22/23 Bahar Dönemi İst268 Bilgisayar Programlamaya Giriş II Dersi Notları

Yorum Satırlarını Kullanma

Yorum satırları (comment lines), herhangi bir programlama dilinde olduğu gibi Python yorumlayıcısı(interpreter) tarafından dikkate alınmayan ve yorumlanmayan ifadelerdir.

Python'da yorum satırları genel olarak aşağıdaki işlemler için kullanılır:

- Bir hatırlatma eklemek
- Program veya kodlarla ilgili bir açıklama yapmak
- Kullanılmayan bir kod satırını pasif hale getirmek
- Süsleme yapmak

Bu tür açıklama satırları kodun başkaları tarafından daha iyi anlaşılmasını sağlar. Python'da tek satırlık açıklama için "#" işareti kullanılır. "#" işareti kullandığınızda o satırdaki metin kod olarak işlenmez.

Örnek1

Aşağıdaki yorum satırları Python tarafından dikkate alınmamaktadır. Bu nedenle sadece "print()" komutu çalışır.

```
1.  #Bu kod ekrana yazı yazılmasını sağlamaktadır.
2.  print ("Konu: Yorum satırlarını kullanma")
3.  #Her satırın başına # işareti eklenerek #alt alta yorum satırları oluşturulabilir.
```

```
In [34]: print ("Konu: Yorum satırlarını kullanma")
    ...: #Her satırın başına # işareti eklenerek #alt alta yorum satırları
oluşturulabilir.
Konu: Yorum satırlarını kullanma
```

Örnek2

Yorum satırlarını kod satırının devamında aynı satırda kullanabilirsiniz. Bu kullanımda önce kod gelir, devamında yorum satırı "#" işareti ile başlar (öndeki kod çalışır), daha sonra satır sonuna kadar yorum satırı olarak dikkate alınmaz.

```
1.  print (2+3) # Bu kod satırı ekrana 2 sayının toplamını yazar.
```

```
In [35]: print (2+3) # Bu kod satırı ekrana 2 sayının toplamını yazar.
5
```

Her satırın başına "#" işareti ekleyerek alt alta yorum satırları oluşturabilirsiniz. Yorum satırlarında özel karakterler etkisizdir.

Örnek3

Birden fazla yorum satırı kullanılacaksa yorumlar üçlü tek tırnak veya üçlü çift tırnak blokları arasına yazılır.

```
1.  '''Python'da birden fazla açıklama satırı kullanmak için üçlü tek tırnak veya çift
2.  tırnak kullanılır. Açıklama satırını bitirmek için aynı işaretler kullanılır.'''
```

```
In [40]: '''Python'da birden fazla açıklama satırı kullanmak için üçlü tek tırnak
veya çift
    ...: tırnak kullanılır. Açıklama satırını bitirmek için aynı işaretler
kullanılır.'''
Out[40]: "Python'da birden fazla açıklama satırı kullanmak için üçlü tek tırnak
veya çift \ntırnak kullanılır. Açıklama satırını bitirmek için aynı işaretler
kullanılır."
```

Konsol çıktısında açıklamalar birden fazla satırda olduğu için satır sonlarına \n kaçış dizisi karakterini konsol otomatik olarak eklemiştir. Aynı yorum satırını """yorum satırları""" üçlü çift tırnak kullanarak da yapılabilir. Yorum satırları içinde kaçış dizisi karakterlerinin de çalışmadığı unutulmamalıdır.

Yorum satırları süsleme amacıyla da kullanılabilir. Bazı program dosyalarında aşağıdaki gibi süslü açıklamalar, etiketler görülebilir. Bunlar hem kod dosyasını tekrar çalıştırdığımızda bizlere bilgi sunacaktır hem de bizden sonra dosyayı açıp kodlarda düzenlemeler yapacaklar için daha anlamlı bir atmosfer oluşturacaktır.

Değişkenler

Değişken kavramına geçmeden önce değer kavramını tanıyalım. Bu noktaya kadar yaptığımız alıştırmalarda sayısal ya da karakter değerler üzerinden işlem yaptık ya da onları print komutu aracılığıyla yazdırdık. Değerleri bir programda iş yapan/yer alan en temel birimler olarak düşünebiliriz. Bir sayı ya da harf bunlara örnek olarak verilebilir. Aslında kullandığımız "her şeyi" değer olarak nitelendirebiliriz.

Kod yazarken sadece sabit değerler üzerinden işlemler yapılmaz. Kullanıcıdan veya başka kaynaklardan veri alınması gerekir. Örneğin, kullanıcıya ismiyle "merhaba" diyeceğimiz bir kod yazmak istersek her kullanıcıdan ismini girdi olarak almamız gerekir. İşlem yapabilmek için bu girdiler (değerler) bellekte tutulmalıdır. Bu girdileri depolamak amacıyla bellekte belirli bir yer ayrılması gerekir. Bir değişken tanımlandığında yorumlayıcı, veri türüne bağlı olarak bellekte yer ayırır ve ayrılan bölümde hangi türden verilerin saklanabileceğini belirler. Değişkenler bir isim, sayı veya farklı türdeki bir veri için bellekte ayrılan bu yeri temsil eder.

Değişkenlere farklı veri tiplerinde değerler atanabilir. Değer atama ile (bir değişkeni bir değere eşleyerek) tam sayı, ondalık sayı, dizi veya karakter dizisi türünde değerler değişkenlerde tutulabilir. Python, bu konuda çok esnektir. Python'da değişkenlerinin veri tiplerini açıkça bildirmeye gerek yoktur. Aynı değişken farklı veri tiplerinde değerler alabilir. Aynı değişkene önce sayı, sonra bir metin daha sonra başka türde bir değer atanabilir. Bir değişkene değer atandığında veri tipi otomatik olarak tanımlanır.

Eşittir operatörü "=" değişkenlere değer atamak için kullanılır. "=" Operatörünün solunda değişkenin adı ve "=" operatörünün sağında ise bu değişkene atanacak değer yer alır.

Değişken oluşturma ve değer atamaya ilişkin örnekler:

```
1.  num1=5
2.  #num1 değişkenine 5 sayısı atandı.
3.  print('Değişkenin içindeki sayı: ', num1)
4.  num1=10
5.  print('Değişkenin içindeki sayı: ', num1, 'oldu')
6.  num1='john'
7.  print ('Değişkenin içindeki değer: ', num1, 'oldu')
8.  num1=10.5
9.  print ('Değişkenin içindeki sayı: ', num1, 'oldu')
```

```
In [42]: num1=5
    ...: #num1 değişkenine 5 sayısı atandı.
    ...: print('Değişkenin içindeki sayı: ', num1)
    ...: num1=10
    ...: print('Değişkenin içindeki sayı: ', num1, 'oldu')
    ...: num1='john'
    ...: print ('Değişkenin içindeki değer: ', num1, 'oldu')
    ...: num1=10.5
    ...: print ('Değişkenin içindeki sayı: ', num1, 'oldu')
Değişkenin içindeki sayı:  5
Değişkenin içindeki sayı:  10 oldu
Değişkenin içindeki değer:  john oldu
Değişkenin içindeki sayı:  10.5 oldu
```

Örnekte aynı değişkene hem karakter dizisi (string) hem de sayısal değerler atanmıştır.

Farklı şekilde atamalar yapılabilir. Değişkenler aralarına virgül eklenerek yan yana yazılır. Değerleri de aynı sıralama ile karşılarına yazılır. Örnekte olduğu gibi aynı satırda birden fazla atama işlemi yapılabilir.

```
1.  x,y,z = 3, 5 ,7
2.  print(x)
```

```
In [45]: x,y,z = 3, 5 , 7
    ...: print(x)
3
```

Aynı değer birden fazla değişkene de atanabilir.

```
1.  a = b = c = 1
2.  print ('1. sayı=', a)
3.  print ('2. sayı=', b)
4.  print ('3. sayı=', c)
```

```
In [46]: a = b = c = 1
    ...: print ('1. sayı=', a)
    ...: print ('2. sayı=', b)
    ...: print ('3. sayı=', c)
1. sayı= 1
2. sayı= 1
3. sayı= 1
```

Örnek

Değişkenlere değer atamak için başka bir yöntem aralarına noktalı virgül ";" ekleyerek değişken- değer ikilileri şeklinde yazmaktır.

```
1.  adı='Aziz'; soyadı='SANCAR'; yaşı=73
2.  print ("Adı=", adı)
3.  print ("Soyadı=", soyadı,)
4.  print ("Yaşı=", yaşı)
```

```
In [48]: adı='Aziz'; soyadı='SANCAR'; yaşı=73
    ...: print ("Adı=", adı)
    ...: print ("Soyadı=", soyadı,)
    ...: print ("Yaşı=", yaşı)
Adı= Aziz
Soyadı= SANCAR
Yaşı= 73
```

Örnek

Değer atanmayan ve/veya tanımlanmamış bir değişken kullanılırsa Python hata verir.

```
1.  print(ilkdefatanimlanandegisken)
```

```
In [49]: print(ilkdefatanimlanandegisken)
Traceback (most recent call last):

  File "<ipython-input-49-335282771322>", line 1, in <module>
    print(ilkdefatanimlanandegisken)

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

Değişkenler veri tiplerine göre kullanılmazsa Python hata verir.

Örnek:

```
1.  sayi1=1
2.  metin1='deneme'
3.  print(sayi1+metin1)
```

```
In [50]: sayi1=1
    ...: metin1='deneme'
    ...: print(sayi1+metin1)
Traceback (most recent call last):

  File "<ipython-input-50-766a13f56f8e>", line 3, in <module>
    print(sayi1+metin1)

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Değişken Adlandırmadaki Kurallar

Değişken adı verilirken uyulması gereken bazı kurallar ve kurallar kadar katı olmasa da yararlı kullanım önerileri vardır.

Değişken adlandırma kuralları:

- Değişkenler bir harf (a- z, A- Z) veya alt çizgi (_) ile başlamalıdır. Bunların dışında sayı veya başka bir karakter ile de başlayamaz.

- Değişken adında rakam, alt çizgi (_), büyük veya küçük harf olabilir.
- Değişken adları herhangi bir uzunlukta olabilir.

Python, değişken adlandırmada Türkçe karakterlerin (ç, ğ, ı, ö, ş ve ü) kullanımına izin vermektedir.

Ek olarak Python programlama dilinde ayrılmış sözcükler kullanılamaz. Bu özel sözcüklerin listesini görmek için aşağıdaki kod kullanılabilir.

```
In [51]: import keyword
    ...: keyword.kwlist
```

```
1.  import keyword
2.  keyword.kwlist
```

['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']

Not: Ayrıca büyük harf ve küçük harf kullanılarak tanımlanan değişkenlerin adı aynı olsa bile farklı değişkenler olduğu unutulmamalıdır.

Python dilinde uygun değişken adı örnekleri:

```
1.  #Uygun değişken isimleri
2.  sayi1=1
3.  Sayi1=2
4.  #Sayi1' ekrana yazdırılırsa çıktı ne olur?
5.  print (sayi1)
6.  print(Sayi1)
```

Yorum: Büyük harf ve küçük harf kullanarak tanımlanan değişkenlerin adı aynı olsa bile farklı değişkenler olduğu unutulmamalıdır.

```
1.  sayı1=3
2.  #Python  değişken adlandırmada Türkçe karakter kullanımına izin vermektedir.
3.  print(sayı1)
```

```
In [55]: '''Python dilinde uygun değişken adı örnekleri:
    ...: Uygun değişken isimleri'''
    ...: sayi1=1
    ...: Sayi1=2
    ...: print (sayi1)
    ...: print(Sayi1)
    ...: '''Büyük harf ve küçük harf kullanarak tanımlanan değişkenlerin adı aynı
olsa bile
    ...: farklı değişkenler olduğunu unutulmamalıdır.'''
    ...: sayı1=3
    ...: #Python  değişken adlandırmada Türkçe karakter kullanımına izin
vermektedir.
    ...: print(sayı1)
1
2
3
```

Örnek: Python'da hatalı değişken adı kullanımı örneği:

```
1.  1sayi=5 #Hatalı değişken adı.
2.  print (1sayi)
```

```
In [56]: 1sayi=5 #Hatalı değişken adı.
    ...: print (1sayi)
  File "<ipython-input-56-d2f85b40b7c4>", line 1
    1sayi=5 #Hatalı değişken adı.
       ^
SyntaxError: invalid syntax
```

Değişken Adlandırma için Standartlar/Stiller

Değişken adlandırma için bazı standartlar/stiller vardır. Bu standartlar değişken adının ve içeriğinin anlaşılmasına yardımcı olarak programcıların daha kolay çalışmasını sağlar. Değişken adı, onun içeriği hakkında ne kadar çok bilgi verirse kodun anlaşılması kolaylaşır. Değişken adlarının yazımında bazı standart kullanımlar vardır. Bu standart kullanımlarla ile kullanıcılar için daha akışkan bir program yazabilirler.

Birden fazla kelimenin kullanılacağı değişken adlarında kelimelerin ilk harfi büyük olabilir.

Camel (Deve) stilinde (başka stiller de bulunmaktadır) değişken adlarının görünüşü deve hörgücüne benzetilmektedir. Değişkenin adına küçük harfle başlanır ve sonraki her kelime büyük harfle başlar.

Ör: HelloWorld / helloWorld

Snake (Yılan) stilinde ise kelimeler arasında alt çizgi kullanılarak bir yılan görünümü verilmeye çalışılır.

Ör: hello_world / HELLO_WORLD

Python Veri / Değişken Tipleri

Python programlama dilinde veri tipleri oluşturduğumuz değişkenlerin hangi bellekte ne kadar alan kaplayacağını ve türde olacağını belirlemek için kullanılmaktadır. Python programlama dilinde sık kullanılan veri tipleri bulunmaktadır. Bu veri tipleri:

- **int**: Tam sayı
- **float**: Ondalık sayı
- **str**: metin veya karakter ifadeleri
- **bool**: Mantıksal ifadeler (Doğru ve Yanlış)

en sık kullanılan ve temel olarak görülen veri tipleridir.

Bunların yanında ileriki haftalarda da işleyeceğimiz farklı veri tipleri de vardır. Bunları aşağıdaki özet tablodan gözlemlemek mümkündür.

Veri Tipi	Sınıfı	Açıklama
Integer	int	Tam sayı. (Örnek: 3, 5, 369963)
Float	float	Ondalıklı sayı. (Örnek: 10.45)
Complex	complex	Karmaşık sayılar A + Bj şeklinde kullanılır. (Örnek: 4+5j)
Karakter dizisi (String)	str	Karakter dizilerini (metinleri) göstermek için kullanılır. Çift tırnak veya tek tırnak içinde gösterilir. "Merhaba Dünya"
Boolean	bool	Sadece True veya False değeri alır. int(True)=1 iken int(False)=0 dır.
Liste	list	Farklı veri türleri içerebilir. listem=['Çınar', 24, 'Mühendis', True]
Demet (tuple)	tuple	Farklı veri türleri içerebilir. demet1=('Çınar', 24, 'Mühendis', True)
Sözlük (dictionary)	dict	Farklı veri türleri içerebilir. sozluk={'adi': 'Çınar','yasi'=24, 'meslekUnvani':'Mühendis', 'askerlikDurumu': True}

Bilgi: Python programlama dili değişkene atanan değere göre veri türünü **kendisi** otomatik olarak **algılamaktadır**.

Sayısal (int, float ve complex) Veri Tipleri

" "int veri tipi Python'da tam sayıların tutulduğu veri tipidir: 3, 5, 198763 gibi değerleri tutar. En çok kullanılan veri tiplerinden biridir.

"float" veri tipi ondalık sayıların tutulduğu veri tipidir: 0.5, 234678.67 gibi değerleri tutar.

"complex" veri tipi ise karmaşık sayıların tutulduğu veri tipidir. A+Bj tipinde veriler tutulur: 4+5j gibi değerleri tutar.

int Veri Tipi

Tam sayıların içerisinde tutulduğu veri tipi'dir. En çok kullanılan veri türlerinden biridir. Herhangi bir sayısal ifade atanmak istendiğinde genellikle int veri türü kullanılmaktadır. Örneğin: Değişkene 5, 15, 1453, 37 gibi değerler girilebilmektedir. Bu değerler girildiğinde programlama dili otomatik olarak **int** türünde olduğunu anlayacaktır. İngilizce olarak tam sayı kelimesinin karşılığı olan integer kelimesinin kısaltılması şeklinde ifade edilmesi bizlere ipucu sağlar.

int Veri Tipi Kullanımı:

Python programlama dilinde atanan değerin tipi otomatik olarak algılandığı için tam sayı bir değer girildiğinde verinin tipi otomatik olarak **int** olacaktır.

```
1.  sayi=15
2.  #sayi isimli değişkenin içierisine 15 değeri atandığında tamsayı tipinde bir veri
    olacaktır
3.  type(sayi)
```

float Veri Tipi

Ondalık sayıların içerisinde tutulduğu veri tipidir. Örneğin: 15.5, 37.7 şeklinde

float Veri Tipi Kullanımı:

Python programlama dilinde atanan değerin tipi otomatik olarak algılandığı için ondalık sayılı bir değer girildiğinde verinin tipi otomatik olarak **float** olacaktır.

```
1.  osayi=37.7
2.  #osayi isimli değişkene 37.7 değeri yani ondalık bir değer girildiğinde tipi float
    olacaktır.
```

Python'da bir değişkene değer atandığında veri tipleri atanan değere göre otomatik olarak belirlenir.

Örnek:

```
1.  pisayisi=3.14 #float tipinde bir veri
2.  print ('pi sayısı=', piSayisi)
3.  r=2 #integer tipinde veri
4.  alan=pisayisi*r**2
5.  print ('Alan=', alan)
6.  print('Yarıçapı 2 olan dairenin alanı ', alan, ' cm 2 dir' )
7.  karmasikSayi=4+5j
8.  print('Bir karmaşık sayı=', karmasikSayi+3j)
```

```
In [59]: pisayisi=3.14 #float tipinde bir veri
    ...: print ('pi sayısı=', pisayisi)
    ...: r=2 #integer tipinde veri
    ...: alan=pisayisi*r**2
    ...: print ('Alan=', alan)
    ...: print('Yarıçapı 2 olan dairenin alanı ', alan, ' cm 2 dir' )
    ...: karmasikSayi=4+5j
    ...: print('Bir karmaşık sayı=', karmasikSayi+3j)
pi sayısı= 3.14
Alan= 12.56
Yarıçapı 2 olan dairenin alanı  12.56  cm 2 dir
Bir karmaşık sayı= (4+8j)
```

Karakter Dizisi (string) Veri Tipi

Karakter dizisi, kullanıcıdan alınan değerlerin metin formatında tutulduğu veri tipleridir. Python karakter dizisi oldukça kullanışlı işlevlere sahiptir.

Örnek

Bir karakter dizisi ekrana yazdırılabilir, başka bir karakter dizisiyle birleştirilebilir.

"len( )" fonksiyonu bir karakter dizisinin uzunluğunu vermektedir.

```
1.  metin1 = 'Merhaba '
2.  metin2 = 'Mars'
3.  print (metin1)            # karakter dizisinin tamamını yazar
4.  print (metin1 * 2)        # karakter dizisini 2 defa yazar
5.  print (metin1 + metin2)   # iki karakter dizisini birleştirir
6.  print ('metin1  adlı değişkendeki değerin uzunluğu:', len(metin1))
7.  #Boşluğun da bir karakter olduğunu gözden kaçırmayınız.
```

```
In [60]: metin1 = 'Merhaba '
    ...: metin2 = 'Mars'
    ...: print (metin1)            # karakter dizisinin tamamını yazar
    ...: print (metin1 * 2)        # karakter dizisini 2 defa yazar
    ...: print (metin1 + metin2)   # iki karakter dizisini birleştirir
    ...: print ('metin1  adlı değişkendeki değerin uzunluğu:', len(metin1))
    ...: #Boşluğun da bir karakter olduğunu gözden kaçırmayınız.
Merhaba
Merhaba Merhaba
Merhaba Mars
metin1  adlı değişkendeki değerin uzunluğu: 8
```

Karakter Dizilerinde (string) Dilimleme İşlemleri

Bir karakter dizisinin içindeki karakterlere tek tek veya belirli bir aralıkta erişilebilir. Köşeli parantez içinde [] tek bir sayı verildiğinde bu karakter dizisinin indisini ifade eder. İndis numarası "0" dan başlayarak karakterin metindeki kaçıncı sırada yer aldığını gösterir. Metin [0] ifadesi metin değişkenindeki 1. karakteri verir. Belirli bir aralıktaki karakteri alırken "[başlangıç indisi: bitiş indisi]" şeklinde ifade edilir. Metin [0:5] metin değişkeninde indisi 0, 1, 2, 3 ve 4 olan karakterleri dilimler. Bitiş indisi dilimlemeye dahil edilmez.

İndislerin "0" dan başladığı unutulmamalıdır.

Başlangıç indisi verilmeyen karakter dizisinde örnek: metin [:7] başlangıç indisi otomatik olarak sıfır (0) olur. Örnek 0, 1, 2, 3, 4, 5 ve 6. karakterlerden oluşan bir metin verir. Bitiş indisi değeri verilmezse başlangıç indisinden başlanarak son karakter dâhil dilimleme işlemi yapılır.

Karakter dizilerinin kullanımı ile ilgili örnekler:

```
1.  metin='Merhaba Mars'
2.  print (metin[0])
3.  # ifadenin ilk karakterini yazar.
4.  print (metin[4:7])  # ifadenin 5, 6 ve 7. karakterlerini yazar.
5.  print (metin[8::])  # 9. karakterden sonuncu karaktere kadar yazar.
6.  print (metin[-2])   # karakter dizisinin en sondan ikinci karakterini yazar.
7.  print (metin [:7])   # indisi 0' dan 7'ye kadar olan (7 dahil değil) karakterleri yazar.
8.  print (metin[8:])   # başlangıç indisinden sonra tüm karakterleri yazar.
9.  print(metin[0:8:2]) # 0, 2, 4 ve 6 indis numaralı karakterleri dilimler.
```

```
In [61]: metin='Merhaba Mars'
    ...: print (metin[0])
    ...: # ifadenin ilk karakterini yazar.
    ...: print (metin[4:7])  # ifadenin 5, 6 ve 7. karakterlerini yazar.
    ...: print (metin[8::])  # 9. karakterden sonuncu karaktere kadar yazar.
    ...: print (metin[-2])   # karakter dizisinin en sondan ikinci karakterini
yazar.
    ...: print (metin [:7])   # indisi 0' dan 7'ye kadar olan (7 dahil değil)
karakterleri yazar.
    ...: print (metin[8:])   # başlangıç indisinden sonra tüm karakterleri yazar.
    ...: print(metin[0:8:2]) # 0, 2, 4 ve 6 indis numaralı karakterleri dilimler.
M
aba
Mars
r
Merhaba
Mars
Mraa
```

bool Veri Tipi

bool veri tipi , mantıksal bir veri tipidir. İçerisine true ya da false değeri alabilmektedir. Bir ifadenin doğruluğu veya bir döngünün kontrolü için kullanılabilmektedir.

bool Veri Tipi Kullanımı

Python programlama dilinde atanan değerin tipi otomatik olarak algılandığı için true ya da false girildiğinde verinin/değişkenin tipi **str** olacaktır.

```
1.  kontrol=true
2.  #kontrol isimli değişkene true değeri girildiğinde otomatik olarak bool veri tipi
    olmaktadır.
```

type( ) Kullanımı

Python, her ne kadar veri tiplerini otomatik olarak verse de bir değişkenin veri tipini kontrol etmek ve kullanım amacına göre değiştirmek gerekebilir. Bir değişkenin veri tipini öğrenmek için "type()" komutu kullanılır.

