işlemi bittiğinde **"close()"** fonksiyonu ile dosya mutlaka kapatılmalıdır. Ancak, yukarıdaki örneklerde olduğu gibi **"with"** bloğu kullanıldığında Python dosyayı otomatik olarak kapatır, bu da olası hataları önler.

# print() ile format() fonksiyonunun kullanımı

**"print()"** fonksiyonu ekrana yada bir dosyaya veri yazmak için kullanılan temel yöntemdir. Veriler yazılırken sıklıkla verilen belli düzende yazılması istenir. Bazen belirli verilerin hazır metinlerde belirli yerde yazılması gerekebilir. Python'da bu tür **"karakter dizisi biçimlendirme"** işlemleri için çok güçlü bir araç olan **"format()"** fonksiyonu kullanılır. **"format()"** fonksiyonu, bir dize içerisinde yer tutucular (süslü parantez **"{}"**) ile belirtilen yerlere, dinamik olarak değerler eklemeye olanak tanır. Bu sayede, statik metinlerle birlikte dinamik veriler de metin içerisinde kolayca kullanılabilir. Örneğin;

```
isim = "Samet"
yas = 32
mesaj = "Merhaba, benim adım {} ve {} yaşındayım.".format(isim, yas)
print(mesaj)
```

Bu örnekte, süslü parantezler sırasıyla **"isim"** ve **"yas"** değişkenleri ile doldurulur ve çıktı şu şekilde olur: "Merhaba, benim adım Samet ve 32 yaşındayım."

**"format()"** fonksiyonu, yer tutuculara sırasız değerler atamak yerine, **isimlendirilmiş argümanlar** kullanılarak da çalışabilir. Bu yöntem, daha okunabilir ve esnek bir kod yazmayı sağlar. Yukarıdaki örneği şu şekilde yazmak da mümkündür;

```
mesaj = "Merhaba, benim adım {ad} ve {yas} yaşındayım.".format(ad="Samet", yas=32)
mesaj2 = "Merhaba, ben {yas} yaşındayım, adım {ad}'tir.".format(ad="Samet", yas=32)
print(mesaj)
```

İsimlendirilmiş argümanlar oldukça esnek bir kullanım sağlarlar. Çünkü argüman isimleri serbest şekilde sırasız olarak kullanılabilir.

## format() fonksiyonu

Bir parametre dizisindeki öğelerin sırasını değiştirmek ya da belirli parametreler tekrarlanmak isteniyorsa, yer tutuculara parametrelerin sıra indeksleri verilerek "format()" fonksiyonu kullanılabilir. Örneğin:

```
mesaj = "{0} yaşında bir {1}.".format(25,"öğrenci")
print(mesaj)
```

şeklindeki bir kodu dikkate alalım. Burada "format()" fonksiyonu ile verilen parametre dizisinde parametreler solda itibaren numaralandırılır ve ilk elemanın indeks numarası sıfır (0)'dır. Buna göre bu örnekte "{0}" 25'i, "{1}" ise "öğrenci" kelimesini temsil eder. İsimlendirilmiş argümanlarda olduğu indeksli argümanlar da serbest şekilde sırasız olarak kullanılabilir.

**"format()"** fonksiyonu sayılar için yuvarlama, ondalık basamak belirleme gibi özellikler de sunar. Bu sayıların, özellikle de bilimsel formata uyacak şekilde, istenilen yazıma uygun şekilde ayarlama imkânı vermektedir. Örneğin, ondalıklı bir sayıyı sadece iki basamak şeklinde göstermek için:

```
pi = 3.141593
mesaj = "Pi sayısı: {:.2f} olarak verilir.".format(pi)
print(mesaj)
```

şeklinde yazılan kodun çıktısı "Pi sayısı: 3.14 olarak verilir." olacaktır. Burada sayı iki hane olarak yazılırken aynı zamanda sayıyı yukarıya yuvarlama işlemede yapılmaktadır.