Ör: Bir değişkenin veri tipi type() komutuyla kontrol edilir. type (degisken_adı) şeklinde yazılır.

```
1.  sayi1=5
2.  sayi2=10.556
3.  metin1="1"
4.  #metin1 değişkenine tırnak içinde verilen sayının string tipinde bir değişken
5.  olduğuna dikkat ediniz.
6.  sayi3=4+5j
7.  askerlikyaptimi=True
8.  #True doğru, 1, evet anlamındadır.
9.  print ("1. değişkenin veri tipi: ", type(sayi1))
10. print ("2. değişkenin veri tipi: ", type(sayi2))
11. print ("3. değişkenin veri tipi: ", type(metin1))
12. print ("4. değişkenin veri tipi: ", type(sayi3))
13. print ("5. değişkenin veri tipi:", type(askerlikYaptiMi))
```

```
In [66]: sayi1=5
    ...: sayi2=10.556
    ...: metin1="1"
    ...: #metin1 değişkenine tırnak içinde verilen sayının string tipinde bir
değişken
    ...: #olduğuna dikkat ediniz.
    ...: sayi3=4+5j
    ...: askerlikYaptiMi=True
    ...: #True doğru, 1, evet anlamındadır.
    ...: print ("1. değişkenin veri tipi: ", type(sayi1))
    ...: print ("2. değişkenin veri tipi: ", type(sayi2))
    ...: print ("3. değişkenin veri tipi: ", type(metin1))
    ...: print ("4. değişkenin veri tipi: ", type(sayi3))
    ...: print ("5. değişkenin veri tipi: ", type(askerlikYaptiMi))
1. değişkenin veri tipi:  <class 'int'>
2. değişkenin veri tipi:  <class 'float'>
3. değişkenin veri tipi:  <class 'str'>
4. değişkenin veri tipi:  <class 'complex'>
5. değişkenin veri tipi:  <class 'bool'>
```

Veri Tiplerini Dönüştürmek

Tam sayılarla işlem yapıldığında integer, kesirli sayılarla işlem yapıldığında float veri tipi kullanılmaktadır. Değerler üzerinde işlem yaparken içinde sadece rakamlar bulunan string ifadeyi sayısal veri tipine dönüştürmek, bazen de bunun tersini yapmak gerekebilir. Bunun için bazı fonksiyonlar bulunmaktadır. Veri tipini dönüştürmek için kullanılan temel fonksiyonlar şunlardır:

- int( ) : Değişkenin veri tipini integer'a(tam sayıya) çevirir.
- float( ) : Değişkenin veri tipini float'a (kayan ondalıklı sayıya) çevirir.
- str( ) : Veri tipini karakter dizisine çevirir.
- complex() : Karmaşık sayı yazılmasını sağlar.

"integer" tipinde bir sayıyla" float" tipinde bir sayı çarpıldığında sonuç" float" tipinde olacağı için Python bu veri tipini otomatik olarak belirler.

Örnek

Aşağıdaki örnekte iki sayının da tırnak içinde verilmiş olduğuna dikkat ediniz.

```
1.  metin1='5'
2.  metin2='3'
3.  print (metin1+metin2)
```

```
In [67]: metin1='5'
    ...: metin2='3'
    ...: print (metin1+metin2)
53
```

Yukarıda kullanılan değerler tırnak içinde verildiğinden karakter dizisi veri tipindedir. Kod sonuç olarak iki karakteri yan yana yazacaktır.

Örnek

Yukarıdaki örnekteki iki değişkenin veri tipini dönüştürelim. Veri tipi dönüştürüldüğünde karakter dizisi sayıya çevrilmiş olacaktır. Böylece iki sayısal değer üzerinde toplama işlemi yapılabilir.

```
1.  print(int(metin1)+int(metin2))
```

```
In [68]: print(int(metin1)+int(metin2))
8
```

Karakter dizisi olan bir değer sayısal bir ifadeye dönüştürüldüğünde bu ifadenin sadece sayısal karakterler içermesi gerekir. "a" harfi veya karakter dizisi bir ifade int("a") kullanılarak sayıya çevrilemez.

Veri tipi, kullanılan değerler ile uyumlu olmalıdır. Veri tipi dönüşümünde bazı değerlerde kayıplar olabilir.

Örnek:

Aşağıdaki örnekte pi değerinin float ve integer veri tiplerinde kullanıldığında çıkan sonuçları kıyaslayalım.

```
1.  pi_sayısı=3.14 #float tipinde bir veri
2.  print ('Veri Tipi: ',type(pi_sayısı) )
3.  r=2 #integer tipinde veri
4.  alan=pi_sayısı*r**2
5.  print('Dairenin Alanı (float)=', alan)
6.  pi_sayısı_int=(int(pi_sayısı))
7.
8.  print('Int veri tipine dönüştürülen pi değeri: ', pi_sayısı_int)
9.  alan2=((pi_sayısı_int*2)*r)
10. print('Dairenin Alanı (int)=', alan2)
```

```
In [49]: pi_sayısı=3.14 #float tipinde bir veri
    ...: print ('Veri Tipi: ',type(pi_sayısı) )
    ...: r=2 #integer tipinde veri
    ...: alan=pi_sayısı*r**2
    ...: print('Dairenin Alanı (float)=', alan)
    ...: pi_sayısı_int=(int(pi_sayısı))
    ...:
    ...: print('Int veri tipine dönüştürülen pi değeri: ', pi_sayısı_int)
    ...: alan2=((pi_sayısı_int*2)*r)
    ...: print('Dairenin Alanı (int)=', alan2)
Veri Tipi:  <class 'float'>
Dairenin Alanı (float)= 12.56
Int veri tipine dönüştürülen pi değeri:  3
Dairenin Alanı (int)= 12
```

Operatörler

Python'da aritmetik, mantıksal işlemler ve ilişkisel karşılaştırmalar gibi işlemler yapmak için operatör (işleç) adı verilen semboller ve özel sözcükler kullanılır. Bu bölümde Python'da kullanılan temel operatörler yer almaktadır.

Aritmetik operatörler

Toplama, çıkarma, çarpma ve bölme gibi işlemler başta olmak üzere aritmetik işlemleri yapmak için kullanılan operatörlerdir. Aşağıdaki tabloda Temel Aritmetik Operatörler derlenmiştir.

İşaret	İşlem	Örnek	Sonuç
+	Toplama	2+4	6
-	Çıkarma%	5-2	3
*	Çarpma	4*3	12
/	Bölme	12/4	3.0
**	Kuvvet	2**3	8
//	Tam Sayı Bölme	5//4	1
%	Mod(Kalan)	5%4	1

Toplama operatörü ile ilgili örnekler:

Sayısal değerler “integer” “float” veya “complex ”tipinde olabilir.

```
1.  print (5+3)
2.  print (5+3+3)
3.  sayi1=10
4.  print (sayi1+5)
5.  #Aynı şekilde değişken kullanarak da yapabiliriz.
6.  sayi2=10.34
7.  print (sayi1+sayi2+5.5) #farklı veri tiplerindeki sayıları ve değişkenleri de
    kullanabiliriz.
```

```
In [52]: print (5+3)
    ...: print (5+3+3)
    ...: sayi1=10
    ...: print (sayi1+5)
    ...: #Aynı şekilde değişken kullanarak da yapabiliriz.
    ...: sayi2=10.34
    ...: print (sayi1+sayi2+5.5) #farklı veri tiplerindeki sayıları ve
değişkenleri de kullanabiliriz
8
11
15
25.84
```

Toplama operatörünün karakter dizilerinde farklı bir kullanımı vardır: İki veya daha fazla karakter dizisini birleştirmek için kullanılabilir.

```
1.  # toplama operatörü str verileri için de onları bir araya getirmek ve bileştirmek
2.  # amacıyla kullanılmaktadır.
3.  print ("umut"+"yamak")
4.  print ("umut "+" yamak") # boşluğun da bir karekter olduğunu hatırlayalım.
5.  text1 = "isim"
6.  text2 = "soyisim"
7.  text = text1 + text2
```

```
In [54]: print ("umut"+"yamak")
umutyamak

In [55]: print ("umut "+" yamak")
umut  yamak

In [56]: text1 = "isim"
    ...: text2 = "soyisim"
    ...: text = text1 + text2

In [57]: text
Out[57]: 'isimsoyisim'
```

Çarpma Operatörü

Çarpma operatörü iki veya daha fazla sayıyı çarpmak için kullanılır.

Örnek

```
1.  print(5*3)
2.  print(5*3*2)
3.  sayi1=5
4.  sayi2=3
5.  print(sayi1*sayi2)
```

```
In [58]: print(5*3)
    ...: print(5*3*2)
    ...: sayi1=5
    ...: sayi2=3
    ...: print(sayi1*sayi2)
15
30
15
```

Bir karakter dizisini istediğiniz kadar yazdırmak için çarpma operatörünü kullanabilirsiniz.

Örnek

```
1.  20*'BA'
```

```
In [80]: 20 * "BA"
Out[80]: 'BABABABABABABABABABABABABABABABABABABA'
```

Bölme Operatörü

Bu operatör, iki sayıyı bölmek için kullanılır.

Örnek

```
1.  5/3
```

```
In [119]: 5/3
Out[119]: 1.6666666666666667
```

Bölme operatörünü ikiden fazla sayıyı bölmek için de kullanabilirsiniz.

Örnek

```
1.  18/3/2
```

```
In [120]: 18/3/2
Out[120]: 3.0
```

Bu tür durumlarda Python işlemleri soldan başlayarak sırayla yapar. Önce 18/3 işlemini yapar ve sonucu 2'ye böler.

Kuvvet Alma Operatörü

Bir sayının kuvvetini almak için kullanılır. Bir sayının kuvveti o sayının kendisiyle kuvvet kadar çarpılmasıyla bulunur.

5**3=5*5*5

Örnek

```
1.  #5'in 3. kuvvetinin bulunması:
2.  5**3
```

```
In [121]: 5**3
Out[121]: 125
```

Kuvvet alma operatörü bir sayının kökünü almak için "1/kuvvet" olarak kullanılabilir.

Örnek

```
1.  #49'un karekökünü almak için 1/2 kuvveti alınır.
2.  49**(1/2)
```

```
In [133]: 49**(0.5)
Out[133]: 7.0
```

Tam Bölüm Operatörü

İki sayının birbirine tam bölüm sonucunu verir bu nedenle sadece bölümü görebiliriz. Kalan ya da ondalık kısımları da görebilmek için klasik bölme operatörü kullanılmalıdır.

Örnek

```
1.  print (121.00//3) # kullanılan sayı float olduğu için sonuç da float olarak çıktı verir.
2.  print (121//3)
```

```
In [67]: print(121.//3)
40.0

In [68]: print(121//3)
40
```

Mod Alma Operatörü

Bir sayının diğer bir sayıya bölümünden kalanını verir.

Örnek

```
1.  print (5%3) #Sayının 3 ile bölümünden kalanı yansıtır.
2.  print (9%2) # İkiye tam bölünmeyle ilgili olması nedeniyle kalan 0 ise sayı çifttir.
```

```
In [71]: print (5%3)
    ...: print (9%2)
2
1
```

Not: Aritmetik işlemlerde olduğu gibi, bir ifadenin hesaplanmasında öncelikle parantez içindeki kısımlar hesaplanır.

İlişkisel Operatörler

İlişkisel operatörler, değerler arasındaki ilişkiyi kontrol ederek sonucu " boolean" bir değer olarak (True, False) döndürür. "True" değeri şartın, ilişkinin veya koşulun sağlandığı anlamına gelirken, " False" değeri ise ilişkinin sağlanmadığı anlamına gelir.

İşaret	İşlem	Örnek	Sonuç
= =	Eşit mi?	5 == 3	False
!=	Farklı mı?	5!=2	False
>	Büyüktür?	4>3	True
>=	Büyük veya eşittir?	12>=4	True
<=	Küçük veya eşittir?	5<=3	False
is	Değerler eşit mi?	"umut"is"Umut"	False
is not	Değerler farklı mı?	"umut"is not"Umut"	True
in	İçeriyor mu?	"hell" in "hello"	True
not in	İçermiyor mu ?	"hell" in "hello"	False

Eşittir Operatörü

"==" operatörü iki değerin birbirine eşit olup olmadığını anlamak için kullanılır. İki değer birbirine eşitse "True", eşit değilse "False" değeri verir.

Örnek

```
1.  print(5==3)
2.  #değişkenlere atadığınız değerleri de aynı şekilde kontrol edebilirsiniz.
3.  sayi1=5
4.  sayi2=3
5.  print(sayi1==sayi2)
6.  print(sayi1==5)
```

```
In [72]: print(5==3)
    ...: #değişkenlere atadığınız değerleri de aynı şekilde kontrol
edebilirsiniz.
    ...: sayi1=5
    ...: sayi2=3
    ...: print(sayi1==sayi2)
    ...: print(sayi1==5)
False
False
True
```

Örnek

"==" Operatörü karakter dizilerinde de değerlerin eşitliğini kontrol etmek için kullanılır.

```
1.  print('Emre'=='emre')
2.  # Küçük büyük harf duyarlılığına (case sensetive) dikkat edilmelidir.
3.  metin1='Emre'
4.  metin2='emre'
5.  print(metin1==metin2)
6.  print(metin1=='Emre')
```

Eşit Değildir Operatörü

"!=" operatörü iki değerin birbirinden farklı olup olmadığını anlamak için kullanılır. "==" operatörünün tersine değerler birbirine eşitse" False", eşit değilse" True"ue değerini döndür

Örnek

```
1.  print(5!=3)
2.  sayi1=5
3.  sayi2=3
4.  print(sayi1!=sayi2)
5.  print(sayi1!=5)
6.  # Çıktıların == operatörünün tersi olduğuna dikkat ediniz.
```

```
In [73]: print(5!=3)
    ...: sayi1=5
    ...: sayi2=3
    ...: print(sayi1!=sayi2)
    ...: print(sayi1!=5)
    ...: # Çıktıların == operatörünün tersi olduğuna dikkat ediniz
True
True
False
```

Örnek

"! =" operatörü karakter dizilerinde de değerlerin farklılığını kontrol etmek için kullanılır.

```
1.  print('Emre'!='emre')
2.  # Küçük büyük harf duyarlılığına (case sensetive) dikkat ediniz.
3.  metin1='Emre'
4.  metin2='emre'
5.  print(metin1!=metin2)
6.  print(metin1!='Emre')
```

Büyüktür Operatörü

Büyüktür ">" operatörü iki değeri karşılaştırmak için kullanılır. 1. sayı 2. sayıdan büyükse "True" değilse "False" değerini döndürür.

Örnek

```
1.  sayi1=6.06
2.  sayi2=6.07
3.  print(sayi1>sayi2)
4.  False
5.  "sayi2" değişkenine yeni bir değer atandığını gözden kaçırmayınız.
6.  sayi2=6
7.  print (sayi1>sayi2)
8.  sayi2=6.06
9.  print(sayi1>sayi2)
```

Küçüktür Operatörü

Küçüktür "<" operatörü iki değeri karşılaştırmak için kullanılır. 1. sayı 2. sayıdan küçükse "True" değilse "False" değerini döndürür. Büyüktür operatörünün tersi işlevini görür.

Örnek

```
1.  sayi1=6.06
2.  sayi2=6.07
3.  print(sayi1<sayi2)
```

```
In [74]: sayi1=6.06
    ...: sayi2=6.07
    ...: print(sayi1<sayi2)
True
```

Programımıza devam edelim. "sayi2" değişkenine yeni bir değer atadığımızı gözden kaçırmayalım

```
1.  sayi2=6
2.  print (sayi1<sayi2)
3.  sayi2=6.06
4.  print(sayi1<sayi2)
```

```
In [75]: sayi2=6
    ...: print (sayi1<sayi2)
    ...: sayi2=6.06
    ...: print(sayi1<sayi2)
False
False
```

Karakter dizileri, alfabetik olarak sıralandığında sonra gelen ifade daha büyük olarak değerlendirilir.

```
1.  print ('z'>'a')
2.  print ('a'<'z')
```

```
In [76]: print ('z'>'a')
    ...: print ('a'<'z')
True
True
```

Büyük Eşittir (>=) ve Küçük Eşittir (<=) Operatörleri

Büyük eşittir ">=" operatörü iki değeri karşılaştırmak için kullanılır. Birinci değer ikinciden büyükse veya ikinciye eşitse "True" değilse "False" değerini döndürür. Küçük eşittir "<=" operatörü ise birinci değer ikinci değerden küçükse veya ikinci değere eşitse "True" değilse "False" değerini döndürür.

```
1.  #Büyük Eşittir (>=) ve Küçük Eşittir (<=) Operatörleri kullanımı:
2.  sayi1=6.06
3.  sayi2=6.07
4.  print(sayi1>=sayi2)
5.  print (sayi1<=sayi2)
```

```
In [78]: sayi2=6.07
    ...: print(sayi1>=sayi2)
    ...: print(sayi1<=sayi2)
False
True
```

```
1.  #sayılar eşit olduğu için iki operatör de True değeri döndürür.
2.  sayi2=6.06
3.  print(sayi1>=sayi2)
4.  print(sayi1<=sayi2)
```

```
In [77]: sayi1=6.06
    ...: sayi2=6.06
    ...: print(sayi1>=sayi2)
    ...: print (sayi1<=sayi2)
True
True
```

```
1.  #1.sayı 2. sayıdan küçük olduğu için sadece <= operatörü True değerini döndürür.
2.  sayi2=6.05
3.  print(sayi1>=sayi2)
4.  print(sayi1<=sayi2)
5.  #sayi2 değişkeni 6.05 olduğu ve sayi1 sayi2 den büyük olacağı için sadece >= operatörü
    True değeri #döndürür.
```

```
In [79]:
    ...: sayi2=6.05
    ...: print(sayi1>=sayi2)
    ...: print(sayi1<=sayi2)
    ...: #sayi2 değişkeni 6.05 olduğu ve sayi1 sayi2 den büyük
olacağı için sadece >= operatörü True değeri #döndürür.
True
False
```

“is” ve “is not” Operatörleri

“is operatörü” ve “==” operatörü benzer işleve sahiptir ve iki değerin eşit olup olmadığını kontrol etmek için kullanılır. Değerler eşitse “True” değilse “False” değerini döndürür.

“is not” operatörü ise “is” operatörünün tersi işlev görür. “is not” operatörü değerler farklı ise True değerini değerler aynıysa “False” değerini döndürür.

Not: “==” operatörü değerlerin eşitliğini kontrol ederken “is” aynı zamanda her iki değerin aynı nesneyi referans gösterip göstermediğini kontrol eder.

Örnek

```
1.  sayi1=5
2.  print (sayi1  is 5)
3.  print (sayi1 is not 5) #is operatörünün tersini verir.
```

```
In [90]: """ is ve is not operatörü """
    ...: sayi1=5
    ...: print (sayi1 is 5)
    ...: print (sayi1 is not 5) #is operatörünün tersini verir.
<>:3: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:4: SyntaxWarning: "is not" with a literal. Did you mean "!="?
<>:3: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:4: SyntaxWarning: "is not" with a literal. Did you mean "!="?
True
False
C:\Users\USER\AppData\Local\Temp/ipykernel_7092/2676406417.py:3:
SyntaxWarning: "is" with a literal. Did you mean "=="?
  print (sayi1 is 5)
C:\Users\USER\AppData\Local\Temp/ipykernel_7092/2676406417.py:4:
SyntaxWarning: "is not" with a literal. Did you mean "!="?
  print (sayi1 is not 5) #is operatörünün tersini verir.
```

Pythonda is ve is not operatörünün kullanımı == veya != ile benzer işlevleri yerini getirmesi sebebiyle bize bu operatörlerin kullanılması önerilmektedir. Fakat uyarının yanında yine de çıktıları bizlere yansıtır.

“is “ operatörü karakter dizisinde de kullanılabilir.

```
1.  print ('elif' is 'Elif')
2.  #Yukarıdaki kod için büyük harf küçük harf duyarlılığını hatırlayınız.
3.  adi='Elif'
4.  print (adi is 'Elif')
5.  print (adi is not 'Elif') #is operatörünün tersini verir.
```

```
In [91]: print ('elif' is 'Elif')
    ...: #Yukarıdaki kod için büyük harf küçük harf duyarlılığını
hatırlayınız.
    ...: adi='Elif'
    ...: print (adi is 'Elif')
    ...: print (adi is not 'Elif') #is operatörünün tersini verir.
<>:1: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:4: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:5: SyntaxWarning: "is not" with a literal. Did you mean "!="?
<>:1: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:4: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:5: SyntaxWarning: "is not" with a literal. Did you mean "!="?
False
True
False
```

"in" ve "not in" Operatörleri

"in" operatörü bir karakter dizisinin başka bir karakter dizisinde yer alıp almadığını kontrol etmek için kullanılır.

Karakter dizisi, diğer karakter dizisi içinde yer alıyorsa "True" değeri, yer almıyorsa "False" değeri döndürür.

"not in" operatörü ise içinde yer almıyorsa "True" yer alıyorsa" False" değeri döndürür.

Örnek

```
# in operatörü kullanımı
print ('Bil' in 'Bilişim')
# 2. karakter dizisi içinde 1. karakter dizisi var mı?
print ('Bil' not in 'Bilişim')
# in operatörünün tersi sonuç verir.
```

```
In [92]:
    ...: print ('Bil' in 'Bilişim')
    ...: # 2. karakter dizisi içinde 1. karakter dizisi var mı?
    ...: print ('Bil' not in 'Bilişim')
    ...: # in operatörünün tersi sonuç verir.
True
False
```

NOT: "in" operatörünün özellikle for döngüsünde çok işlevsel bir kullanımı vardır ve sayılarla birlikte kullanımına orada tekrar değinilecektir.

Atama Operatörleri

Atama operatörleri değişkene değer atamak için kullanılır. Değişkeni başka bir değerle işleme alarak sonucun yine aynı değişkene atanmasını sağlar.

sayı = 5 değişkeni tanımlanmış olsun. Tablo ile operatörler kullanıldıktan sonra hangi sonucu okuyabileceğimizi görebiliriz.

İşaret	İşlem	Örnek	Sonuç
+ =	Arttırarak atama	sayı += 2	7
-=	Eksilterek atama	sayı -= 2	3
*=	Çarparak atama	sayı *= 3	15
/=	Bölerek atama	sayı /= 3	1.6667
**=	Kuvvet alarak atama	sayı **=3	125
//=	Tam sayı bölerek atama	sayı//=3	1
%=	Mod alarak atama	sayı%=3	2

Artırarak Atama Operatörü

Bir değişkenin üstüne sayısal değerleri toplayarak sonucun aynı değişkene atanmasını sağlar.

Örnek

```
1.  #Aşağıdaki kod "sayi1=sayi1+3" koduyla aynı işlevi görür.
2.  sayi1=5
3.  sayi1+=3
4.  print (sayi1)
5.  metin1='Merhaba '
6.  metin1+='Mars' #metin1=metin1+\Mars\ koduyla aynı işlevi görür.
7.  print(metin1)
```

```
In [94]:
    ...:
    ...: sayi1=5
    ...: sayi1+=3
    ...: print (sayi1)
    ...:
    ...: # artırma operatörünün metinlerde
    ...: metin1='Merhaba '
    ...: metin1+='Mars' #metin1=metin1+"Mars" koduyla aynı işlevi
görür.
    ...: print(metin1)
8
Merhaba Mars
```

Eksilterek Atama Operatörü

Bir değişkenden bir sayısal değeri eksilterek sonucun aynı değişkene atanmasını sağlar.

Örnek

```
1.  #Aşağıdaki kod "sayi1=sayi1-3" koduyla aynı işlevi görür.
2.  sayi1=5
3.  sayi1-=3
4.  print (sayi1)
```

```
In [95]: sayi1=5
    ...: sayi1-=3
    ...: print (sayi1)
2
```

Çarparak Atama Operatörü

Bir değişkeni bir sayısal değer ile çarparak sonucun aynı değişkene atanmasını sağlar.

Örnek

```
1.  #Aşağıdaki kod "sayi1=sayi1*3" koduyla aynı işlevi gör
2.  sayi1=5
3.  sayi1*=3
4.  print (sayi1)
5.  #Aynı şekilde karakter dizilerinde de kullanabilirsiniz.
6.  metin1='Merhaba '
7.  metin1*=3 #metin1=metin1*3 koduyla aynı işlevi görür.
8.  print(metin1)
```

```
In [96]:
    ...: sayi1=5
    ...: sayi1*=3
    ...: print (sayi1)
    ...: #Aynı şekilde karakter dizilerinde de kullanabilirsiniz.
    ...: metin1='Merhaba '
    ...: metin1*=3 #metin1=metin1*3 koduyla aynı işlevi görür.
    ...: print(metin1)
15
Merhaba Merhaba Merhaba
```

Bölerek Atama Operatörü

Bir değişkeni bir sayısal değere bölerek sonucun aynı değişkene atanmasını sağlar.

Örnek

```
1.  #Aşağıdaki kod "sayi1=sayi1/3" koduyla aynı işlevi
    görür.
2.  sayi1=5
3.  sayi1/=3
4.  print(sayi1)
```

```
In [97]:
    ...: sayi1=5
    ...: sayi1/=3
    ...: print(sayi1)
1.6666666666666667
```

Kuvvet Alarak Atama Operatörü

Bir değişkenin kuvvetini alarak sonucun aynı değişkene atanmasını sağlar.

```
1.  #Aşağıdaki kod "sayi1=sayi1**3  koduyla aynı işlevi görür
2.  sayi1=5
3.  sayi1**=3
4.  print (sayi1)
```

```
In [98]:
   ...:
   ...: sayi1=5
   ...: sayi1**=3
   ...: print (sayi1)
125
```

Tam Sayı Bölerek Atama Operatörü

Bir değişken sayıya bölündüğünde sonucun (ondalık kısmını atarak) aynı değişkene atanmasını sağlar.

Örnek

```
1.  #Aşağıdaki "kod sayi1=sayi1//3" koduyla aynı işlevi görür.
2.  sayi1=5
3.  sayi1//=3
4.  print (sayi1)
```

```
In [104]:
      ...: sayi1=5
      ...: sayi1//=3
      ...: print (sayi1)
1
```

Mod Alarak Atama Operatörü

Bir değişkenin bir sayıya bölümünden kalanın aynı değişkene atanmasını sağlar.

Örnek

```
1.  #Aşağıdaki kod "sayi1=sayi1%3" koduyla aynı işlevi görür.
2.  sayi1=5
3.  sayi1%=3
4.  print (sayi1)
```

```
In [105]:
      ...: sayi1=5
      ...: sayi1%=3
      ...: print (sayi1)
2
```

Mantıksal Operatörler

İfadeleri mantıksal olarak bağlamak için kullanılan "and", "not" ve "or" operatörleridir. Belirtilen koşullardan birinin sağlanması durumunda "True" değeri döndürür.

"or" Operatörü

"or" operatörü "veya" anlamındadır.

Örnek

Bir sayı 6'dan küçük veya 10'dan büyükse koşulunu düşünelim. Sayımız 5 ise 6'dan küçük olduğu için bu şartı sağlayacaktır. Sayımız 6, 7, 8, 9 veya 10 olursa şartların her ikisini de sağlamadığı için "False" değeri döndürülür.

```
1.  sayi1=5
2.  #or operatörü kullanımı
3.  print (sayi1<6 or sayi1>10)
4.  adi='Elif'
5.  print (adi=='Elif' or adi=='Emre')
6.  #Adı Elif veya Emre ise True değerini döndürür.
7.  meslek='Mühendis'
8.  print (meslek=='Öğretmen' or meslek=='Doktor')
9.  #Meslek Öğretmen veya Doktor olmadığı için False döndürür.
10. print (meslek=='Öğretmen' or meslek=='Doktor' or
11. meslek=='Mühendis')
```

```
In [106]: sayi1=5
     ...: #or operatörü kullanımı
     ...: print (sayi1<6 or sayi1>10)
     ...: adi='Elif'
     ...: print (adi=='Elif' or adi=='Emre')
     ...: #Adı Elif veya Emre ise True değerini döndürür.
     ...: meslek='Mühendis'
     ...: print (meslek=='Öğretmen' or meslek=='Doktor')
     ...: #Meslek ünvanı Öğretmen veya Doktor olmadığı için False
döndürür.
     ...: print (meslek=='Öğretmen' or meslek=='Doktor' or
     ...: meslek=='Mühendis')
     ...: #Meslek Ünvanı Öğretmen veya Doktor veya Mühendis'ten
biri ise True değerini döndürür.
     ...: # İkiden fazla koşul için de kullanılabilir.
True
True
False
True
```

"and" Operatörü

Bu operatör "ve" anlamındadır. Belirtilen koşulların hepsinin sağlanması durumunda "True" değerini döndürür.

Örnek : Bir öğrencinin ders puanı 50'den büyük ve 60'tan küçükse koşulunu düşünüldüğünde öğrencinin puanı 50 ise her iki şartı da sağlayacaktır ve "True" değerini döndürecektir.

```
1.  puan = 50
2.  print (puan>50 and puan<60)
3.  adi='Emre'
4.  yasi=24
5.  print (adi=='Emre' and yasi>=20)
6.  #Adı Emre ve yaşı en az 20 ise True değerini döndürür.
7.  meslek='Mühendis'
8.  askerlik_durumu='Yaptı'
9.  tecrube=2
10. print (meslek=='Mühendis' and askerlik_durumu=='Yaptı')
11. print (meslek=='Mühendis' and askerlik_durumu=='Yaptı' and tecrube>=3)
```

```
In [107]:
     ...: puan = 50
     ...: print (puan>50 and puan<60)
     ...: adi='Emre'
     ...: yasi=24
     ...: print (adi=='Emre' and yasi>=20)
     ...: #Adı Emre ve yaşı en az 20 ise True değerini döndürür.
     ...: meslek='Mühendis'
     ...: askerlik_durumu='Yaptı'
     ...: tecrube=2
     ...: print (meslek=='Mühendis' and askerlik_durumu=='Yaptı')
     ...: print (meslek=='Mühendis' and askerlik_durumu=='Yaptı'
and tecrube>=3)
False
True
True
False
```

"not" Operatörü

Bu operatör "değil" anlamındadır. Belirtilen koşulun tersi doğruysa True değeri verir. Bir sınavda puanı 45'ten küçük değilse ifadesi düşünüldüğünde öğrencinin puanı 50 ise "True" değerini döndürür.

Örnek

```
1.  puan = 50
2.  print(puan<45)
3.  print ( not (puan<45) )
4.  print(puan >= 45) #Yukarıdaki ifade ile aynı işlevi görür.
```

```
In [115]:
     ...: puan = 50
     ...: print(puan<45)
     ...: print ( not (puan<45) )
     ...: print(puan >= 45)
False
True
True
```

Operatörlerde Öncelik Sırası

Farklı operatörler birlikte kullanılabilir. Operatörler birlikte kullanılırken hangi işlemin önce yapılacağına dair bir sıralama vardır.

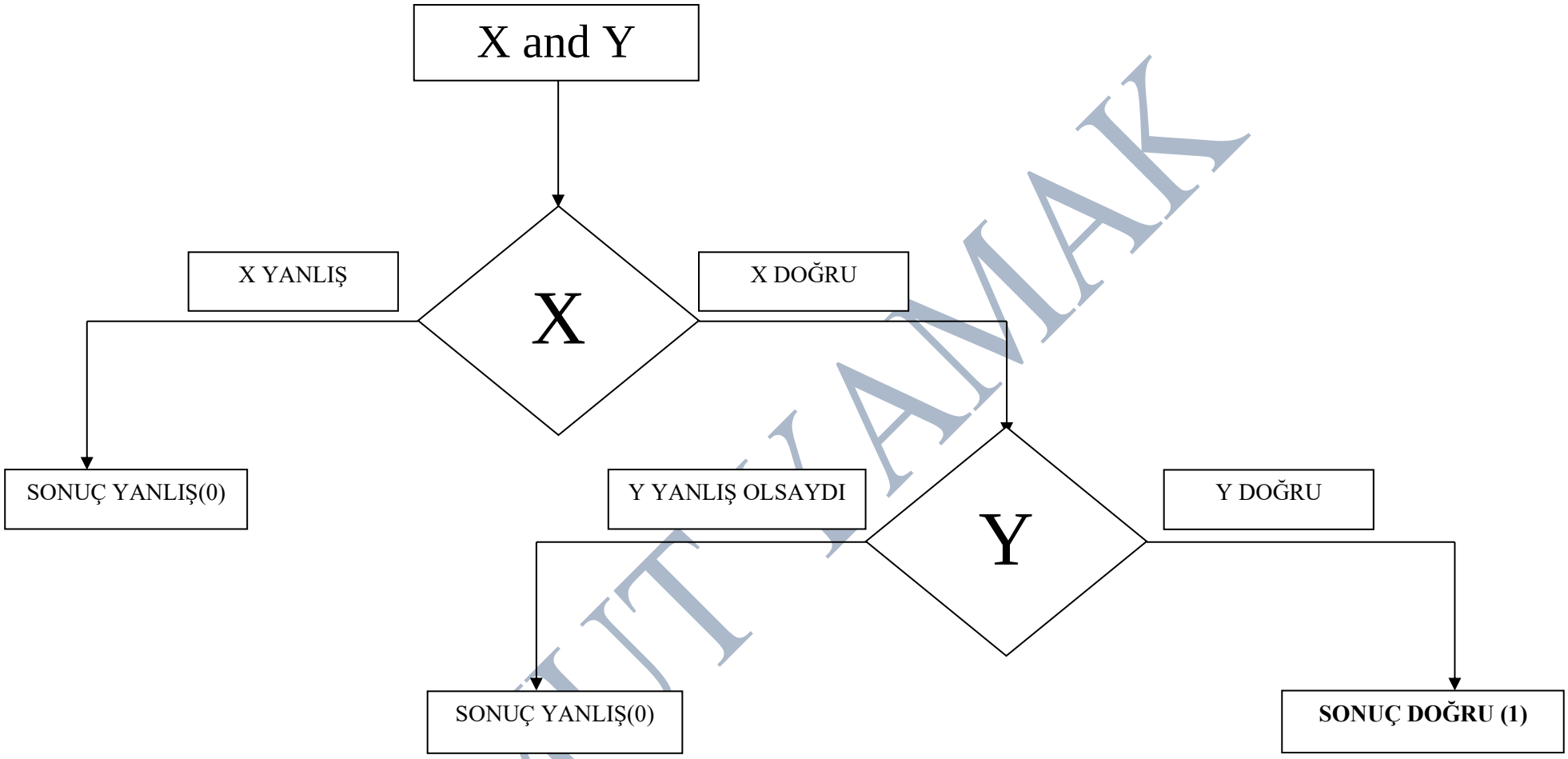
- Parantez içindeki işlemler her zaman öncelikli olarak yapılır.
- Çarpma ve bölme işlemleri toplama ve çıkarma işlemine göre önce yapılır.
- Aynı derecedeki operatörlerde işlem sırası önceliği soldan sağa doğrudur.
- İşlemlerde öncelik sırasını belirtmek için en iyi yöntem operatörleri parantez içinde kullanmaktır.

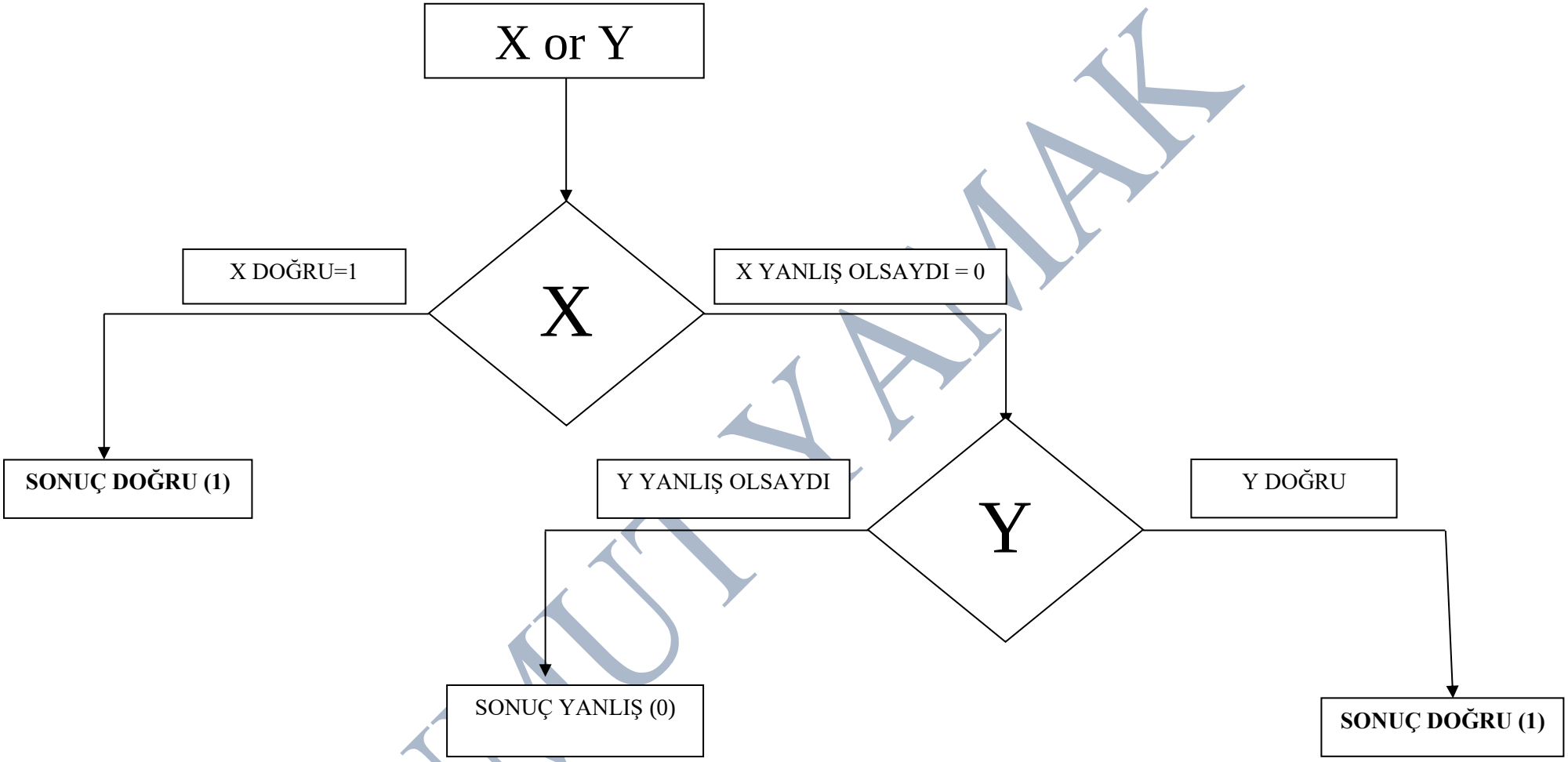
Örnek

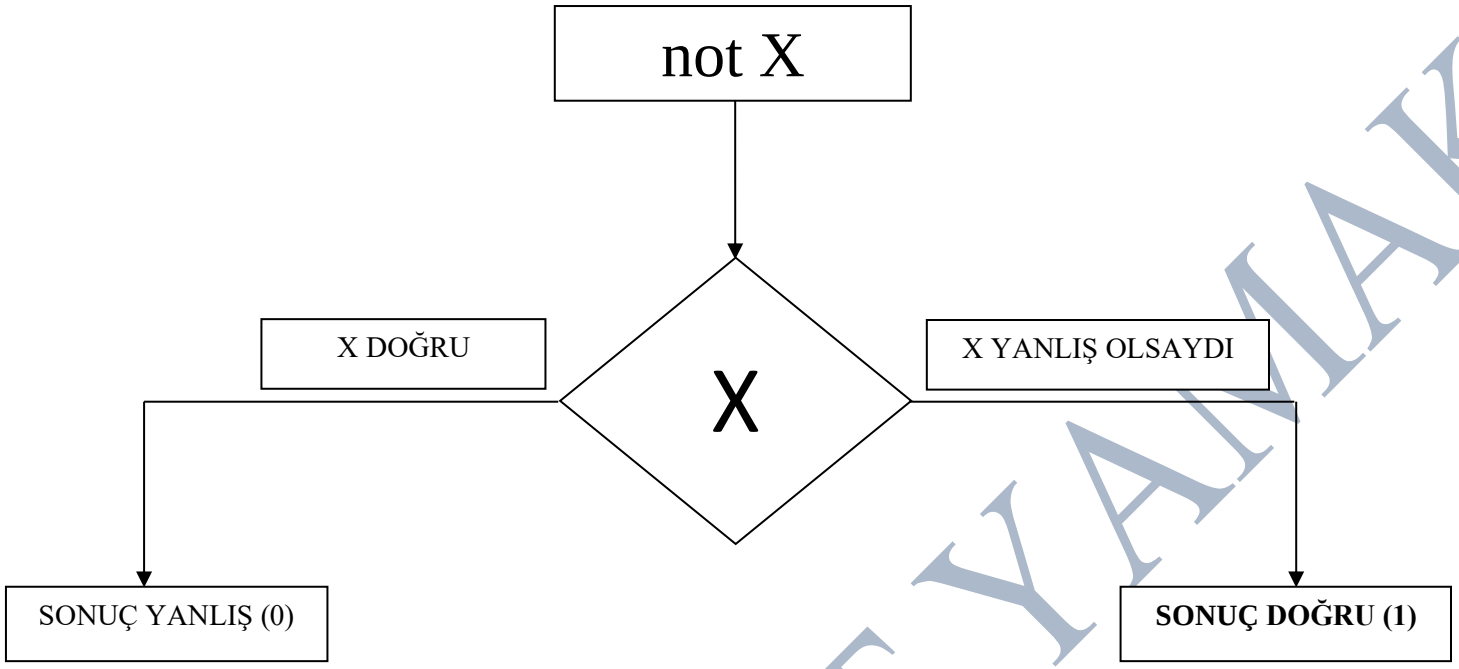
```
1.  print((3+5)*2) #Bu işlemin sonucunu tahmin ediniz.
2.  #Öncelikle parantez içi yapıldığında 8*2=16
3.  print (3+5*2) #Bu işlemin sonucu kaçtır?
4.  #Öncelikle çarpma işlemi yapıldığından 3+10=13
5.  print (3**2*2)
6.  print (6*7/7)
7.  print (6*3/2+8/2*3)
```

```
In [116]:
     ...:
     ...: print((3+5)*2) #Bu işlemin sonucunu tahmin ediniz.
     ...: #Öncelikle parantez içi yapıldığında 8*2=16
     ...: print (3+5*2) #Bu işlemin sonucu kaçtır?
     ...: #Öncelikle çarpma işlemi yapıldığından 3+10=13
     ...: print (3**2*2)
     ...: print (6*7/7)
     ...: print (6*3/2+8/2*3)
16
13
18
6.0
21.0
```

Bu bölümde temel operatörler üzerinde durulmuştur. Python'da bu operatörlerin yanında başka operatörlerde yer almaktadır. Ayrıca kullanılan Python sürümüne göre farklı operatörler kullanılabilir.


X and Y
X
X YANLIŞ
X DOĞRU
SONUÇ YANLIŞ(0)
Y
Y YANLIŞ OLSAYDI
Y DOĞRU
SONUÇ YANLIŞ(0)
SONUÇ DOĞRU (1)


X or Y
X
X DOĞRU=1
X YANLIŞ OLSAYDI = 0
SONUÇ DOĞRU (1)
Y
Y YANLIŞ OLSAYDI
Y DOĞRU
SONUÇ YANLIŞ (0)
SONUÇ DOĞRU (1)



Bu bölümde kullanılan temel operatörler üzerinde durulmuştur. Python'da bu operatörlerin yanında başka operatörlerde yer almaktadır. Bunlardan en önemlisi bitsel operatörlerdir.