# Karşılaştırma ve Mantık operatörleri

Python'da karşılaştırma ve mantık operatörleri, programlarda koşullu ifadeleri kontrol etmek, iki veya daha fazla değeri karşılaştırmak ve sonuçlara göre işlem yapmak için kullanılır. Bu operatörler, genellikle **"if"**, **"while"** gibi yapılarla birlikte kullanılır. Karşılaştırma operatörleri iki değeri karşılaştırırken, mantık operatörleri birden fazla koşulun sonucunu birleştirir.

**Karşılaştırma Operatörleri**

Karşılaştırma operatörleri, iki değerin birbirine göre ilişkisini değerlendirir ve **"True"** veya **"False"** sonucunu döner. Örneklerde "a = 5" kabul ediliyor.

**==** : **Eşitlik.** İki değerin eşit olup olmadığını kontrol eder.

```
print(a == 5)  # True
print(a == 3)  # False
```

**!=** : **Eşit değil.** İki değerin eşit olmadığını kontrol eder.

```
print(a != 3)  # True
print(a != 5)  # False
```

**<** : **Küçüktür.** Soldaki değerin, sağdakinden küçük olup olmadığını kontrol eder.

```
print(a < 7)  # True
print(a < 3)  # False
```

**>** : **Büyüktür.** Soldaki değerin, sağdakinden büyük olup olmadığını kontrol eder.

```
print(a > 3)  # True
print(a > 7)  # False
```

**>=** : **Büyük veya eşittir.** Soldaki değerin, sağdakine büyük veya eşit olup olmadığını kontrol eder.

```
print(a >= 3)  # True
print(a >= 7)  # False
```

**<=** : **Küçük veya eşittir.** Soldaki değerin, sağdakine küçük veya eşit olup olmadığını kontrol eder.

```
print(a <= 7)  # True
print(a <= 3)  # False
```

## Mantık Operatörleri

Karşılaştırma ve döngü deyimlerinde koşulun sınanması için karşılaştırma operatörleri kullanılır. Ancak birden fazla koşul olduğunda bunların birleştirilip tek bir koşul durumuna getirilmesi gerekir. Böyle durumlarda mantıksal operatörler kullanılır. Diğer bir deyişle, mantık operatörleri birden çok koşulun birleştirilmesi amacıyla kullanılır. Aritmetik operatörlerde olduğu gibi bu operatörlerde de işlem soldan sağa gerçekleştirilir. Ancak sonuçlar YANLIŞ ve DOĞRU'ları ifade ettiğinden karışıklıkları önlemek için parantezlerle kullanılmaları daha uygundur. Mantık operatörleri 3 tane olup bunlar, VE (AND), VEYA (OR) ve DEĞİL (NOT) işaretleridir.

1. **"and"** : Tüm koşulların doğru olması durumunda "True" döner. Aksi halde "False" döner.
2. **"or"** : Koşullardan en az birinin doğru olması durumunda "True" döner. İki koşul da yanlışsa "False" döner.
3. **"not"** : Mantıksal değeri tersine çevirir. "True" olan bir ifadeyi "False", "False" olanı "True" yapar.

Mantık operatörleri "VE / VEYA" her zaman iki koşulu karşılaştırmak için kullanılır. Yani **"A and B"** ya da **"A or B"** şeklinde kullanılırlar. Ayrıca gruplama suretiyle birden fazla karşılaştırma yapmak da mümkündür. A ve B birer koşul olmak üzere bu üç operatörün doğruluk tabloları şu şekildedir.

| VE (and) | | |
|---|---|---|
| A | B | Sonuç |
| Doğru 1 | Doğru 1 | Doğru 1 |
| Doğru 1 | Yanlış 0 | Yanlış 0 |
| Yanlış 0 | Doğru 1 | Yanlış 0 |
| Yanlış 0 | Yanlış 0 | Yanlış 0 |

| VEYA (or) | | |
|---|---|---|
| A | B | Sonuç |
| Doğru 1 | Doğru 1 | Doğru 1 |
| Doğru 1 | Yanlış 0 | Doğru 1 |
| Yanlış 0 | Doğru 1 | Doğru 1 |
| Yanlış 0 | Yanlış 0 | Yanlış 0 |

| DEĞİL (not) | |
|---|---|
| A | Sonuç |
| Doğru 1 | Yanlış 0 |
| Yanlış 0 | Doğru 1 |
| | |
| | |