Python Bitsel Operatörler

Bitsel operatörler (ikili) sayıları karşılaştırmak için kullanılır.

Operatör	Ad	Açıklama
&	AND	Her iki bit de 1 ise her bit'i 1'e ayarlar
|	OR	İki bitten biri 1 ise, her biti 1 olarak ayarlar.
^	XOR	İki bitten yalnızca biri 1 ise, her biti 1 olarak ayarlar.
~	NOT	Tüm bitleri ters çevirir.
<<	Zero fill left shift	Sıfırları sağdan içeri doğru iterek sola kaydırır ve en soldaki bitlerin düşmesine izin verir.
>>	Signed right shift	En soldaki bitin kopyalarını soldan iterek sağa kaydırır ve en sağdaki bitlerin düşmesine izin verir

Operatörlerde Öncelik Sırası

Farklı operatörler birlikte kullanılabilir. Operatörler birlikte kullanılırken hangi işlemin önce yapılacağına dair bir sıralama vardır.

- Parantez içindeki işlemler her zaman öncelikli olarak yapılır.
- Çarpma ve bölme işlemleri toplama ve çıkarma işlemine göre önce yapılır.
- Aynı derecedeki operatörlerde işlem sırası önceliği soldan sağa doğrudur. Bunlar mantıksal ve ilişkisel operatörlerin sırası için önem arz etmektedir.
- İşlemlerde öncelik sırasının karıştırılması çok olasıdır. Bu nedenle öncelik sırasını belirtmek için en iyi yöntem operatörleri parantez içinde kullanmaktır.Bu yöntem bizleri garanti altına alacaktır.

Pythonda kullanılan tüm operatörlerin işlem öncelik sırası aşağıdaki tabloda özetlenmiştir. Bu bilgiler linteki adresten alınmıştır. https://www.geeksforgeeks.org/precedence-and-associativity-of-operators-in-python/

Operatör	Açıklama
( )	Parantez
**	Kuvvet

Operatör	Açıklama
* / %	Çarpma / Bölme /Mod
+ –	Toplama / Çıkarma
<< >>	Bitsel Sola Kaydır, Bitsel Sağa Kaydır
< <= > >=	Büyük / Küçük Büyük Eşit / Küçük Eşit
== !=	Eşittir / Eşit değildir.
is, is not in, not in	Kimlik operatörü Aitlik operatörü
&	Bitsel Ve(AND)
|	Bitsel Veya(OR)
not	Mantıksal Değil (NOT)
and	Mantıksal Ve
or	Mantıksal Veya
= += -= *= /= %= &= ^= |= <<= >>=	Atama Operatörleri

PYTHON'DA TEMEL FONKSİYONLAR

Kullanıcının yaptıklarını bastırmak ve kullanıcıdan bilgi almak amacıyla kullanılan fonksiyonlar

Programlama dillerinde fonksiyonlardan sıkça faydalanılmaktadır. Fonksiyonları kendimiz yazabileceğimiz gibi, programlama dillerinin kütüphanelerinde bulunan hazır fonksiyonlar da kullanılabilir. Python'da pek çok hazır fonksiyon bulunmaktadır. Bunlara yerleşik fonksiyonlar da denilmektedir. Peki, neden fonksiyonlara ihtiyaç duyulmaktadır? Fonksiyonlar yapılan işlemleri kolaylaştırır ve zaman alıcı işlevleri kolay bir şekilde yerine getirilmesini sağlar. Bunun yanında print() ve input( ) gibi kullanıcıya bir çıktı vermek ve kullanıcıdan girdi almak için kullanılan, programlama dilinin olmazsa olmaz fonksiyonları vardır. Kullanıcıdan bir girdi almadan ya da yapılan işlemlerin sonucunu kullanıcıya vermeden program yazmanın bir anlamı olmaz. Örneğin, kullanıcının vücut kitle indeksini hesaplamak için kullanıcının boy ve kilo verisine ihtiyacımız vardır. Eğer kullanıcıdan bir girdi alınmazsa bu hesaplamayı yapmak mümkün değildir.

Aynı şekilde yapılan işlemin sonucunu görmek için de sonucun ekranda görüntülenmesi, yani programın kullanıcıya çıktı vermesi gerekmektedir. Python'da konsola (etkileşimli kabuk) veriler yazdırmak istendiğinde print( ) fonksiyonu, kullanıcıdan bir girdi almak istendiğinde ise input( ) fonksiyonu kullanılmalıdır. Ayrıca her iki fonksiyonun alacağı parametreler ve/veya beraber kullanılabileceği fonksiyonlar bulunmaktadır. Kullanıcıdan girdi alırken kimi zaman sayısal ifadeler (integer) kimi zaman da metinsel ifadeler (string) istenebilir. Yani programın bir aşamasında kullanıcıdan ismini ya da yaşadığı şehri girmesi, başka bir aşamasında ise yaş verisini alarak bununla ilgili işlem yapılabilir. Peki, Python girilen verinin sayısal değer mi, yoksa metin gibi bir ifade mi olduğunu nasıl anlayacak? İşte bu gibi durumlarda input( ) fonksiyonu ile beraber farklı fonksiyonlar da kullanılabilir.

print( ) Fonksiyonu

print( ) fonksiyonu, konsola çıktı göndermek amacıyla kullanılır. Programların genellikle yapılan işlemler sonucunu kullanıcıya sunması gerekir. Programda veri print( ) fonksiyonu ile Python'daki konsolda görüntülenebilir. print( ) fonksiyonunun kullanımı:

- print
- print( ) parantez açılır
- print("değer")
- print('değer')
- print('''değer''')
- print(tanımlanmış değişken)

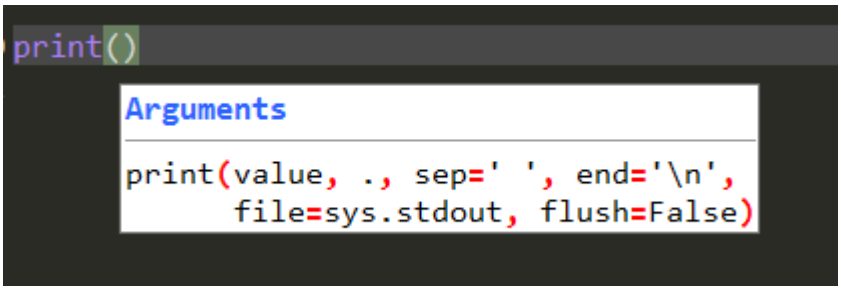
konsola gönderilecek değer bir metin gibi bir ifade ise bu ifade çift tırnak, tek tırnak ya da üç çift tırnak içinde yazılmalıdır.

Her üç kullanım da bize aynı çıktıyı verir. Eğer konsola gönderilecek değer bir değişken ya da sayısal bir ifade ise bu durumda tırnak içerisinde yazmaya gerek yoktur.

Örnek

```
print (5)
```

Burada print( ) fonksiyonu kullanılırken parantez içerisinde kullanılan değerlere argüman denilmektedir. Python, print( ) fonksiyonu argümanını kontrol ederek, belirtilen kurallara uyup uymadığını kontrol eder. Söz diziminin doğruluğu veya değişkenin tanımlanmış olması kontrol edilir. Kod, Python'un izin verdiği tanımlamalara uyuyorsa çalıştırılarak konsol üzerinden sonuç görüntülenir. Aksi durumda hata mesajı alınır.



Yandaki resimde print fonksiyonu için gerekli olan argümanlar belirtilmiştir. Bu argümanlar, parametreler ve özellikleri üzerinde örnekler de yapılacaktır.

Örnek 2

print(“python)

```
In [22]: print("python)
  File "<ipython-input-22-92890605342d>", line 1
    print("python)
                  ^
SyntaxError: EOL while scanning string literal
```

Örnek 3

```
1.  print(a)
2.
```

```
In [17]: print(a)
Traceback (most recent call last):

  File "<ipython-input-17-c5a4f3535135>", line 1, in <module>
    print(a)

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

Yukarıdaki durumlarda eksik olan tanımlamalar neticesinde, hata mesajı ile karşılaşılmıştır. Örnek 3’te tırnak kapatılmamış, örnek 4’te ise değişken tanımlanmadığı için hata vermiştir. Doğru kullanım işe aşağıdaki gibidir:

Örnek 4

```
1.  print(“Merhaba, Python!”)
```

```
In [23]: print("Merhaba Python!")
Merhaba Python!
```

print( ) fonksiyonu kullanılırken, karakter dizilerinde çift tırnak, tek tırnak ya da üç çift tırnak kullanılabilir. Aslında bu ayrıntının Python’da çok önemli bir yeri vardır.

Örnek 5

print('Türkiye'nin en kalabalık ili İstanbul'dur')

```
In [6]: print('Türkiye'nin en kalabalık ili İstanbul'dur')
  File "C:\Users\USER\AppData\Local\Temp/
ipykernel_3456/1299646185.py", line 1
    print('Türkiye'nin en kalabalık ili İstanbul'dur')
                   ^
SyntaxError: invalid syntax


In [7]: print("Türkiye"nin en kalabalık ili İstanbul"dur")
  File "C:\Users\USER\AppData\Local\Temp/
ipykernel_3456/265345706.py", line 1
    print("Türkiye"nin en kalabalık ili İstanbul"dur")
                   ^
SyntaxError: invalid syntax
```

Örnek 6'daki kullanıma bakıldığında karakter dizisi tek tırnak işareti ile başlamış, ancak kesme işareti olarak kullanıldığı yerde sonlanmıştır 'Türkiye'nin daha sonraki ifadeler ise karakter dizisinden ziyade farklı bir argüman olarak değerlendirilmiş ancak bunlarla ilgili bir tanımlama yapılmadığı için hata mesajı vermiştir. Burada çift tırnak ve tek tırnak beraber kullanılarak sorunun çözümüne gidilebilir.

Örnek 6

print("Türkiye'nin en kalabalık ili İstanbul'dur")

```
In [25]: print("Türkiye'nin en kalabalık ili İstanbul'dur")
Türkiye'nin en kalabalık ili İstanbul'dur
```

print( ) fonksiyonu kullanılırken argüman değerleri arasında aritmetiksel işlemler yapılabilir.

Örnek 7

```
In [32]: print(3+43)
46
```

print( ) fonksiyonu, bu şekilde aritmetik işlemleri yapıp ve onları bastırmak amacıyla da kullanılabilir.

print( ) Fonksiyonu ile Kullanılabilen Karakterler ve Parametreler

print( ) fonksiyonu kullanılırken karakter dizilerinde tek tırnak ve çift tırnak işaretleri (' , ", """") kullanılır. Bu işaretler karakter dizilerinin nerede başladığını ve bittiğini ifade eder.

Ters Taksim(\ )(backslash)

Örnek 8

print("Merhaba,"Python "kullanıyorum")

```
In [27]: print("Merhaba,"Python "kullanıyorum")
  File "<ipython-input-27-682df50ce03d>", line 1
    print("Merhaba,"Python "kullanıyorum")
                          ^
SyntaxError: invalid syntax
```

Örnek 8'de çift tırnak işareti arasındaki " Merhaba, " kelimesi karakter dizisi olarak algılandı ama sonrasında Python kelimesi çift tırnak içinde ve argümanlar arasında ", " virgül olmadığı için hata verdi. Bu problemi çözmek için kaçış karakterleri (escape character) kullanılmalıdır. Aslında bu sorun, bir önceki bölümde çift tırnak ve tek tırnak'lar beraber kullanılarak çözülmüştü. Ancak burada ters taksim (\ ) işareti kullanılarak da bu sorun çözülebilir.

```
In [28]: print("Merhaba,\"Python \"kullanıyorum")
Merhaba,"Python "kullanıyorum
```

Örnek 9

print('Bursa\'nın iskenderi meşhurdur.')

```
In [29]: print('Bursa\'nın iskenderi meşhurdur.')
Bursa'nın iskenderi meşhurdur.
```

Örnek 9'da ters taksim işareti kullanılmamış olsaydı önceki bölümdeki gibi hata ile karşılaşılırdı. Burada \ karakteri kendinden sonra gelen kesme işaretinin dikkate alınmaması gerektiği anlamı vermektedir.

Alt Satır Başı (\n)

print( ) fonksiyonu kullanılırken, karakter dizilerinde bazen alt satıra inme ihtiyacı duyulabilir. Python'da en sık kullanılan kaçış parametresi \n parametresidir.

Örnek10

print('1. satır\n2. satır\n3. satır')

```
In [31]: print('1. satır\n2. satır\n3. satır')
1. satır
2. satır
3. satır
```

Örnek 10'da görüldüğü üzere **\n** parametresi kullanılarak program çıktısının alt alta yazılması sağlanmış oldu.

Sekme(\t)

Klavyeden tab tuşuna basıldığındaki gibi belirli karakter boşluk bırakılmasını sağlayan bir parametredir.

Örnek 11

print("pazartesi \tsalı \tçarşamba")

```
In [32]: print("pazartesi \tsalı \tçarşamba")
pazartesi       salı    çarşamba
```

end( ) Parametresi

Bu parametre print( ) fonksiyonu ile ekrana gönderilen değerlerin sonunda hangi işlemin yapılacağını belirtmektedir.

Örnek12

print("Merhaba!")

print("Python")

```
In [35]: print("Merhaba!")
    ...: print("Python")
Merhaba!
Python
```

Örnek 12'deki gibi bir kullanımda kodların çıktısı alt alta verilmiştir. Ancak bazı durumlarda programın çıktısı birleştirilmek istenebilir. İlerleyen bölümlerde döngü konusunda, döngü değerini her seferinde konsola yazdırılmak istendiğinde bu değerler alt alta yazılacaktır. Döngünün büyük olduğu düşünülürse program sayfalar dolusu çıktı verebilir. İşte bu gibi durumlarda end parametresi çok işe yaramaktadır.

Örnek 13

print("Merhaba!",end=" ")

print("Python")

```
In [37]: print("Merhaba!",end=" ")
    ...: print("Python")
Merhaba! Python
```

Örnek 13'te görüldüğü üzere end parametresi içinde iki tırnak arasında boşluk karakteri kullanıldığı için program çıktısını birleştirerek araya boşluk eklendi. Aynı işlem virgül kullanılarak yapılabilir:

Örnek 14

print('Merhaba!',end=", ")

print('Python')

```
In [39]: print("Merhaba!",end=", ")
    ...: print("Python")
Merhaba!, Python
```

Aslında end parametresinin içerisinde standart olarak \n parametresi vardır. Burada değer atanarak varsayılan değer değiştirilmiştir.

sep () Parametresi

Önceki örnekte, end parametresi ile değerlerin sonunda hangi işlemin yapılacağı gösterilmişti. Tek bir print( ) fonksiyonu birden fazla argüman alabilir. Her bir argümanın arasında farklı işlemler sep parametresi ile yapılabilir.

Örnek 15

print('pazartesi','salı','çarşamba','perşembe','cuma')

```
In [41]: print("pazartesi","salı","çarşamba","perşembe","cuma")
pazartesi salı çarşamba perşembe cuma
```

Örnek 15'teki gibi bir kullanımda, her bir argüman birbirinden virgül işareti ile ayrılmış ve program çıktısı olarak tüm argümanların arasına birer boşluk bırakılarak ekrana yazdırılmıştır. Burada dikkat edilmesi gereken bir başka nokta, print( ) fonksiyonu içerisine gönderilen tüm değerleri belirli kurallar çerçevesinde ekrana yazılmıştır. Her bir argüman'ın arasına sep parametresi yardımıyla birer kural belirlenmek istenirse;

Örnek 16

print('pazartesi','salı','çarşamba','perşembe','cuma',sep='-')

```
In [42]: print("pazartesi","salı","çarşamba","perşembe","cuma",sep="-")
pazartesi-salı-çarşamba-perşembe-cuma
```

format( ) Metodu ile Ekrana Bastırılacak Değerleri Biçimlendirme İşlemleri

Program yazarken bazı durumlarda bir string'in içinde daha önceden tanımlı string, float, int gibi farklı türden değerleri yerleştirmek isteyebiliriz. Böyle durumlar için Pythonda format () metodu bulunmaktadır. Örneğin, programda 3 adet tam sayı değeri, bir string ifade ile beraber ekrana yazdırılmak istenebilir. Bunun için format () metodu kullanılmalıdır.

Örnek 17

a=5

b=6

c=9

```
In [44]: a=5
    ...: b=6
    ...: c=9
    ...: print("girdiğiniz",a, b, "ve",c,"değerlerinin toplamı: ",a+b+c,"dir")
girdiğiniz 5 6 ve 9 değerlerinin toplamı:  20 dir
```

Örnek 17'deki gibi bir kullanım ve hata yapmaya müsait bir kullanımdır. Python bu gibi durumlar için print( ) fonksiyonunda format metodunun kullanımına olanak sağlar. Kullanımı aşağıdaki şekildedir:

Örnek18

print('çıktı işlemi {} {} {}'.format(1,2,3))

```
In [45]: print("çıktı işlemi {} {} {}".format(1,2,3))
çıktı işlemi 1 2 3
```

Burada print( ) fonksiyonunda kullanılan her bir {} ifadesine karşılık olarak format( ) metoduna bir adet argüman verilmelidir. Önceki örnek bu şekilde yapılmak istenirse.

Örnek20

a=5

b=6

c=9

```
In [46]: print("girdiğiniz {},{} ve {},değerlerinin toplamı,{} dir".format(a,b,c,(a
+b+c)))
girdiğiniz 5,6 ve 9,değerlerinin toplamı,20 dir
```

Süslü parantez içine sayılar girerek format metodu ile hangi sıradaki değerin geleceği belirlenebilir.

Örnek21

print('{1} {0} {2}'.format(10,'Python',20))

```
In [48]: print("{1} {0} {2}".format(10,"Python",20))
Python 10 20
```

input( ) Fonksiyonu

Python'da print( ) fonksiyonu sayesinde konsol ekranına çıktılar gönderilebilir. Önceki örneklerde bu işlem tanımlı veriler üzerinden yapılmıştır, yani kullanıcıdan bir değer almadan, program içerisinde yapılan değişken tanımlalamaları ile print( ) fonksiyonu ile değerler ekrana yazdırılmıştır. Neredeyse tüm programlama dilleri verileri okur ve işler. Kullanıcıdan girdi almayan bir program sağır bir programdır. input( ) fonksiyonu kullanıldığında genellikle kullanıcının klavyeden bir girdi yapmasını bekler.

Python'da input( ) fonksiyonu diğer programlama dillerinden daha işlevsel bir yapıya sahiptir. input( ) fonksiyonuyla, print( ) fonksiyonuna ihtiyaç duymadan kullanıcıya bilgi verilebilir. Ancak input( ) fonksiyonu kullanılırken kullanıcıdan alınan değer bir değişkene atanmalıdır.

Örnek 22

isim=input('isminizi giriniz: ')

print('merhaba! ',isim)

```
In [51]: isim=input("isminizi giriniz: ")
    ...: print("merhaba! ",isim)

isminizi giriniz: ahmet
merhaba!  ahmet
```

Örnek 22'de input( ) fonksiyonu ile kullanıcıdan bir karakter dizisi girmesi beklenmiş ve sonuç ekrana yazdırılmıştır. Peki, Python girilen değerin sayısal bir değer mi yoksa bir karakter dizisi mi olduğunu nasıl anlayacak?

Örnek23

a=input("birinci sayıyı giriniz: ")
b=input("ikinci sayıyı giriniz: ")
print("girdiğiniz sayıların toplamı: ",a+b)

```
In [52]: a=input("birinci sayıyı giriniz: ")
    ...: b=input("ikinci sayıyı giriniz: ")
    ...: print("girdiğiniz sayıların toplamı: ",a+b)

birinci sayıyı giriniz: 7

ikinci sayıyı giriniz: 8
girdiğiniz sayıların toplamı:  78

In [53]: a=input("birinci sayıyı giriniz: ")
    ...: b=input("ikinci sayıyı giriniz: ")
    ...: print("girdiğiniz sayıların toplamı: ",a+b)

birinci sayıyı giriniz: umut

ikinci sayıyı giriniz: yamak
girdiğiniz sayıların toplamı:  umutyamak
```

Örnek 23'te görüldüğü üzere uygulama bize hata vermese de aslında hatalı bir çıktı vermiştir. Bu kullanımda her iki değer bir karakter dizisi olarak algılanmış ve Python iki değer üzerinde toplama işlemi yapamadığı için yan yana yazarak birleştirmiştir. Burada yapılan işlemle gerçekte bir toplama işlemi yapılmamış iki argüman birbirleriyle birleştirilmiştir. input( ) fonksiyonu kullanılırken girdi olarak sayısal ifadeler kullanılacağı zaman bu durumun Python'a bildirilmesi gerekmektedir. Geçtiğimiz hafta değişkenler arası dönüşümleri incelemiştik. Oradaki metni sayıya çevirme hilemiz ile üstteki problemi giderrebiliriz. Bunun için Örnek 24'teki kullanım uygulanmalıdır.

Örnek 24

a=int(input("Bir sayı giriniz: "))

```
In [55]: a=int(input("Bir sayı giriniz: "))

Bir sayı giriniz: 6

In [56]: type(a)
Out[56]: int
```

Örnek 24'te görüldüğü üzere input( ) fonksiyonu, int( ) fonksiyonunun içerisine alınarak girdi sayısal ifadeye çevrilmiştir.

Örnek 25

a=int(input("birinci sayıyı giriniz: "))

b=int(input("ikinci sayıyı giriniz: "))

print("girdiğiniz sayıların toplamı:",a+b)

```
In [57]: a=int(input("birinci sayıyı giriniz: "))
    ...: b=int(input("ikinci sayıyı giriniz: "))
    ...: print("girdiğiniz sayıların toplamı:",a+b)

birinci sayıyı giriniz: 6

ikinci sayıyı giriniz: 9
girdiğiniz sayıların toplamı: 15
```

Görüldüğü üzere burada yapılan işlemle girdiler sayılara dönüştürülmüş ve toplama işlemi yapılmıştır. input( ) fonksiyonu kullanılırken sık yapılan hataların başında hatalı veri girişleri gelmektedir. int( ) fonksiyonu ile beraber integer ifade yerine string bir değer girilirse Python hata verir. Çünkü metinsel ifadeler (a, b, isim, soyisim) sayılara dönüştürülemez.

Örnek 26

a=int(input("birinci sayıyı giriniz: "))

b=int(input("ikinci sayıyı giriniz: "))

print("girdiğiniz sayıların toplamı:",a+b)

```
In [58]: a=int(input("birinci sayıyı giriniz: "))
    ...: b=int(input("ikinci sayıyı giriniz: "))
    ...: print("girdiğiniz sayıların toplamı:",a+b)
    ...:

birinci sayıyı giriniz: 6

ikinci sayıyı giriniz: umut
Traceback (most recent call last):

  File "<ipython-input-58-ee1c7cb5cba4>", line 2, in <module>
    b=int(input("ikinci sayıyı giriniz: "))

ValueError: invalid literal for int() with base 10: 'umut'
```

Yine benzer şekilde girilen ifadelerden birisi sayısal olup diğer ifade sayıya dönüştürülmezse, sayılarla metinsel ifadeler arasında aritmetiksel işlem yapılamayacağı için Python hata verecektir. Hataların ayıklanmasına yönelik çözümlere try / except kısımlarında değinilecektir.

Örnek27

```
a=int(input("birinci sayıyı giriniz: "))
b=input("ikinci sayıyı giriniz: ")
print("girdiğiniz sayıların toplamı:",a+b)
```

```
In [59]: a=int(input("birinci sayıyı giriniz: "))
    ...: b=input("ikinci sayıyı giriniz: ")
    ...: print("girdiğiniz sayıların toplamı:",a+b)

birinci sayıyı giriniz: 6

ikinci sayıyı giriniz: 8
Traceback (most recent call last):

  File "<ipython-input-59-e921a287f432>", line 3, in <module>
    print("girdiğiniz sayıların toplamı:",a+b)

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Örnek 27'de integer ve string değerler arasında işlem yapılamayacağı için Python hata vermiştir.

Çalışma Soruları

1. Ekrana "Merhaba Dünya" yazan programı yazınız.
2. Kullanıcıdan ismini alarak "merhaba Ali" şeklinde çıktı veren programı yazınız.
3. Girilen iki sayının toplamını bulan ve ekrana yazdıran programı yazınız.
4. Girilen iki sayının ortalamasını bulan ve ekrana yazdıran programı yazınız.
5. Kullanıcıdan abone numarası ve tüketim bilgisi alarak fatura tutarını hesaplayan programı yazınız.

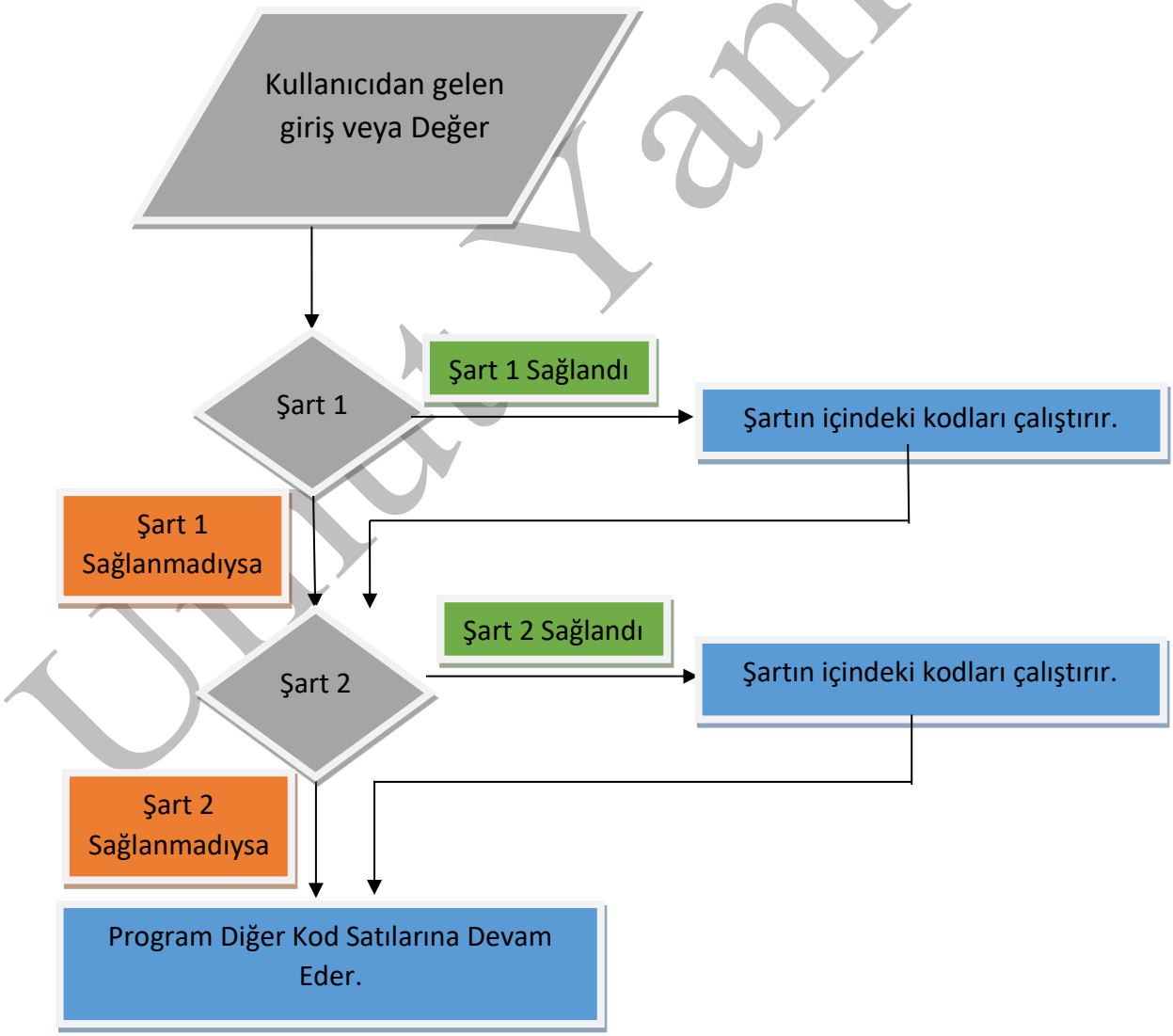
KOŞULLU VE MANTIKSAL İFADELER

Koşullu İfadeler

Program yazarken kodlar sıralı olarak alt alta satırlar şeklinde yazılır. Programın akışında bazı dallanmalar (farklı durumlar için farklı kodların çalışmasını istediğimiz durumlar) olabilir. Programlarda belirli koşul veya durumlar için çalışması istenen kodlar koşullu ifade blokları kullanılarak oluşturulur.

Örneğin, sadece üyelerin giriş yapabileceği bir program hazırlarken kullanıcı adı ve parolası doğru olan kişilerin sisteme erişimine izin veren kodlar yazılır. Bunun için programa koşul ifadeleri eklenmesi gerekir. Koşul sağlanıyorsa menülere erişim izni verecek kodlar çalışmaya başlar. Koşul sağlanmıyorsa uyarı mesajı verilerek sisteme giriş izni verilmez.

Mantıksal operatörler sonuç olarak "boolean" veri tipinde değer verir. Eğer koşul sağlanırsa "True" değeri döndürürken koşul sağlanmazsa "False" değerini döndürür. Boolean veri tipi başka bir değer alamaz. Bu durum koşullu ifadeler üretme olanağı sağlar. Koşullu ifadelerin sonucu "boolean" değer kontrol edilerek program akışı yönlendirilebilir. Koşul ifadesi ve "True-False" akışı şekilde eğer iç 2 tane koşul ifadesi varsa neler olabileceği hakkında bizlere bilgi sunar.



Şekilde görüldüğü üzere kullanıcıdan alınan veri Şart 1 yapısına geldiğinde şartı sağlıyorsa (True) bu kod girintisi (blok) içindeki komutlar çalışır. Şart 1 yapısında şart sağlanmıyorsa atlanarak sonraki kodlara geçilir. Sonrasında yeni şart yapısı (Şart 2) aynı şekilde kontrol edilir ve akış devam eder.

Mantıksal İfadeler ve Bağlaçlar Hatırlatması

Bir mantıksal ifadeyi diğer mantıksal ifadelerle bağlamanın farklı yolları vardır. Daha önce işlemiş olduğumuz ilişkisel operatörler ve mantıksal bağlaçlar kullanılarak (or, and, not vb. gibi) farklı koşul durumları oluşturulabilir.

Örneğin kullanıcı adı ve parola ile ilgili bir örnek yapalım. Kullanıcı tarafından kullanıcı adı admin şifresi ile 123456 olarak belirlenmiş olsun.

kullanıcı_adı=input('Kullanıcı Adı:')

kullanıcı_parolası=input('Parola:')

print(kullanıcı_adı=='admin' and kullanıcı_parolası=='123456')

```
In [12]: kullanıcı_adı=input('Kullanıcı Adı:')
    ...: kullanıcı_parolası=input('Parola:')
    ...: print(kullanıcı_adı=='admin' and
kullanıcı_parolası=='123456')

Kullanıcı Adı:admin

Parola:123456
True
```

Örneğin Bölümü İstatistik veya Matematik olanları seçmek için bir kod yazalım.

bölüm == "İstatistik" or bölüm == "Matematik" koşullarından biri doğruysa “True” değilse“ False” değeri döndürür.

bölüm =input('Bölümünüzü giriniz: ')

print(bölüm == "İstatistik" or bölüm == "Matematik")

```
In [15]: bölüm =input('Bölümünüzü giriniz: ')
    ...: print(bölüm == "İstatistik" or bölüm == "Matematik")

Bölümünüzü giriniz: İstatistik
True
```

Örneklerde sadece mantıksal operatörlerin sonucu “boolean ”değeri ekrana yazdırıldı. Mantıksal operatörler koşul ifadeleriyle birlikte kullanıldığında belirli şartlarda belirli kod blokları çalıştırılabilir.

Python Blok Yapısı

Python’da (başka programlama dillerinde de) kodlar belirli alt kümeler hâlinde (blok) ifade edilir. Bu yapı Python’da girinti (Indentation) ile oluşturulur. Python’da dikey olarak aynı hizadaki kodlar aynı blok yapısında yer alır. Kod bloklarının kolaylıkla ayırt edilebilmesi için bir sekme (4 karakter boşluk) kullanılması önerilmektedir. Eğer bir IDE (bütünleşik program geliştirme ortamı) kullanılıyorsa bu girintiler otomatik olarak ayarlanacaktır. Kullanılan geliştirme ortamında ayarlar bölümünde bir girintinin kaç karakter olacağının belirlendiği ayarlar bulunmaktadır. Bir karakter boşluk ile oluşturulan girinti bile yeni blok yapısını oluşturur. Bir kod bloğu kendi içinde tutarlı bir yapıdır. Döngüler, fonksiyonlar ve koşul ifadeleri kod blokları kullanılarak oluşturulur.Python’da kod yazarken girintilere

dikkat etmek gerekir. Girintiler blok yapısını belirlediği için programın yanlış çalışmasına veya çalışmamasına neden olabilir.

Örneğin, aşağıda herhangi bir karar yapısı, döngü veya fonksiyon olmadan ikinci satırdaki kodu girintili yazıldığı için hata verir.

```
print('Blok yapısı')
  print('Girinti')
```

```
In [17]: print('Blok yapısı')
    ...:     print('Girinti')
  File "<ipython-input-17-a3012ae96b64>", line 2
    print('Girinti')
    ^
IndentationError: unexpected indent
```

if yapısı

Bu yapıda, belirli komutların çalışması, bir koşula bağlıdır. Koşul sağlanmazsa herhangi bir işlem yapılmaz.

Kullanımı: Aşağıda bir "if" bloğu gösterilmektedir "if" bloğunun dikey hizasının sağında olan kod satıları koşul gerçekleştiğinde çalışır. Bu kodlar "if" bloğunda yer almaktadır. Büyük eşittir operatörü karşılaştırma sonucu "boolean" (True veya False) bir değer verir. True değer verirse "if" bloğu içinde (girinti olan) kodlar çalışır. Koşul sağlanmazsa yani "False" değeri verirse bloğun içine girilmez bloktaki kodlar atlanır.

Örnek

Kullanıcının yaş değerini alarak 18'e eşit veya büyük olması hâlinde ona mesaj veren kod:

```
yaşı = int(input('Lütfen yaşınızı giriniz: '))
if (yaşı>=18):
   print('Oy kullanabilirsiniz.')
```

```
In [24]: yaşı = int(input('Lütfen yaşınızı giriniz: '))
    ...: if (yaşı>=18):
    ...:     print('Oy kullanabilirsiniz.')

Lütfen yaşınızı giriniz: 18
Oy kullanabilirsiniz.
```

"if" bloğunun içindeki kod ancak şart sağlandığında çalışır ve blok bittikten sonra program akışı devam eder. Şart sağlanmazsa blok atlanır. Python alt satırdaki kodları yorumlar ve ona göre işlem yapar. Aşağıdaki örnek 18'den küçük bir yaş girilerek çalıştırılmıştır.

```
yaşı = int(input('Lütfen yaşınızı giriniz: '))
if (yaşı>=18):
   print('Oy kullanabilirsiniz.')
```

print ('Program bitti.')

```
In [25]: yaşı = int(input('Lütfen yaşınızı giriniz: '))
    ...: if (yaşı>=18):
    ...:     print('Oy kullanabilirsiniz.')
    ...:
    ...: print ('Program bitti.')

Lütfen yaşınızı giriniz: 17
Program bitti.
```

Örnek

yaşı = 17

adı = 'Umut'

if (yaşı >= 18):

    print('1. Şart sağlandı.')

    print ('1. if bloğunun içindesiniz.')

if (adı == 'Umut'):

    print('2. Şart sağlandı')

    print ('2. if bloğunun içindesiniz.')

print('Normal program akışında girinti kullanılmamaktadır.')

```
In [26]: yaşı = 17
    ...: adı = 'Umut'
    ...: if (yaşı >= 18):
    ...:     print('1. Şart sağlandı.')
    ...:     print ('1. if bloğunun içindesiniz.')
    ...:
    ...: if (adı == 'Umut'):
    ...:     print('2. Şart sağlandı')
    ...:     print ('2. if bloğunun içindesiniz.')
    ...:
    ...: print('Normal program akışında girinti kullanılmamaktadır.')
2. Şart sağlandı
2. if bloğunun içindesiniz.
Normal program akışında girinti kullanılmamaktadır.
```

Koşul ifadelerinde birden fazla koşul birlikte kullanılabilir. Bunun için operatörler konusunda açıklanan mantıksal operatörler kullanılır. Örnekte “and” operatörü ile iki koşulun birlikte sağlanma şartı koşulmuştur.

Örnek: Kullanıcının girdiği kullanıcı adı ve parolayı kontrol ediniz.

kullanıcı_adı=input('Kullanıcı Adı:')

kullanıcı_parolası=input('Parola:')

```
if (kullanıcı_adı == 'admin' and kullanıcı_parolası == '123456'):
    print("Giriş Başarılıdır.")
    print("Menülerinize erişebilirsiniz.")
print("blok bitti")
```

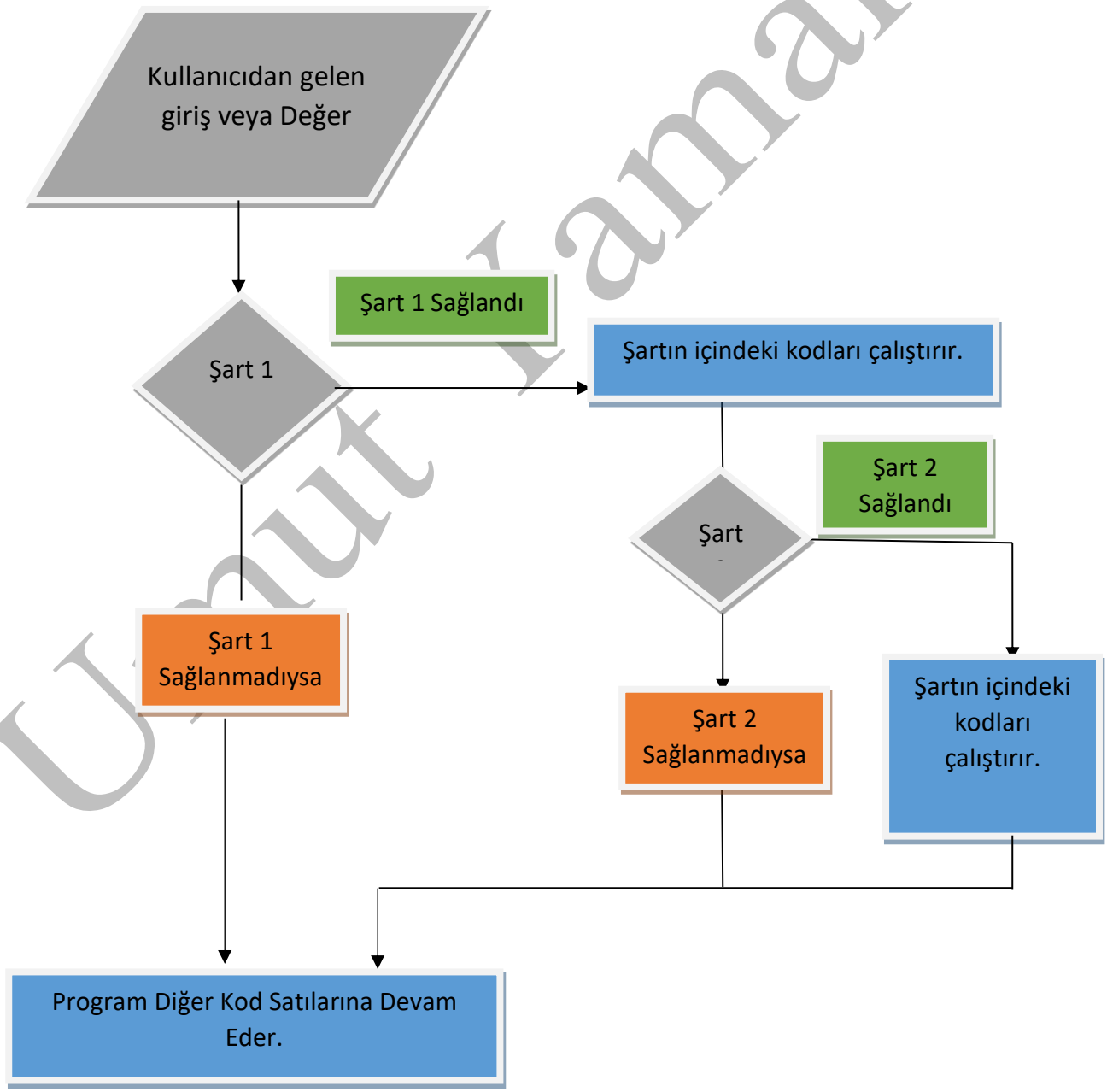
```
In [27]: kullanıcı_adı=input('Kullanıcı Adı:')
    ...: kullanıcı_parolası=input('Parola:')
    ...: if (kullanıcı_adı == 'admin' and kullanıcı_parolası ==
'123456'):
    ...:     print("Giriş Başarılıdır.")
    ...:     print("Menülerinize erişebilirsiniz.")
    ...:
    ...: print("blok bitti")

Kullanıcı Adı:admin

Parola:123456
Giriş Başarılıdır.
Menülerinize erişebilirsiniz.
blok bitti
```

Örnekte şartlardan biri bile sağlanmazsa “ if ” bloğundaki kodlar çalışmaz.

İç İçe Koşul İfadeleri

Yukarıdaki örnekte birden fazla koşulu “and ” operatörünü kullanarak kontrol edilmiştir. Aynı işlem, iç içe koşul ifadeleri kullanarak da yapılabilir.

Şekilde görülen koşullu ifadelerden birincisinin yani ilk “if” bloğundaki şart sağlanırsa (Şart 1) o bloktaki kodların çalıştığı bilinmektedir. Bu şart ifadesinin içine bir koşul ifadesi yani ikinci bir “ if ” bloğu (Şart 2) daha eklenebilir. Şart 2’deki koşul da sağlanırsa bu kez “if” bloğunun içindeki ikinci “if” bloğundaki kodlar da çalışır. İkinci koşul ifadesindeki şart sağlandığında birinci koşul ifadesindeki şart zaten sağlanmış olacağından (sağlanmasaydı ikinci “if” bloğu kodu birincinin içinde olduğu için zaten çalışmazdı) iki koşul ifadesi de (Şart 1 and Şart 2) sağlanmış olur. Koşul ifadelerinden birincisi sağlanır ikincisi sağlanamazsa ikinci “if” bloğundaki kodlar atlanır. Bu kullanım örnekteki and operatörü kullanımına benzemekle birlikte koşullardan sadece birincisinin sağlandığı durumlar için kodlar oluşturmaya olanak verir. İç içe koşul ifadelerinin sayısı (iç içe 3 koşul gibi) ihtiyaca göre artırılabilir.



Şekil : İç içe koşul ifadeleri

Örnek: Kolay anlaşılması için verilen örnekteki ikinci " if " bloğu içindeki kodların girintisine dikkat ediniz.

```
kullanıcı_adı=input('Kullanıcı Adı:')
kullanıcı_parolası=input('Parola:')
if (kullanıcı_adı == 'admin'):
    print('Kullanıcı adı doğru')
    if (kullanıcı_parolası == '123456'):
        print('Giriş başarılı.')
        print('Menülere erişebilirsiniz.')
```

```
In [30]: kullanıcı_adı=input('Kullanıcı Adı:')
    ...: kullanıcı_parolası=input('Parola:')
    ...: if (kullanıcı_adı == 'admin'):
    ...:     print('Kullanıcı adı doğru')
    ...:     if (kullanıcı_parolası == '123456'):
    ...:         print('Giriş başarılı.')
    ...:         print('Menülere erişebilirsiniz.')

Kullanıcı Adı:admin

Parola:123456
Kullanıcı adı doğru
Giriş başarılı.
Menülere erişebilirsiniz.
```

İlk şart yapısında kullanıcı adının doğru olup olmadığı kontrol edilmektedir. Eğer şart doğruysa içteki 2.şart bloğu çalışarak parola kontrolü yapacaktır. İki şart sağlanırsa ekrana tüm mesajlar yazdırılacaktır. Sadece 1. şart sağlanırsa kullanıcı adı doğru girilip parola yanlış girilirse" Kullanıcı adı doğru" mesajı ekranda görünecektir. Kullanıcı adı yanlış girilirse hiçbir mesaj görünmeyecektir.

Not: input fonksiyonu ile elde edilen değerin tipini öğrenelim. Aşağıdaki resimde de görüldüğü üzere input fonksiyonuna girişi yapılan "Kullanı Adı" değişkeni string tipindedir. Buradan anlaşıldığı üzere input fonksiyonu kullanıcıdan aldığı bilgiyi string tipinde veri olarak depolar.

```
In [31]: type("Kullanıcı Adı:")
Out[31]: str
```

if-else Yapısı

" if " yapısında şart sağlanırsa blok içindeki kodlar çalışmaktadır. Ancak şartın sağlanmadığı durumlarda herhangi bir işlem yapılmaz." else" ifadesi değilse anlamındadır. Yani şartın sağlanmadığı durumda çalışacak kodlar "else" bloğuna yazılır.