### Mantık operatörleri kullanım örnekleri

Mantık operatörleri birden fazla koşulu kontrol etmek için kullanılır. Aşağıdaki **"and"** mantık operatörü ile yapılan örneğini inceleyelim.

```
yas = 20
ehliyet_var = True

if yas >= 18 and ehliyet_var:
     print("Araba sürebilirsiniz.")

else:
     print("Araba süremezsiniz!!!")
```

Bu örnekte yaşın hem 18'den büyük veya eşit olması hem de ehliyetin var olup olmaması kontrol edilir. İki koşul aynı anda sağlanıyorsa araba sürülebilir. Şimdi de **"or"** mantık operatörü ile yapılan örneği gözden geçirelim.

```
yas = 15
veli_izni = True

if yas >= 18 or veli_izni:
     print("Etkinliğe katılabilirsiniz.")

else:
     print("Etkinliğe katılamazsınız!!!")
```

Bu örnekte kişinin ya yaşı 18'e eşit ve büyük olacak ya da kişinin veli izninin olması durumunda etkinliğe katılmasına izin verilir.

# Aitlik ve Kimlik Operatörleri

Python'da aitlik ve kimlik operatörleri, veri yapılarını ve nesneleri karşılaştırmak veya kontrol etmek için kullanılır. **Aitlik operatörleri** bir değerin veya öğenin bir veri yapısında bulunup bulunmadığını kontrol ederken, **kimlik operatörleri** iki nesnenin aynı bellek adresini paylaşıp paylaşmadığını kontrol eder. Bu operatörler, özellikle nesnelerle çalışırken ve veri yapılarının üyelik durumlarını kontrol ederken önemlidir.

### Aitlik Operatörleri

Aitlik operatörleri, bir değerin bir veri yapısının (list, tuple, dictionary, string vb.) içinde olup olmadığını kontrol eder. Python'da iki temel aitlik operatörü vardır:

- **"in"** : Bir değerin bir veri yapısında bulunup bulunmadığını kontrol eder. Eğer değer veri yapısında mevcutsa, **"True"** değeri döneri, aksi takdirde **"False"** değeri döner.
- **"not in"** : Bir değerin bir veri yapısında bulunmadığını kontrol eder. Değer veri yapısında yoksa **"True",** varsa **"False"** döner.

```
liste = [1, 2, 3, 4, 5]

# "in" operatörü ile aitlik kontrolü
print(3 in liste)    # True (3, listede mevcut)
print(6 in liste)    # False (6, listede mevcut değil)

"not in" operatörü ile aitlik kontrolü
print(6 not in liste)    # True (6, listede yok)
print(3 not in liste)    # False (3, listede mevcut)
```

### Metin Değişkenlerde Aitlik Kontrolü

Bir metin içinde belirli kelimelerin olup olmadığı aitlik operatörü "in" ya da "not in" ile kontrol edilirken aitlik operatöründe önce verilecek değişken metin bir türü değişken olmalıdır. Aşağıdaki örneğe bir göz atalım.

```
metin = "Python programlama dili"

# "in" operatörü ile aitlik kontrolü
print("Python" in metin)    # True (Python kelimesi metinde geçiyor)
print("Java" in liste)      # False (Java kelimesi metinde geçmiyor)

"not in" operatörü ile aitlik kontrolü
print("kim" not in liste)    # True (kim kelimesi metinde yoktur)
print("dil" not in liste)    # False (dil kelimesi metinde vardır)
```

### Sözlük tipi verilerde aitlik kontrolü

Sözlüklerde aitlik kontrolu eğer sadece sözlük değişkeni ile kullanılırsa söz konusu operatörler varsayılan olarak sözlüğün anahtarlarını kontrol eder. Değerler üzerinde doğrudan bir işlem yapılmaz.

```
sozluk = {"ad": "Ahmet", "yas": 25}

# "in" operatörü ile aitlik kontrolü
print("ad"  in sozluk)    # True ('ad' anahtarı sözlükte vardır)
print("yas" in sozluk)    # True ('yas' anahtarı sözlükte vardır)

print("soyad" in sozluk)  # False ('soyad' anahtarı sözlükte yoktur)
```

Bu örnekte "in" operatörü sözlük değişkeninde anahtarları kontrol eder ve kontrol edilen anahtarın durumunu göre "True" veya "False" değerlerinden biri ile döner.