Kullanımı:

"else" bloğu da if bloğu gibi ayrı bir blok olarak yazılır. Bir " if " bloğundan sonra gelen else bloğu aynı girinti seviyesinde olmalıdır. "else" bloğu "if " bloğu ile birlikte kullanılır.

Örnek

Aşağıdaki örnekte kullanıcının girdiği sayının çift - tek olduğunu bulan bir kod yazılmıştır.

Çift bir sayı girildiğinde " if " bloğunun içindeki kodlar çalışır.

```
sayı1=int (input ('Lütfen bir sayı giriniz: ')) #input içindeki değer int olarak dönüştürüldü
if ((sayı1%2)==0):
    print('Girdiğiniz sayı çifttir.')
else :
    print('Girdiğiniz sayı tektir: ')
```

```
In [33]: sayı1=int (input ('Lütfen bir sayı giriniz: ')) #input
içindeki değer int olarak dönüştürüldü
    ...: if ((sayı1%2)==0):
    ...:     print('Girdiğiniz sayı çifttir.')
    ...:
    ...:
    ...: else :
    ...:     print('Girdiğiniz sayı tektir: ')

Lütfen bir sayı giriniz: 24
Girdiğiniz sayı çifttir.
```

Tek bir sayı girildiğinde "else" bloğunun içindeki kodlar çalışır.

```
In [32]: sayı1=int (input ('Lütfen bir sayı giriniz: ')) #input
içindeki değer int olarak dönüştürüldü
    ...: if ((sayı1%2)==0):
    ...:     print('Girdiğiniz sayı çifttir.')
    ...:
    ...:
    ...: else :
    ...:     print('Girdiğiniz sayı tektir: ')

Lütfen bir sayı giriniz: 5
Girdiğiniz sayı tektir:
```

Örnek: Koşullu ifade, operatörler ve bağlaçlarla daha etkili yapılabilir. Yukarıdaki kullanıcı adı ve parolası örneğinde kullanıcının girdiği kullanıcı adı ve parola bu örnekte "and" ile birlikte kontrol edilmiştir.

```
kullanıcı_adı=input('Kullanıcı Adı:')

kullanıcı_parolası=input('Parola:')

if (kullanıcı_adı  == 'admin' and kullanıcı_parolası  == '123456' ):

print('Giriş başarılı.')
print('Menülere erişebilirsiniz.')

else:

    print ('Yanlış kullanıcı adı veya şifre')
```

```
In [38]: kullanıcı_adı=input('Kullanıcı Adı:')
    ...: kullanıcı_parolası=input('Parola:')
    ...: if (kullanıcı_adı  == 'admin' and kullanıcı_parolası  ==
'123456' ):
    ...:     print('Giriş başarılı.')
    ...:     print('Menülere erişebilirsiniz.')
    ...: else:
    ...:     print ('Yanlış kullanıcı adı veya şifre')

Kullanıcı Adı:admin

Parola:123456
Giriş başarılı.
Menülere erişebilirsiniz.
```

Örnek: "if -else" yapısına ilişkin başka bir örnek aşağıda verilmiştir. Bu örnekte şartlar "and" bağlacıyla birleştirilmiştir.

```
yaşı = int(input('Lütfen yaşınızı giriniz: '))

bölüm = 'İstatistik'

yabancı_dil = True

#Aşağıdaki kodun çalışması için yukarıdaki 3 şartın da sağlanması gerekir.

if (yaşı >= 18 and yaşı < 35 and bölüm == 'İstatitsik' and yabancı_dil == True):

   print("Mülakata katılabilirsiniz.")

else:

   print("Şartlarınız tutmadığı için maalesef başvuru yapamazsınız.")
```

```
In [49]: yaşı = int(input('Lütfen yaşınızı giriniz: '))
    ...: bölüm = 'İstatistik'
    ...: yabancı_dil = True
    ...: #Aşağıdaki kodun çalışması için yukarıdaki  3 şartın da
sağlanması gerekir.
    ...: if (yaşı>=17 and yaşı<35 and bölüm == 'İstatistik' and
yabancı_dil == True):
    ...:     print("Mülakata katılabilirsiniz.")
    ...: else:
    ...:     print("Şartlarınız tutmadığı için maalesef başvuru
yapamazsınız.")

Lütfen yaşınızı giriniz: 20
Mülakata katılabilirsiniz.
```

```
In [50]: yaşı = int(input('Lütfen yaşınızı giriniz: '))
    ...: bölüm = 'İstatistik'
    ...: yabancı_dil = True
    ...: #Aşağıdaki kodun çalışması için yukarıdaki  3 şartın da
sağlanması gerekir.
    ...: if (yaşı>=17 and yaşı<35 and bölüm == 'İstatistik' and
yabancı_dil == True):
    ...:     print("Mülakata katılabilirsiniz.")
    ...: else:
    ...:     print("Şartlarınız tutmadığı için maalesef başvuru
yapamazsınız.")

Lütfen yaşınızı giriniz: 56
Şartlarınız tutmadığı için maalesef başvuru yapamazsınız.
```

if-elif-else Yapısı

Bu yapıda koşullar art arda verilir. if ile verilen koşulun devamında 'değilse şu ise' anlamına gelen "elif" ifadesi yer alır. Yapının en sonunda ise 'hiçbiri değilse' anlamında else ifadesi yer almaktadır. Her ifade kendi bloğundaki kodları çalıştırır. "if", "elif" ve "else" bloklarının girinti düzeyleri aynı olmak zorundadır.

Her koşul ifadesi bir "if" bloğu formatında yazılabilirdi ancak bu durumda program akışında tüm koşul ifadeleri tek tek kontrol edilirdi. "if-elif-else" yapısında ise şart sağlandığında veya else ifadesine gediğinde ilgili bloktaki kodlar çalışır ve tüm" if-elif-else" bloğundan çıkılır. Birbirleriyle bağlantısı olmayan koşullar ayrı "if" blokları şeklinde verilebilir. Ama bir değerin belirli aralıktaki şartları sağlayıp sağlamadığı kontrol edilirken "if-elif-else" yapısını kullanmak daha uygundur. Bu yapıda koşullardan biri sağlıyorsa diğer koşullar kontrol edilmez. Alınan değer "if-elif-else" yapısındaki yalnız bir koşulu sağlayabilir.

Örnek: Bir kullanıcının sınav puanını alarak durumunun değerlendirilmesi:

```
sınav_puanı = int (input("Puanınızı giriniz (0-100): "))
if sınav_puanı >= 85:
    print(5)
elif sınav_puanı >= 70:
    print(4)
elif sınav_puanı >= 55:
    print(3)
elif sınav_puanı >= 40:
    print(2)
else :
    print(1)
```

```
In [69]: sınav_puanı = int (input("Puanınızı giriniz (0-100): "))
    ...: if sınav_puanı >= 85:
    ...:     print(5)
    ...: elif sınav_puanı >= 70:
    ...:     print(4)
    ...: elif sınav_puanı >= 55:
    ...:     print(3)
    ...: elif sınav_puanı >= 40:
    ...:     print(2)
    ...: else:
    ...:     print(1)

Puanınızı giriniz (0-100): 55
3
```

Fakat girilen değer 0 ile 100 dışında bir değerse ne olacaktır?

```
In [67]: sınav_puanı = int (input("Puanınızı giriniz (0-100): "))
    ...: if sınav_puanı >= 85:
    ...:     print(5)
    ...: elif sınav_puanı >= 70:
    ...:     print(4)
    ...: elif sınav_puanı >= 55:
    ...:     print(3)
    ...: elif sınav_puanı >= 40:
    ...:     print(2)
    ...: else:
    ...:     print(1)

Puanınızı giriniz (0-100): 105
5
```

```
In [68]: sınav_puanı = int (input("Puanınızı giriniz (0-100): "))
    ...: if sınav_puanı >= 85:
    ...:     print(5)
    ...: elif sınav_puanı >= 70:
    ...:     print(4)
    ...: elif sınav_puanı >= 55:
    ...:     print(3)
    ...: elif sınav_puanı >= 40:
    ...:     print(2)
    ...: else:
    ...:     print(1)

Puanınızı giriniz (0-100): -5
1
```

Görüldüğü üzere tanım aralığı dışında değerler girildiğinde de bu program çalışmaktadır ve hatalı bir sonuç yansıtmaktadır. Bu nedenle bir ekleme yapılarak kullanıcıyı bu hatadan döndürmeliyiz. Bunu da 0 ile 100 arasında bir değer girildiğinde hatalı giriş yapıldığına dair bilgi vererek gerçekleştirebiliriz.

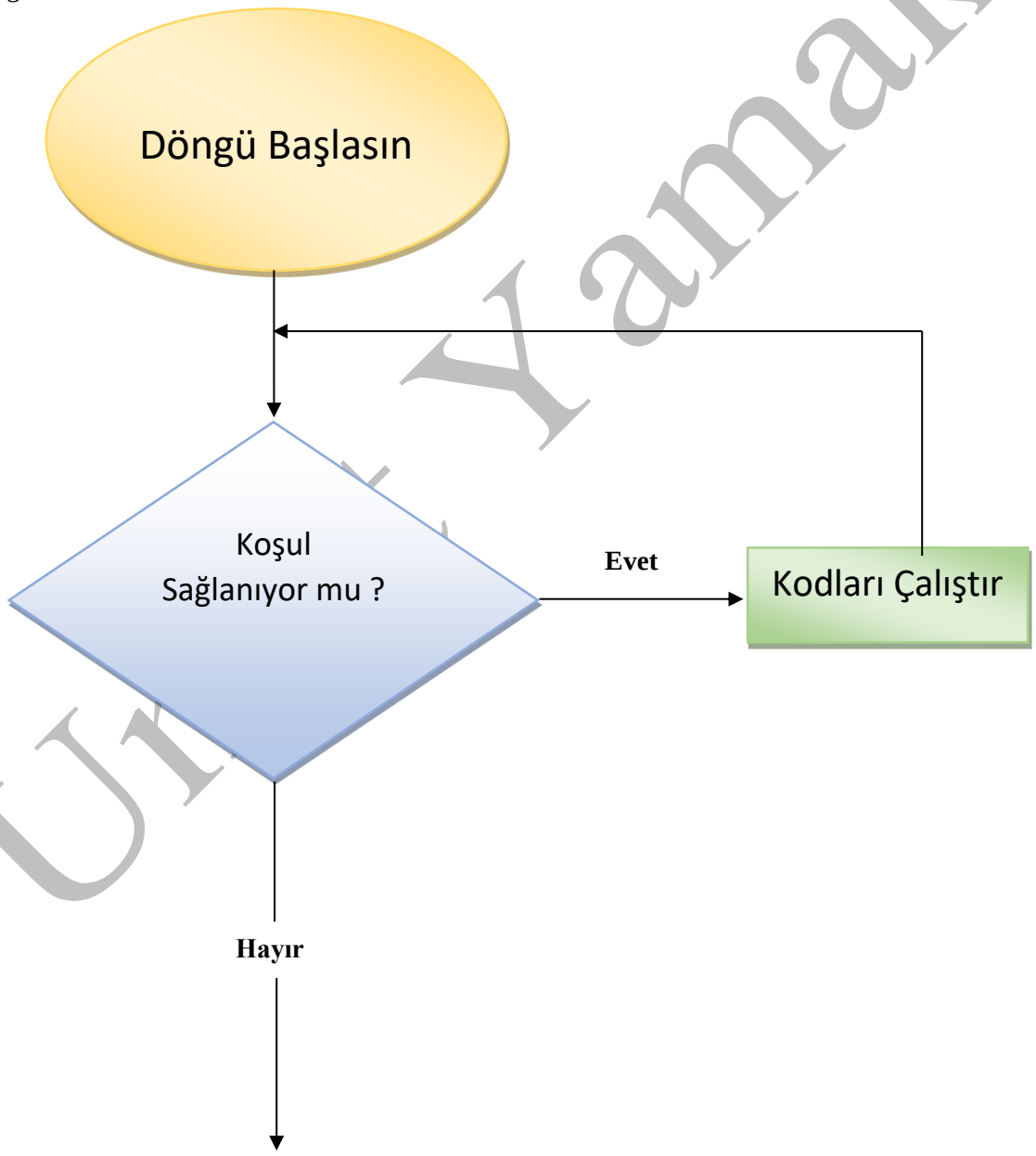
```
In [70]: sınav_puanı = int (input("Puanınız giriniz (0-100): "))
    ...: if (sınav_puanı <0 or sınav_puanı>100):
    ...:     print("Hatalı giriş lütfen 0 ile 100 arasında değer
giriniz.")
    ...: elif sınav_puanı >= 85:
    ...:     print(5)
    ...: elif sınav_puanı >= 70:
    ...:     print(4)
    ...: elif sınav_puanı >= 55:
    ...:     print(3)
    ...: elif sınav_puanı >= 40:
    ...:     print(2)
    ...: else:
    ...:     print(1)

Puanınız giriniz (0-100): -7
Hatalı giriş lütfen 0 ile 100 arasında değer giriniz.
```

DÖNGÜ YAPILARI

Bu bölüme kadar yapılan uygulamalar tek seferde çalışmaktadır. Örnek olarak 50 sayının ortalamasını almak istendiğinde, 50 sayıyı kullanıcıdan alarak her birini bir değişkene atamak, sonra da bu sayıların ortalamasının bulunması gerekiyordu. 50 sayı değil de bin adet sayının ortalamasını bulmak istersek bu iş daha da zorlaşacaktır. Döngüler, istenen kodların belirli sayı veya koşullar sağlandığı sürece tekrar tekrar çalıştırılması temeline dayanır.

Burada tekrarlama işlemi belirli sayıda olursa for döngü yapısı, belirli koşullara bağlı tekrar söz konusu ise while döngü yapısı tercih edilir. Örnek verilecek olursa her sabah güneş doğar ve her akşam güneş batar. Bu işlem süreklilik arz etmektedir.

Aşağıdaki şekilde koşul sağlandığı sürece döngü devam edecektir. Ne zaman ki koşul şartı gerçekleşmezse o durumda döngüden çıkılacaktır. Python'da while ve for döngüleri olmak üzere iki tür döngü bulunur.



1. While Döngüsü

While döngüsü, koşul gerçekleştiği sürece çalışan bir döngü çeşididir. Genellikle döngünün kaç defa çalışacağı belirli değilse while döngüsü tercih edilir. Ancak koşullar verilerek de while döngüsünün belirli sayıda çalışması sağlanabilir. Döngülerde koşullu ifadelerde olduğu gibi blok yapısı kullanılmaktadır. while ifadesinden sonra koşul durumu yazılır, ardından iki nokta işareti konularak alt satıra geçilir. Koşul durumu sağlandığı sürece çalışacak kodlar bir blok içeriden çalışır.

while (koşul durumu):

1. adım

2. adım

3. adım

.

n. adım

Bu durum bir örnekle incelenmek istenirse,

Örnek

```
1.  # şartın başlangıç değeri
2.  sayaç = 1
3.  #sayaç 6 dan küçük olduğu sürece
4.  while sayaç < 6:
5.      print("merhaba dünya")
6.      sayaç = sayaç+1
```

```
In [2]: sayaç = 1
   ...: #sayaç 6 dan küçük olduğu sürece döngü çalışacaktır.
   ...: while sayaç < 6:
   ...:     print("merhaba dünya")
   ...:     sayaç = sayaç+1
merhaba dünya
merhaba dünya
merhaba dünya
merhaba dünya
merhaba dünya
```

Örnekte sayaç isimli değişkenin değeri 1'den başlamış, yine değişken değeri 6'dan küçük olduğu sürece konsol üzerinde "merhaba dünya" yazılır. Sayacın değeri 1 arttırılarak döngünün başına döner ve sayaç değeri 6'ya eşit olana kadar bu durum devam eder.

Aynı işlemi ekrana sayaç isimli değişkenin değeri yazılarak yapmak istenirse,

Örnek:

```
1.  sayaç = 1
2.  while sayaç <6:
3.      print(sayaç)
4.      sayaç = sayaç+1
```

```
In [3]: sayaç = 1
   ...: while sayaç <6:
   ...:     print(sayaç)
   ...:     sayaç = sayaç+1
1
2
3
4
5
```

1.1 While Döngünün Kapsamı

Hatırlanacağı üzere while döngüsü açıklanırken döngü koşulunun sağlandığı sürece daha içteki bloklarda bulunan kodların çalıştırılacağında bahsedilmişti. Döngünün o andaki adımı bittiği zaman Python bir üstteki bloğa dönerek çalışmasına devam eder.

Örnek

```
# şartın başlangıç değeri
sayaç=1
#sayaç 6 dan küçük olduğu sürece
while sayaç < 6:
    print(sayaç)
    sayaç = sayaç+1
#döngü bittiği zaman
print("döngü sonlandı")
```

```
In [4]: sayaç=1
   ...: #sayaç 6 dan küçük olduğu sürece
   ...: while sayaç < 6:
   ...:     print(sayaç)
   ...:     sayaç = sayaç+1
   ...:
   ...: #döngü bittiği zaman
   ...: print("döngü sonlandı")
1
2
3
4
5
döngü sonlandı
```

while döngüsü ile çalışırken sık yapılan hatalardan birisi döngü içerisinde koşulu sağlayan değişkenin değerini arttırma işleminin unutulmasıdır. Bu durumda koşul sürekli sağlanacağı için döngü sürekli çalışır ve dışarıdan bir müdahale ile sonlandırılması gerekir.

Örnek

```
1.  sayaç=1
2.  while sayaç<6:
3.      print(sayaç)
```

Yukarıdaki örnekte bilinçli olarak sayaç isimli değişkenin değeri arttırılmamıştır. Uygulamayı çalıştırıldığında görüleceği üzere program hiç durmadan çalışacak ve ekrana sürekli 1 değeri yazacaktır. Bu işlem sırasında klavyeden "ctrl + c" tuşuna basarak Python'a kesme gönderebilir ve uygulama sonlandırıbilir.

Örnek

while döngüsü kullanarak 1-100 arasındaki (100 dâhil) çift sayıları bularak ekrana yan yana yazan programı yazalım.

```
1.  a = 1
2.  while a <= 100:
3.      if a%2 == 0:
4.          print(a,end="\t")
5.      a = a+1
```

```
In [8]: a = 1
   ...: while a <= 100:
   ...:     if a%2 == 0:
   ...:         print(a,end="\t")
   ...:     a=a+1
2       4       6       8       10      12      14      16      18
20      22      24      26      28      30      32      34      36
38      40      42      44      46      48      50      52      54
56      58      60      62      64      66      68      70      72
74      76      78      80      82      84      86      88      90
92      94      96      98      100
```

Örnek

```
1.  #while döngüsü kullanarak 1 – 100 arasındaki sayıların toplamını bulan programı yazınız.
2.  toplam = 0
3.  i = 1
4.  while i <= 100:
5.      toplam = toplam + i
6.      i = i+1
7.  print("sayıların toplamı",toplam)
```

Örnekte yapılan işlemde toplam isimli bir değişken oluşturularak başlangıç değeri 0 olarak belirlenmiştir. Çünkü toplam değerini hesaplarken toplam=toplam+i şeklinde bir işlem yapılmaktadır. Eğer toplam tanımlı olmasaydı, tanımlanmamış_değer=tanımlanmamış_değer+sayı şeklinde bir işlem yapılmaya çalışacak ve hata verecekti. Bu nedenle bir değişken ile işlem yapılmadan önce öncelikle bu değişkenin tanımlanması gerekmektedir.

```
In [10]: i = 1
    ...: while i < 100:
    ...:     topla = topla + i
    ...:     i += 1
    ...:
    ...: print("toplam",topla)
Traceback (most recent call last):

  File "<ipython-input-10-b22ebfd95394>", line 3, in <module>
    topla = topla + i

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

1.2 While True – Break İfadeleri ve Sonsuz Döngüler

Program yazarken bazen döngünün ne zaman sonlanacağı bilinmeyebilir. Örnek olarak bir markette müşterilerin alışveriş yaparak sepetlerini doldurdukları ve sepette kaç adet ürün olduğu bilinmeyebilir. O müşteriye ait tüm ürünler barkod okuyucu ile okutulmalı ve toplam tutar hesaplanmalıdır. İşte bu gibi belirsiz durumlar için while döngüsü ile beraber True ifadesi ya da benzer yapılar kullanılabilir.

Break ifadesi ise döngü sürekli çalışırken istenilen bir anda döngüden çıkmak için kullanılır.

Örnek

```
1.  i = 1
2.  while True:
3.      print(i)
4.      i += 1
5.      if i == 6:
6.          break
7.  print("döngü sonlandı")
```

```
In [11]: i = 1
    ...: while True:
    ...:     print(i)
    ...:     i += 1
    ...:     if i == 6:
    ...:         break
    ...:
    ...: print("döngü sonlandı")
1
2
3
4
5
döngü sonlandı
```

Örnek

```
1.  #Girilen sayının faktöriyelini hesaplayan programı yazalım.
2.  i=1
3.  f=int(input("faktöriyeli alınacak sayıyı giriniz: "))
4.  sonuç=1
5.  while i<=f:
6.      sonuç=sonuç*i
7.      i+=1
8.  print(sonuç)
```