**Sözlüklerde değerler için aitlik kontrolü**

Bir sözlüğe ait değerler üzerinden direkt olarak sözlük değişkeni kullanılarak aitlik kontrolu yapılamamaktadır. Ancak sözlüğe ait ".values()" fonksiyonu bir liste tipi veri oluşturmasıyla bu aitlik kontrolu yapılabilir. Alttaki örneği inceleyelim.

```
kisi = {"ad":"Semih", "yas":45, "meslek":"öğretmen"}
deger = kisi.values()

# "in" operatörü ile sözlük değerleri için aitlik kontrolü
print("Semih"  in deger)    # True ("Semih" değerler içinde vardır)
print(30 in deger)          # False (30 değerler içinde yoktur)
```

Bir başka örnek olarak mesela bir marketin ürünlerinin fiyatlarının bulunduğu bir sözlük üzerinden, belirli bir fiyatın listede olup olmadığını kontrol edelim.

```
urun_fiyatlari = {
    "elma": 30,
    "armut": 40,
    "muz": 25,
    "kiraz": 20
}

fiyat = urun_fiyatlari.values()

# değerleri için aitlik kontrolü
print(40 in fiyat)    # True (40 fiyatı liste içinde vardır)
print(55 in fiyat)    # False (55 fiyatı liste içinde yoktur)
```

### Kimlik Operatörleri

Kimlik operatörleri, iki değişkenin aynı nesne olup olmadığını kontrol eder. Python'da nesneler bellek adresleriyle temsil edilir ve bu operatörler bellek adreslerini karşılaştırır. Python'da iki temel kimlik operatörü vardır:

- **"is"** : İki değişkenin aynı nesne olup olmadığını kontrol eder. Eğer aynı nesneyi işaret ediyorlarsa, **"True",** aksi takdirde **"False"** döner.
- **"is not"** : İki değişkenin aynı nesne olmadığını kontrol eder. Eğer farklı nesnelerse, **"True"**, aynı nesne ise **"False"** döner.

```
a = [1, 2, 3]
b = [1, 2, 3]
c = a

# "is" operatörü ile kimlik kontrolü
print(a is b)    # False (a ve b aynı değerlere sahip ama farklı nesneler)
print(a is c)    # True  (a ve c aynı nesneyi işaret ediyor)

"is not" operatörü ile kimlik kontrolü
print(a is not b)    # True  (a ve b farklı nesneler)
print(a is not c)    # False (a ve c aynı nesne)
```

Yukarıdaki örnekte, "a" ve "b" aynı değerlere sahip olmasına rağmen farklı nesnelerdir. Python'da her iki liste için farklı bellek adresleri ayrılmıştır. Ancak "c" değişkeni, "a" değişkenine atandığından, aynı bellek adresini işaret eder. Bu nedenle "a is c" ifadesi True döner.

"is" ve "==" arasında anlam bakımından büyük fark vardır: "==" iki değişkenin değerlerini karşılaştırırken, "is" iki değişkenin aynı bellek adresine sahip olup olmadığını karşılaştırır.

## Nesne Kimlik Bilgisi: id() fonksiyonu

Python'da **"id()"** fonksiyonu, bir nesnenin benzersiz kimliğini (bellek adresini) döndüren yerleşik bir fonksiyondur. Her nesne, Python'da oluşturulduğunda, bellekte bir adresle ilişkilendirilir ve **"id()"** fonksiyonu bu adresi sağlar. Bu, aynı değere sahip iki farklı nesnenin aynı mı yoksa farklı mı olduğunu kontrol etmek için kullanılabilir. Fonksiyona ait önemli noktalar:

- - "id()" fonksiyonu, bir nesnenin bellekteki konumunu temsil eden bir tamsayı döner.
- - Aynı nesneye atıfta bulunan iki değişken, aynı "id" değerine sahip olur.
- - Python, küçük tamsayılar ve bazı değiştirilemez (immutable) veri tipleri için aynı kimliği kullanabilir, bu nedenle küçük sayılar aynı kimliği paylaşabilir.

```
x = 10
y = 10
z = 20

print(id(x))   # x'in kimliği    id_x: 140736263545928, id_y: 140736263545928
print(id(y))   # y'nin kimliği, x ile aynı olabilir çünkü x ve y aynı değeri tutuyor
print(id(z))   # z'nin kimliği, farklı çünkü z'nin değeri farklı id_z:140736263546248
```

Bu örnek "id()" fonksiyonunun teme kullanımını gösterir:

- "x" ve "y" aynı değere (10) sahip olduğu için Python, bu iki değişken için aynı kimliği atayabilir. Bu, Python'un küçük tamsayılar için optimizasyonundan kaynaklanır.
- "z" değişkeni 20 değerine sahip olduğu için farklı bir id döner.

## Değiştirilebilir (Mutable) ve Değiştirilemez (Immutable) Nesnelerde Kimlikler

Değiştirilemez (Immutable) nesneler (örneğin, sayılar ve stringler) değiştirildiğinde yeni bir kimlik alırken, değiştirilebilir (Mutable) nesnelerde (örneğin listeler) aynı kimlik korunur.

```
# Değiştirilemez (Immutable) örnek: Integer
a = 5
print(id(a))   # a'nın kimliği       id_a :140736263545768

a = a + 1      # Yeni bir nesne oluşturuluyor
print(id(a))   # Yeni kimlik, çünkü a'nın değeri değiştirildi  id_a: 140736263545800

# Değiştirilebilir (Mutable)  örnek: Liste
liste = [1, 2, 3]
print(id(liste))    # Listenin kimliği    id_liste: 1846692356096

liste.append(4)     # Liste değiştirildi ama aynı nesne üzerinde
print(id(liste))    # Aynı kimlik, çünkü liste değiştirilebilir bir veri tipi
```

Bu örnekte:

- "a = 5" ve "a = a + 1" işlemi sonrasında "id(a)" değişir, çünkü integer (tamsayı) değiştirilemez bir veri tipidir. "a + 1" işlemi yeni bir integer nesnesi oluşturur.
- Ancak, "liste.append(4)" işlemi listenin içeriğini değiştirir fakat liste aynı bellek adresinde kalır. Bu yüzden "id(liste)" aynı kalır.

**Kimlik Operatörleri ile "id()" Kullanımı**

Python'daki **"is"** operatörü, iki değişkenin aynı nesneye işaret edip etmediğini kontrol eder. Bu kontrol, aslında **"id()"** fonksiyonu ile yapılır. Eğer iki değişkenin **"id()"** değerleri aynıysa, **"is"** operatörü True döner

```
x = [1, 2, 3]
y = x          # y, x ile aynı nesneyi işaret eder

print(id(x))   # x'in kimliği                      id_x: 1761665551808
print(id(y))   # y'nin kimliği, x ile aynı olmalı  id_y: 1761665551808

print(x is y)   # True, çünkü x ve y aynı nesneyi işaret ediyor

z = [1, 2, 3]   # z farklı bir nesne
print(id(z))    # z'nin kimliği farklı olacak      id_z: 1761665619136
print(x is z)   # False, çünkü z farklı bir nesne
```

Bu örnekte:

- **"x"** ve **"y"** aynı nesneyi işaret ettiği için **"x is y"** ifadesi True döner.
- **"x"** ve **"z"** aynı içeriğe sahip olabilir, ancak farklı nesneler olduğu için **"x is z"** False döner.