```
In [1]: i=1
   ...: f=int(input("faktöriyeli alınacak sayıyı giriniz: "))
   ...: sonuç=1
   ...: while i<=f:
   ...:     sonuç=sonuç*i
   ...:     i+=1
   ...:
   ...: print(sonuç)

faktöriyeli alınacak sayıyı giriniz: 8
40320
```

Örnekte i isimli değişkeni 1 değerinden başlayarak while döngüsünü i değeri girilen sayıdan küçük olduğu sürece çalıştırılmış, daha sonra i değeri arttırarak sonuç isimli değişkenle çarpma işlemi yapılmış, i değeri f değerine eşit olduğunda döngüden çıkılarak, sonuç isimli değişkenin değeri gösterilmiştir.

Örnek

while döngüsü kullanarak sayı tahmin oyunu yapalım. Bu oyuna göre Kullanıcıdan 1-100 arası bir sayı istenmektedir. Girilen sayı, tahmin edilen sayıdan büyükse daha küçük bir sayı girmesi, büyükse daha küçük bir sayı girmesi istensin. Kullanıcı sayıyı bulana kadar bu işlem tekrar etsin. Ayrıca bir sayaç eklenerek kaç defa da tahmin ettiğini bulalım.

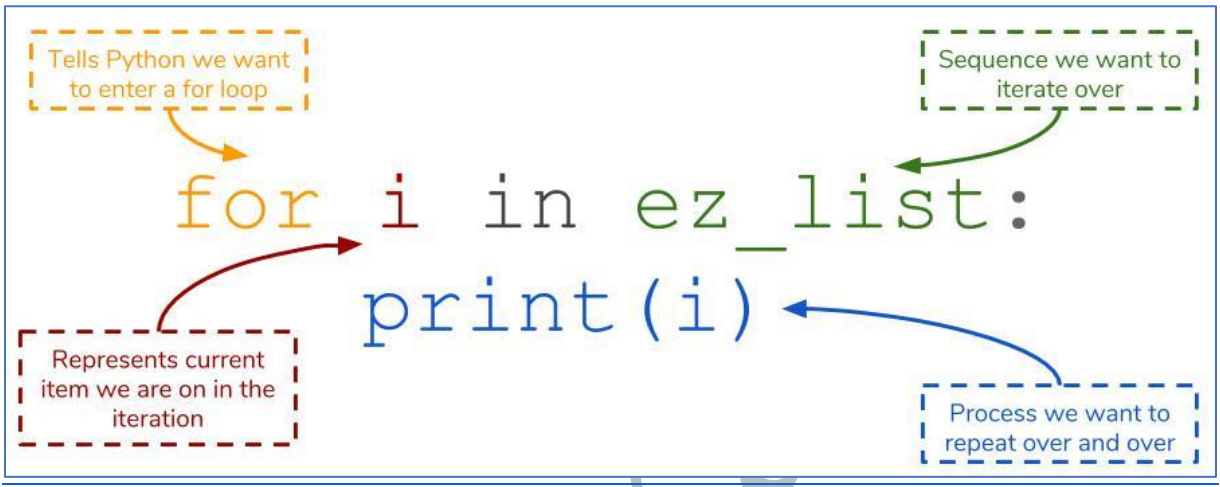
```
1.  tahmin_edilecek_sayı = 45
2.  sayaç=0
3.  print("1-100 arası bir sayı tuttum tahmin et")
4.  while 1 == 1:
5.      sayaç += 1
6.      cevap=int(input("1-100 arası bir sayı girin: "))
7.      if cevap> tahmin_edilecek_sayı:
8.          print("daha küçük bir sayı girmelisin")
9.      elif cevap< tahmin_edilecek_sayı:
10.         print(" daha büyük bir sayı girmelisin")
11.     else:
12.         print(" tebrikler tuttuğum sayıyı bildin")
13.         break
14. print("tebrikler {} seferde sayıyı bulabildin".format(sayaç))
```

```
In [13]: tahmin_edilecek_sayı = 45
    ...: sayaç=0
    ...: print("1-100 arası bir sayı tuttum tahmin et")
    ...: while 1 == 1:
    ...:     sayaç += 1
    ...:     cevap=int(input("1-100 arası bir sayı girin: "))
    ...:     if cevap>tahmin_edilecek_sayı:
    ...:         print("daha küçük bir sayı girmelisin")
    ...:     elif cevap<tahmin_edilecek_sayı:
    ...:         print("daha büyük bir sayı girmelisin")
    ...:     else:
    ...:         print("tebrikler aklımda tuttuğum sayıyı bildin")
    ...:         break
    ...:
    ...: print("tebrikler {} seferde sayıyı bulabildin".format(sayaç))
1-100 arası bir sayı tuttum tahmin et

1-100 arası bir sayı girin: 12
daha büyük bir sayı girmelisin

1-100 arası bir sayı girin: 50
daha küçük bir sayı girmelisin

1-100 arası bir sayı girin: 45
tebrikler aklımda tuttuğum sayıyı bildin
tebrikler 3 seferde sayıyı bulabildin
```

Yine burada while True yapısına benzer bir yapı kullanarak while 1==1: şeklinde bir yapı ile sonsuz döngü oluşturulmuştur.

2. For Döngüsü

For döngüsü, Python'da genellikle döngünün tekrar sayısı programcı tarafından belirlenmiş veya öngörülmüş ve belli ise kullanılır. Hatırlanacağı üzere while döngüsü ile sonsuz döngüler yapılabiliyor ve istenilen bir anda döngüden çıkılabiliyordu. For döngüsü daha çok belirli sayıdaki işlemi gerçekleştirmek için kullanılır. Bunun yanında Python'da for döngüsünün iterasyon denilen önemli bir özelliği bulunmaktadır. İterasyon işlemi sayesinde karakter dizileri ve listeler üzerinde gezinme işlemi, yani ilk elemandan son elemana kadar işlem yapabilmektedir. For döngüsü kullanmak için "in" işlecinden faydalanmak gerekmektedir. "in" işleci iterasyon işleminin yapılacağı metin/liste/sayı aralığı gibi kavramlar olabilir. Aşağıdaki resimde bir liste içerisinde döngü işlemi tanımlanmıştır.



*Bu resim https://www.dataquest.io/blog/python-generators-tutorial/ adresinden alınmıştır.

2.1. in İşleci

in işleci bir değerin, bir liste ya da karakter dizisi içerisinde olup olmadığını kontrol eder. Önce karakter dizisi içinde bir karakter olup olmadığına bakar. Eğer değer karakter dizisi içerisinde varsa True, yoksa False değeri döndürecektir.

Örneğin:

"p" in "python"

"a" in "python"

```
In [8]: "p"  in  "python"
Out[8]: True

In [9]: "a"  in "python"
Out[9]: False
```

liste=[1,2,3,4,5,6]

3 in liste

10 in liste

```
In [20]: liste=[1,2,3,4,5,6]
    ...: print(3 in liste)
    ...: print(10 in liste)
True
False
```

2.2. Karakter Dizileri Üzerinde İterasyon İşlemi

Python'da for döngüsü ile karakter dizileri üzerinde kolaylıkla iterasyon işlemi yapılabilir.

Örnek

```
1.  isim = "Mustafa"
2.  for i in isim:
3.      print(i,end=",")
4.
```

```
In [23]: """ metin üzerinde iterasyon"""
    ...: isim = "Mustafa"
    ...: for i in isim:
    ...:     print(i,end=",")
M,u,s,t,a,f,a,
```

Örnekte yapılan işlemle isim adlı değişken üzerinde ilk karakterden son karaktere kadar tüm değerleri hiçbir harf kalmayana kadar sırayla i değişkenine atayarak ekrana yazdırılmıştır.

Aynı işlemi while döngüsü ile yapılmak istenirse şu şekilde bir dizayn yapılabilir.

Örnek

```
isim="Mustafa"
i=0
while i<len(isim):
    print(isim[i],end=",")
    i=i+1
```

```
In [24]: isim="Mustafa"
    ...: i=0
    ...: while i<len(isim):
    ...:     print(isim[i],end=",")
    ...:     i=i+1
M,u,s,t,a,f,a,
```

Örnekte while döngüsü ile karakter dizileri üzerinde işlem yapılmak istendiğinde hem daha fazla kod yazılması hem de daha karışık bir yapı kullanılması gerekmektedir. Python'da genellikle listeler veya karakter dizileri üzerinde işlem yapılmak istenildiği zaman yani belirli sayıda işlem yapma ya da **iterasyon** yapılacağı zaman for döngüsü kullanılmaktadır.

Örnek: Bir cümle içerisinde geçen bir harfin kaç defa geçtiğini bulalım.

```
yazı="Python üst düzey basit sözdizimine sahip modülerliği, okunabilirliği desktekleyen, platform bağımsız nesne yönelimli yorumlanabilir bir script dilidir. "
harf="a"
sayaç=0
for i in yazı:
    if i=="a":
        sayaç=sayaç+1
print("cümle içerisinde geçen a harfi sayısı: ",sayaç)
```

```
In [28]: yazı= """Python üst düzey basit sözdizimine sahip modülerliği,
    ...: okunabilirliği desktekleyen, platform bağımsız nesne yönelimli
    ...: yorumlanabilir bir script dilidir. """
    ...: harf="a"
    ...: sayaç=0
    ...: for i in yazı:
    ...:     if i=="a":
    ...:         sayaç=sayaç+1
    ...:
    ...: print("cümle içerisinde geçen a harfi sayısı: ",sayaç)
cümle içerisinde geçen a harfi sayısı:  7
```

Örnek:Bir cümle içerisinde geçen sesli harfleri bulan programı yazınız.

yazı="Python üst düzey basit sözdizimine sahip

modülerliği, okunabilirliği desktekleyen, platform bağımsız nesne yönelimli

yorumlanabilir bir script dilidir. "

sesli="aeıioöuü"

for i in yazı:

 if i in sesli:

  print(i,end=",")

```
In [29]: yazı="""Python üst düzey basit sözdizimine sahip modülerliği,
    ...: okunabilirliği desktekleyen, platform bağımsız nesne yönelimli
    ...: yorumlanabilir bir script dilidir. """
    ...: sesli="aeıioöuü"
    ...: for i in yazı:
    ...:   if i in sesli:
    ...:     print(i,end=",")
o,ü,ü,e,a,i,ö,i,i,i,e,a,i,o,ü,e,i,i,o,u,a,i,i,i,i,e,e,e,e,a,o,a,ı,ı,e,e,ö,e,i,i,o,u,a,a,i,i,i,i,
i,i,i,
```

Örnek: İki farklı karakter dizisi belirleyerek, birinci de olup, diğerinde olmayan karakterleri bulalım.

```
1. yazı="""Python üst düzey basit sözdizimine sahip modülerliği,
2. okunabilirliği desktekleyen, platform bağımsız nesne yönelimli
3. yorumlanabilir bir script dilidir. """
4. cümle1=" Python interaktif yani etkileşimli bir programlama dilidir. "
5. for i in cümle1:
6.     if not i in yazı:
7.         print(i,end="")
```

```
In [30]: yazı="""Python üst düzey basit sözdizimine sahip modülerliği,
    ...: okunabilirliği desktekleyen, platform bağımsız nesne yönelimli
    ...: yorumlanabilir bir script dilidir. """
    ...: cümle1=" Python interaktif yani etkileşimli bir programlama dilidir. "
    ...: for i in cümle1:
    ...:     if not i in yazı:
    ...:         print(i,end="")
sg
```

2.3. range Fonksiyonu ile For Döngüsü Kullanımı

Python'da for döngüsüyle belirli değerler arasında döngü kurmak istenirse range fonksiyonu kullanılmalıdır. range fonksiyonu bir sayı dizisi oluşturur ve bu sayede oluşturulan sayı dizisi üzerinde for döngüsünün iterasyon yapması sağlanır.

Örnek

print(*range(10))

range fonksiyonu, girilen aralık arasında integer değerler oluşturur. Örnekte aralık belirtilmediği için başlangıç değeri 0 alınmıştır. Başlangıç değeri verilerek girilen değerler arasında sayı dizisi oluşturulması sağlanabilir.

Örnek

print(*range(5,20))

Ayrıca range fonksiyonuna üçüncü bir parametre verilerek atlama değeri de verilebilir.

Örnek

print(*range(1,20,3))

```
In [37]: print(*range(10))    ## durulacak nokta kendisi dahil değil
    ...: print(*range(0,10))  ## 1.parametre başlangıç 2.nokta ise durulacak nokta
    ...: print(*range(0,10,2))## 3.parametre de aralık miktarı
    ...:
    ...:
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 2 4 6 8
```

Örnek: for döngüsü ile girilen sayıya kadar olan sayıların toplamını bulalım.

```
1.  toplam=0
2.  for i in range(20):
3.      toplam=toplam+i
4.  print("girdiğiniz sayıların toplamı: ",toplam)
```

```
In [38]: toplam=0
    ...: for i in range(20):
    ...:     toplam=toplam+i
    ...:
    ...: print("girdiğiniz sayıların toplamı: ",toplam)
girdiğiniz sayıların toplamı:  190
```

Örnek: 20'den geriye doğru 0'a kadar olan sayıları ekrana yazdıralım.

```
1.  for i in range(20,0,-1):
2.      print(i,end=",")
```

```
In [39]: for i in range(20,0,-1):
    ...:     print(i,end=",")
20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,
```

Örnek: 100'e kadar 5'in katları olan sayıyı bulalım.

```
1.  for i in range(0,100,5):
2.      print(i,end=",")
```

```
In [40]: for i in range(0,100,5):
    ...:     print(i,end=",")
0,5,10,15,20,25,30,35,40,45,50,55,60,65,70,75,80,85,90,95,
```

Örnek: Girilen bir metindeki sesli harf, sayı ve özel karakterlerin sayısını bulan programı yazınız.

```
1.  sesli_harfler=["a","e","ı","i","o","ö","u","ü"]
2.  rakamlar="1234567890"
3.  özel_harf=["@","!","&","?"]
4.  özel_harf_sayısı,rakam_sayısı,sesli_sayısı=0,0,0
5.  kelime=input("lütfen incelemek için bir metin giriniz: ")
6.  for karakter  in kelime:
7.      if karakter  in sesli_harfler:
8.          sesli_sayısı+=1
9.      if karakter  in rakamlar:
10.         rakam_sayısı+=1
11.     if karakter  in özel_harf:
12.         özel_harf_sayısı+=1
13. print("girdiğiniz metinde {} adet sesli harf {} adet rakam ve {} adet özel karakter
14. bulunmaktadır. ".format(sesli_sayısı))
```

```
In [55]: sesli_harfler=["a","e","ı","i","o","ö","u","ü"]
    ...: rakamlar="1234567890"
    ...: özel_harf=["@","!","&","?"]
    ...: özel_harf_sayısı,rakam_sayısı,sesli_sayısı=0,0,0
    ...: kelime=input("lütfen incelemek için bir metin giriniz: ")
    ...: for karakter  in kelime:
    ...:     if karakter  in sesli_harfler:
    ...:         sesli_sayısı+=1
    ...:     if karakter  in rakamlar:
    ...:         rakam_sayısı+=1
    ...:     if karakter  in özel_harf:
    ...:         özel_harf_sayısı+=1
    ...:
    ...: print("""girdiğiniz metinde {} adet sesli harf {} adet rakam ve {} adet özel karakter
    ...: bulunmaktadır.""".format(sesli_sayısı,rakam_sayısı,özel_harf_sayısı))

lütfen incelemek için bir metin giriniz: girilen metindeki sesli harf sayı ve özel
    ...: karakterli bulan program123456!@&
girdiğiniz metinde 23 adet sesli harf 6 adet rakam ve 3 adet özel karakter
bulunmaktadır.
```

2.4 Continue İfadesi

Hatırlanacağı üzere break ifadesi döngünün dışına çıkılmasını sağlamaktadır. Döngülerde kullanılan continue ifadesi, döngünün baştan sona kadar çalışmasını engellemeyen ancak belirli durumlar sağlandığında o adımı atlamamızı sağlayan yapılardır. Döngü sona ermez ancak verilen koşulun sağlanması durumunda döngüyü direk başa alır.

Örnek

```
1.  for i in range(1,6):
2.      if i ==2 or i==4:
3.          continue
4.      print(i)
5.
```

```
In [59]: for i in range(1,6):
    ...:     if i ==2 or i==4:
    ...:         continue
    ...:     print(i)
1
3
5
```

Görüldüğü üzere Python, 2 veya 4 değerlerini görünce döngünün başına gitmiş ve alt satırdaki ifadeler çalıştırılmamıştır.

Örnek: Aynı işlem while döngüsü ile yapılmak istenirse,

```
1.  i=0
2.  while i < 5:
3.      i=i+1
4.      if i == 2 or i==4:
5.          continue
6.      print(i)
```

```
In [60]: i=0
    ...: while i < 5:
    ...:     i=i+1
    ...:     if i == 2 or i==4:
    ...:         continue
    ...:     print(i)
1
3
5
```

while döngüsü ile continue deyimi kullanırken döngü, sonsuz döngüye girebilir. Eğer i=i+1 ifadesi continue deyiminin altında kullanılırsa i değeri sürekli 2 olarak kalır. Değişkenin değeri artmaz ve uygulama sonsuz döngüye girer. Aşağıdaki örnekte hatalı kullanım şekli gösterilmiştir.

Örnek

```
1.  i=0
2.  while i < 5:
3.      if i == 2 or i==4:
4.          continue
5.       i=i+1
6.      print(i)
7.
```

```
In [62]: i=0
    ...: while i < 5:
    ...:     if i == 2 or i==4:
    ...:         continue
    ...:     i=i+1
    ...:     print(i)
1
2
```

2.5 İç İçe Döngüler: Döngü bloklarının içerisine kodlar eklenebileceği gibi, koşul durumları hatta başka bir döngü koymak bile mümkündür. Bir döngü yapısının içine başka bir döngü yapısının yerleştirilmesi ile elde edilen yapıya iç içe döngü (nested loop) adı verilir.

Örnek

```
1.  for i in range(4):
2.      for j in range(3):
3.          print(i,j)
```

```
In [65]: for i in range(4):
    ...:     for j in range(3):
    ...:         print(i,j)
0 0
0 1
0 2
1 0
1 1
1 2
2 0
2 1
2 2
3 0
3 1
3 2
```

Örnek: Biraz önce yaptığımız döngüyü iç içe while döngüsü kullanarak, i ve j olarak tanımlanan iki değişkenin değerlerini ekrana yazdıralım.

```
1.  i=1
2.  while i<4:
3.      j=1
4.      while j<4:
5.          print("i nin değeri: {} j nin değeri: {} ".format(i,j))
6.          j=j+1
7.      i=i+1
```

```
In [66]: i=1
    ...: while i<4:
    ...:     j=1
    ...:     while j<4:
    ...:         print("i nin değeri: {} j nin değeri: {} ".format(i,j))
    ...:         j=j+1
    ...:     i=i+1
i nin değeri: 1 j nin değeri: 1
i nin değeri: 1 j nin değeri: 2
i nin değeri: 1 j nin değeri: 3
i nin değeri: 2 j nin değeri: 1
i nin değeri: 2 j nin değeri: 2
i nin değeri: 2 j nin değeri: 3
i nin değeri: 3 j nin değeri: 1
i nin değeri: 3 j nin değeri: 2
i nin değeri: 3 j nin değeri: 3
```

Örnek: Kullanıcıdan satır ve sütun sayısı değerlerini alarak sayıları tablo şeklinde yazan programı yazalım.

```
1.  a=int(input("tablonun satır uzunluğunu giriniz: "))
2.  b=int(input("tablonun sütun uzunluğu giriniz: "))
3.  for i in range(1,a+1):
4.      for j in range(1,b+1):
5.          print(j,end=" ")
6.      print(" ")
```

```
In [52]: a=int(input("tablonun satır sayısını giriniz: "))
    ...: b=int(input("tablonun sütun sayısını giriniz: "))
    ...: for i in range(1,a+1):
    ...:     for j in range(1,b+1):
    ...:         print(j,end=" ")
    ...:     print(" " )

tablonun satır sayısını giriniz: 5

tablonun sütun sayısını giriniz: 6
1 2 3 4 5 6
1 2 3 4 5 6
1 2 3 4 5 6
1 2 3 4 5 6
1 2 3 4 5 6
```

PYTHON'DA LİSTELER

Programlamada bir değişken üzerinde sadece bir değer tutulabilmektedir. Listeler ise bir değişkenin altında birden fazla değer tutulabilmesine yarar. Listeler, içinde farklı türlerden verileri barındırabilen taşıyıcılar olarak adlandırılmaktadır. Pythonda listeler dinamik bir dizi olarak tanımlanabilir.

- Listeler Python'daki veri tiplerinden biridir.
- Listeler sıralı olarak kaydedilebilen veri yapılardır. Verilere döngü gibi yapılarla sıralı olarak erişmek istenildiğinde bize büyük avantaj sağlayıp, iki köşeli parantez arasında tanımlanırlar.
- Listeler daha sonra üzerinde duracağımız sözlüklerden farklı olarak dilimlenebilir ve elemanları sonradan değiştirilebilir.

1. Liste oluşturmak

Python'da liste oluşturmanın iki yolu vardır.

Birincisi, öğelerinizi/nesnelerinizi istediğiniz sırada köşeli parantezler [] arasına yerleştirmek ve bir değişkene atamaktır. Söz dizimini(syntaxı/sentaksı) görelim:

liste= [] #boş liste

Örnek

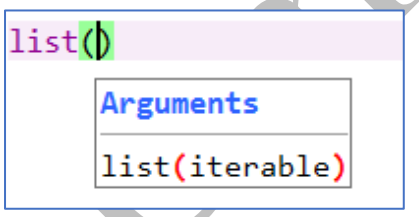
liste1=[1,2,3,4,5,6]

liste2=['a','b','c']

liste3=[0.5,1.7,67.89,3.14]

liste4=['ali','veli','yılmaz','hayri']

İkinci yöntem, list () fonksiyonu kullanmak ve öğeleri bunun içine parantezler () arasına yerleştirmektir.



Bu fonksiyon ile bir liste tanımlamak için argüman olarak iterable yani iterasyona müsait bir nesne yazılmalıdır.

Örnek

liste=list(())#listfonksiyonu

```
In [15]: liste5 = list(2,"mm",4.5)
Traceback (most recent call last):

  File "<ipython-input-15-205c01cab3f9>", line 1, in <module>
    liste5 = list(2,"mm",4.5)

TypeError: list() takes at most 1 argument (3 given)
```

Hatadan anlaşıldığı üzere sadece 1 argüman ile bu fonksiyon çalıştırılmalıdır. Ayrıca daha önce de bahsedildiği gibi İterasyona müsait bir değer girilmelidir.

Örnek:

```
In [14]: list("bilgisayar")
Out[14]: ['b', 'i', 'l', 'g', 'i', 's', 'a', 'y', 'a', 'r']
```

Eğer liste elemanları teker teker tanımlanacaksa bir parantez içi daha açılarak bu değerler girilmelidir.Bir nevi liste içerisinde liste yazılmış gibi çalışacaktır ve liste yaratılacaktır.

Örnek:

```
In [18]: liste6 = list((3,4,5,6,))
    ...: print(liste6)
[3, 4, 5, 6]
```

2. Listede Veri Tipleri

Yukarıdaki örneklerde de görülebilcceği üzere bir listede her veri tipinden eleman saklanabilir. Bu anlamda sıralı bir diziye benzemektedir.

Örnek

liste=[1,2,'ali',0.25]

print(liste)

```
In [23]: """ liste içerisindeki elemanların tipleri"""
    ...: liste=[1,2,'ali',0.25,True]
    ...: print(liste)
[1, 2, 'ali', 0.25, True]
```

Örnekte int, string ,bool ve float gibi farklı veri tiplerini içerisinde barındıran 4 elemanlı bir listedir.

3. Liste Kavramı ve İndis Değerleri

Öncelikle liste üzerinden veri okuyabilmek için hangi indis elemanının okunmak istendiği doğru bir şekilde belirtilmelidir. Örneğin Harfler isminde bir liste olduğunu düşünün. Listelerde ilk eleman her zaman 0. indistir. Listenin ilk elemanına erişmek veya yazmak istendiğinde Harfler[0] yazılması gerekmektedir. Diğer elemanları için de sırasıyla Harfler[1], Harfler[2] şeklinde yazılması gerekmektedir. Buradaki bir diğer önemli nokta da listenin ilk elemanı 0. indisten başladığı için son elemanı da liste uzunluğun bir eksiğidir.

Başlangıç			Bitiş
Liste[0]	Liste[1]	……	Liste[n-1]

Örnek

renkler = ['Red', 'Orange', 'Yellow', 'Green', 'Blue', 'Indigo', 'Violet']

print('listeyi ekrana yazdırıyoruz')

print(renkler)

Örnekte görüldüğü üzere renkler listesindeki tüm ögeleri listelenmiştir.

```
In [24]: renkler = ['Red', 'Orange', 'Yellow', 'Green', 'Blue', 'Indigo',
'Violet']
    ...: print('listeyi ekrana yazdırıyoruz')
    ...: print(renkler)
listeyi ekrana yazdırıyoruz
['Red', 'Orange', 'Yellow', 'Green', 'Blue', 'Indigo', 'Violet']
```

Python'da listeler üzerinde değişik türden veriler bir arada tutulabilir. Python'ı güçlü kılan özelliklerden biri olan listelerde, her bir eleman bir indis (indis) numarasına sahiptir. Bir listenin başlangıç indisi 0 (sıfır)'dır.

Başlangıç elemanları belli olan bir listenin tanımlanması ve yazdırılması:

Örnek

#liste1 adında bir liste tanımladık. İçine verileri girdik

#bu verileri ekrana yazdırdık

liste1 = ['a','b','c','d','e','f']

print(liste1)

Örnekte liste1 adında liste tanımlandı ve char(karakter) olarak ögeleri bulunmaktadır. İndis numarasına bakıldığı zaman; aşağıdaki şekilde ögeler vardır.

liste1[0]='a'	liste1[1]='b'	liste1[2]='c'	liste[3]='d'	liste1[4]='e'	liste1[5]='f'

4. Liste Elemanlarına Erişim

Listedeki ögelere ulaşmak için bir değer, listenin indis numarasına göre atanıp çağrılabilir. Liste oluştururken içine herhangi bir değer girilmeden de oluşturulup, sonradan değer ataması yapılabilir ya da herhangi bir indis numarasından başlanıp belirlenen indis numarasına kadar olan ögelere ulaşılır.

Aşağıdaki örneklerde görüldüğü gibi hem indis numarasından hem de başlangıcı ve bitişi belli olan indis numarası aralıklarına kadar ifadelere erişilmiştir. Listenin belirli aralıktaki öğelerini alma işlemine dilimleme denir. Liste dilimlerken adımlama da başlangıç indisi alır ama bitiş indisi almaz.

Liste dilimleme Kullanımı:

eleman = liste[indis] → herhangi bir elemanı seçme

dilim = liste[başlangıç:bitiş] → listeden bir eleman dilimi seçme

Liste, elemanlarına erişim için tamamını veya indis numarasına göre çağırmaktadır.

Örnek

liste=["birinici veri","ikinci veri","üçüncü veri","dördüncü veri","beşinci veri"]

#beş elemenlı listenin ilk verisi

print(liste[0])

#beş elemenlı listenin son verisi

print(liste[4])

Örnekte listenin indis numaraları yazılarak, başlangıç ve bitiş ögelerine ulaşılmıştır.

```
In [29]: liste=["birinici veri","ikinci veri","üçüncü veri","dördüncü
veri","beşinci veri"]
    ...: #beş elemenlı listenin ilk verisi
    ...: print(liste[0])
    ...: #beş elemenlı listenin son verisi
    ...: print(liste[4])
birinici veri
beşinci veri
```

Liste, içindeki nesneleri kontrol edilebilir.

Örnekte Eşya adında bir liste var. Liste tanımlanır, daha sonra if karar yapısı ile içindeki ögeler de“ perde” olup olmadığı kontrol edilir.

Örnek

eşya = [‘ayna’, ‘televizyon’, ‘perde’]

if(‘perde’ in eşya):

    print(‘Bu değer listede var.’)

else:

    print(‘Bu değer listede yok’)

```
In [34]: eşya = ["ayna", "televizyon", "perde"]
    ...: if("perde" in eşya):
    ...:     print("Bu değer listede var.")
    ...: else:
    ...:     print("Bu değer listede yok")
Bu değer listede var.
```

Aşağıdaki örnekte 5 elemanlı bir liste tanımlanmıştır. İndis numaralarını kullanarak, listenin 1. indisten başlayarak, 3. indise kadar öge listesine aktarıp listeleme işlemi yapılmıştır.

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

öge = listem[1:3]

print(öge)

```
In [49]: liste = [1,2,3,4,5]
    ...: öge = liste[1:3]
    ...: print(öge)
[2, 3]
```

Örnek

listem=[10,20,30,40,50]

eleman = listem[3]

print(eleman)

```
In [50]: listem=[10,20,30,40,50]
    ...: eleman = listem[3]
    ...: print(eleman)
40
```

Örnekte listem adında bir liste tanımlanarak 5 elemanlı ifade girilmiştir. Listem adlı liste için 3. indis numarasına ait ifade eleman adlı değişkene atama işlemi yapılmıştır. Eleman adlı değişken de ekrana yazdırılarak 40 sonucu elde edilmiştir.

Örnek

listem = [10,20,30,40,50]

eleman = listem[1:3]

print(eleman)

```
In [52]: listem = [10,20,30,40,50]
    ...: eleman = listem[1:3]
    ...: print(eleman)
[20, 30]
```

Örnekte listem adlı listede eleman adlı değişkene sadece 1 ve 2. indis numaralarının dâhil edildiği 3. indis numarasının dâhil edilmediği atama işlemi yapılmıştır. Eleman adlı değişken de ekrana yazdırılarak 20 ve 30 değerleri listelenmiştir.

Örnek

liste= [1,2,3,4,5,6,7,8,9,10]

print(liste)

1. eleman

print(liste[0])

6. eleman

print(liste[5])

Baştan 5. indekse kadar (dahil değil)

print(liste[:5])

1.indisten 5.indise kadar

print(liste[1:7])

#5. İndisten sonuncu elemana kadar

print(liste[5:])

tüm indisler 2 ile adımlayarak

print(liste[::2])

```
In [53]: liste = [1,2,3,4,5,6,7,8,9,10]
    ...: print(liste)
    ...: # 1. eleman
    ...: print(liste[0])
    ...: # 6. eleman
    ...: print(liste[5])
    ...: # Baştan 5. indekse kadar (dahil değil)
    ...: print(liste[:5])
    ...: # 1.indisten 5.indise kadar
    ...: print(liste[1:7])
    ...: #5. İndisten sonuncu elemana kadar
    ...: print(liste[5:])
    ...: # tüm indisler 2 ile adımlayarak
    ...: print(liste[::2])
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
1
6
[1, 2, 3, 4, 5]
[2, 3, 4, 5, 6, 7]
[6, 7, 8, 9, 10]
[1, 3, 5, 7, 9]
```

Örnekte, 10 elemanlı bir liste tanımlanmış ve indis numaraları kullanılarak ekrana listeleme işlemi yapılmıştır.

Listelerde negatif indisler de kullanılabilir. Negatif indis numarası listenin sonuncu elemanından başlayarak sayıldığında (sondan başa) sıra numarasını verir.

Örnekte listenin sonuncu elemanına liste[-1] olarak, sondan ikinciye ise [-2] olarak ulaşılır. Güle güle elemanı liste de -1.indis olarak görülmektedir. Merhaba ise geriye doğru sayıldığında -4. indis'tir.

Örnek

liste = ["merhaba", "dünya", "merhaba", "güle güle"]

print (liste [- 1]) #son ögeyi listeler

print(liste [- 3]) #sondan üçüncü ögeyi listeler

print(liste [- 4]) #sondan dördüncü ögeyi listeler

print(liste[::-1]) #sondan başa doğru listeleme yapmak için kullanılır

```
In [54]: liste = [ "merhaba", "dünya", "merhaba", "güle güle" ]
    ...: print (liste [- 1 ]) #son ögeyi listeler
    ...: print( liste [- 3 ]) #sondan üçüncü ögeyi listeler
    ...: print(liste [- 4 ]) #sondan dördüncü ögeyi listeler
    ...: print(liste[::-1]) #sondan başa doğru listeleme yapmak için
kullanılır
güle güle
dünya
merhaba
['güle güle', 'merhaba', 'dünya', 'merhaba']
```

5. Temel Liste Metotları

Metotlar listelerin işlevlerine erişilmesini sağlar. Listenin metotları için dir( ) fonksiyonunu kullanarak tüm metotlar görülebilir. Bu metotlar yardımıyla listeler ekleme, çıkarma, arama, sıralama vb. birçok işlemin kolaylıkla yapabilmesini sağlamaktadır.

Metot	Açıklama
append ()	Listeye yeni eleman ekleme işlemini yapar. Bu metot ile listeye sadece bir eleman eklenebilir ve eklenen eleman listenin sonunda yer alır.
clear ()	Listeyi değil içindeki tüm ifadeleri silmeye yarar.
copy ()	Listeden listeye kopyalama işlevine yaramaktadır.
count ()	Listenin içinde sorgulanan elemandan kaç adet olduğunu bulmamızı sağlar.
extend ()	Listeler arası genişletme işlevini görür.
index ()	Listedeki elemanları almamızı sağlar.
insert ()	Listenin istenilen indis numarasına eleman eklenebilir.
pop ()	Listedeki elemanın indisi ile silme işlem yapar. İndis belirtmediğinizde ise varsayılan olarak listenin son elemanını siler. Ayrıca bu metot silinen elemanı ekrana yazmaktadır.
remove ()	Listede istenilen elemanın değerini yazarak silme işlemi yarar.
reverse ()	Bu metot sort metodunun aksine listedeki elemanları ters alfabetik olarak sıralar.
sort ()	Listenin elemanlarını alfabetik olarak sıralar.

5.1. ‘append ’kullanımı

Örnek

takımlar=["gs","fb","bjk"]

takımlar.append("ts")

print(takımlar)

['gs', 'fb', 'bjk', 'ts']

```
In [56]: takımlar=["gs","fb","bjk"]
    ...: takımlar.append("ts")
    ...: print(takımlar)
['gs', 'fb', 'bjk', 'ts']
```

Örnekte takimlar listesine "ts" öğesi eklenmiş ve son indis numarasında yer almıştır.

5.2. 'insert' kullanımı

Örnek

sebzeler =["lahana","marul","pırasa","ıspanak","fasulye"]

sebzeler.insert(2, "patlıcan")

print(sebzeler)

```
In [57]: sebzeler =["lahana","marul","pırasa","ıspanak","fasulye"]
    ...: sebzeler.insert(2, "patlıcan")
    ...: print(sebzeler)
['lahana', 'marul', 'patlıcan', 'pırasa', 'ıspanak', 'fasulye']
```

Örnekte insert metodu kullanılarak 2. indis numarasına "patlıcan" ögesi eklenerek, listeleme işlemi yapılmıştır.

5.3. 'copy' kullanımı

Örnek

iller1 =["konya","karaman","kocaeli","kayseri","kahramanmaraş"]

iller2=[]

iller2 = iller1.copy()

print(iller2)

```
In [58]: iller1 =["konya","karaman","kocaeli","kayseri","kahramanmaraş"]
    ...: iller2=[]
    ...: iller2 = iller1.copy()
    ...: print(iller2)
['konya', 'karaman', 'kocaeli', 'kayseri', 'kahramanmaraş']
```

5.4. 'count' kullanımı

Örnek

takımlar=["gs","fb","bjk"]

print(takımlar.count('gs'))

```
In [59]: takımlar=["gs","fb","bjk"]
    ...: print(takımlar.count('gs'))
1
```

Örnek 17'de takimlar listesinde 'gs' ögesinin kaç adet olduğu count metodu bulunmuştur.

5.5. 'extend' kullanımı

Örnek

takımlar1=["gs","fb","bjk"]

takımlar2=["ts","samsunspor","göztepe"]

takımlar1.extend(takımlar2)

print(takımlar1)

```
In [71]: takımlar1=["gs","fb","bjk"]
    ...: takımlar2=["ts","samsunspor","göztepe"]
    ...: takımlar1.extend(takımlar2)
    ...: print(takımlar1)
['gs', 'fb', 'bjk', 'ts', 'samsunspor', 'göztepe']
```

Örnekte extend komutu listelerdeki ögelerin kendi elemanlarını koruyarak genişletme işlemi yapılmıştır.

5.6. 'indis' kullanımı

Örnek

sebzeler =["lahana","marul","pırasa","ıspanak","fasulye"]

print(sebzeler.index("ıspanak"))

```
In [74]: sebzeler =["lahana","marul","pırasa","ıspanak","fasulye"]
    ...: print(sebzeler.index("ıspanak"))
3
```

İndis metodu yardımıyla görüldüğü gibi verilen bir ögenin indis numarasını vermektedir.

5.7. 'clear' kullanımı

Örnek

sebzeler =["lahana","marul","pırasa","ıspanak","fasulye"]

sebzeler.clear()

print(sebzeler)

```
In [75]: sebzeler =["lahana","marul","pırasa","ıspanak","fasulye"]
    ...: sebzeler.clear()
    ...: print(sebzeler)
[]
```

clear( ) metodu kullanılarak örnekteki listenin tüm ögeleri silinmiştir.

5.8. 'pop' kullanımı

Örnek

sebzeler =["lahana","marul","pırasa","ıspanak","fasulye"]

sebzeler.pop(2)

print(sebzeler)

```
In [76]: sebzeler =["lahana","marul","pırasa","ıspanak","fasulye"]
    ...: sebzeler.pop(2)
    ...: print(sebzeler)
['lahana', 'marul', 'ıspanak', 'fasulye']
```

Örnekte pop metodu ile sebzeler listesinden 2.indis numarasına ait olan "pırasa" adlı öge silinmiştir.

5.9. 'remove 'kullanımı

Örnek

sebzeler =["lahana","marul","pırasa","ıspanak","fasulye"]

sebzeler.remove("marul")

print(sebzeler)

```
In [79]: sebzeler =["lahana","marul","pırasa","ıspanak","fasulye"]
    ...: sebzeler.remove("marul")
    ...: print(sebzeler)
['lahana', 'pırasa', 'ıspanak', 'fasulye']
```

Örnekte öge adına göre silme işlemi yapılmıştır ve "marul" adlı öge sebzeler listesinden silinmiştir.

5.10. 'reverse' kullanımı

Örnek

sayilar=[10,20,30,40,50,60,70]

sayilar.reverse()

print(sayilar)

```
In [80]: sayilar=[10,20,30,40,50,60,70]
    ...: sayilar.reverse()
    ...: print(sayilar)
[70, 60, 50, 40, 30, 20, 10]
```

Örnekte reverse metodu ile liste öge elemanları tersten sıralanmıştır.

5.11. 'sort' kullanımı

Örnek

sebzeler =["lahana","marul","pırasa","ıspanak","fasulye"]

sebzeler.sort()

print(sebzeler)

```
In [81]: sebzeler =["lahana","marul","pırasa","ıspanak","fasulye"]
    ...: sebzeler.sort()
    ...: print(sebzeler)
['fasulye', 'lahana', 'marul', 'pırasa', 'ıspanak']
```

Sort metodu ile sebzeler listesi ögeleri alfabetik olarak sıralanmıştır.

Yukarıdaki örneklerde temel liste metotlarının her birine yönelik örnek ve çıktıları verilmiştir. Ayrıca __xxx__ şeklinde özel metotlar da bulunmaktadır. Bu metotlarda dir(list) şeklinde komut satırına yazıldığında aşağıdaki şekilde çıktı alınır.

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

6. Len() Fonksiyonu ile Uzunluk Bilgisi

len( ) fonksiyonu, İngilizce length'in (uzunluk) kısaltılmış hâlidir. String ifadesinin uzunluğunu yani karakter sayısını verir. Değişkene değer atadığımızda değişkendeki karakterlerin sayısını vermektedir.

Örnek

metin="Python"

print(len(metin))

```
In [87]: metin="Python"
    ...: print(len(metin))
6
```

Örnekte "len") kullanımına bakıldığında, takimlar adında bir liste tanımlanmış. Bu listeye veri girişi yapılmıştır. len( ) komutu ile listenin adı yazılarak, kaç elemanlı olduğu ekrana yazdırılmıştır. Örnekte len( ) fonksiyonu ile sebzeler listesinin eleman listesi 4 olarak verilmiştir.

Örnek

sebzeler =["lahana","marul","pırasa","ıspanak","fasulye"]

print(len(sebzeler))

```
In [88]: sebzeler =["lahana","marul","pırasa","ıspanak","fasulye"]
    ...: print( len(sebzeler))
5
```

Örnek

liste1, liste2 = ['abc',56,74 ,'python'], [12, 'opencv','a']

print ("İlk liste uzunlugu : ", len(liste1))

print("İkinci listenin uzunluğu : ", len(liste2))

```
In [91]: liste1, liste2 = ['abc',56,74 ,'python'], [12, 'opencv','a']
    ...: print ("İlk liste uzunlugu       : ", len(liste1))
    ...: print( "İkinci listenin uzunluğu : ", len(liste2))
İlk liste uzunlugu       :  4
İkinci listenin uzunluğu :  3
```

Örnekte liste1 ve liste2 adında 2 adet liste tanımlanmıştır. Bu listelere elemanlar girilerek, len()komutu ile kaç elemanlı olduğu bulunmuştur.

6. İç İçe Listeler

Bir liste herhangi bir sıralama nesnesi içerebilir, hatta başka bir liste (alt liste) içerebilir, alt listeler de alt listeler içerebilir ve bu şekilde devam eder. Bu yuvalanmış liste olarak bilinir. Hiyerarşik yapılara veri düzenlemek için bunlar kullanılabilir.

Örnek

liste1 = [1,2,3]

liste2 = [4,5,6]

liste3 = [7,8,9]

yeniliste = [liste1,liste2,liste3]

print(yeniliste)

```
In [92]: liste1 = [1,2,3]
    ...: liste2 = [4,5,6]
    ...: liste3 = [7,8,9]
    ...: yeniliste = [liste1,liste2,liste3]
    ...: print(yeniliste)
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Örnekte 3 adet liste oluşturuldu. Bu listelerin her biri ayrı öge olacak şekilde birleştirilerek iç içe liste oluşturuldu ve ekrana yazdırıldı.

Farklı bir şekilde aynı işlevi yerine getirelim.

sebzeler=[]

sebzeler.append(['yeşil','ıspanak'])

sebzeler.append(['beyaz','lahana'])

sebzeler.append(['turuncu','havuç'])

sebzeler.append(['siyah','turp'])

sebzeler.append(['kırmızı','domates'])

append( ) metodunu kullanarak sebzeler adlı listeye 5 adet eleman girildi. Girilen verilerin listelenmesi işe şu şekilde gerçekleştirilir:

print(sebzeler)

```
In [97]: sebzeler=[]
    ...: sebzeler.append(['yeşil','ıspanak'])
    ...: sebzeler.append(['beyaz','lahana'])
    ...: sebzeler.append(['turuncu','havuç'])
    ...: sebzeler.append(['siyah','turp'])
    ...: sebzeler.append(['kırmızı','domates'])
    ...: """append( ) metodunu kullanarak sebzeler adlı listeye 5 adet
eleman girildi.
    ...: Girilen verilerin listelenmesi işe şu şekilde gerçekleştirilir:"""
    ...: print(sebzeler)
[['yeşil', 'ıspanak'], ['beyaz', 'lahana'], ['turuncu', 'havuç'], ['siyah',
'turp'], ['kırmızı', 'domates']]
```

Sebzeler adında listede 4 adet eleman bulunmaktadır. Bunları teker teker sıralanırsa:

print(sebzeler[0])

```
In [98]: print(sebzeler[0])
['yeşil', 'ıspanak']
```

Liste adı	sebzeler				
Eleman değeri	‘yeşil’, ‘ıspanak’	‘beyaz’, ‘lahana’	‘turuncu’, ‘havuç’	‘siyah’, ‘turp’	‘kırmızı’, ‘domates’
İndis numarası	0	1	2	3	4

Örnek

sebzeler=[['yeşil','ıspanak'],['beyaz','lahana'],['turuncu','havuç']]

sebze=sebzeler[1]

print(sebze)

Örneğe bakıldığı zaman, sebze değişkenine sadece sebzeler [1] listesindeki değer atanmıştır.

Örnek

Tablodaki sebzeler listesinde sadece havuç değeri listelenmek isteniyorsa indis numarasından faydalanılır.

sebzeler=[['yeşil','ıspanak'],['beyaz','lahana'],['turuncu','havuç']]

print(sebzeler[2][1])

```
In [102]: sebzeler=[['yeşil','ıspanak'],['beyaz','lahana'],
['turuncu','havuç']]
     ...: print(sebzeler[2][1])
havuç
```

Örnek

3 Adet liste oluşturalım.

birinci_liste = [1,2,3]

ikinci_liste = ['a','b','c']

üçüncü_liste = [40,50,60]

son_liste= [birinci_liste,ikinci_liste, üçüncü_liste]

print(son_liste)

```
In [100]: birinci_liste = [1,2,3]
     ...: ikinci_liste = ['a','b','c']
     ...: üçüncü_liste = [40,50,60]
     ...: son_liste= [birinci_liste,ikinci_liste, üçüncü_liste]
     ...: print(son_liste)
[[1, 2, 3], ['a', 'b', 'c'], [40, 50, 60]]
```

Üstteki örnekte 4 adet liste oluşturulmuştur. İlk üç listenin tüm ögeleri son_liste adında listeye aktarılarak, ekrana yazdırılmıştır.

Örnekte 3 adet liste oluşturulmuştur. Bunlara birinci_liste, ikinci_liste, üçüncü_liste isimleri verilmiştir. son_liste adında liste oluşturularak diğer üç listenin elemanları bu listeye kaydedilmiştir.Ekrana sadece a ve 50 değerlerini yan yana listelemek için aşağıdaki kodlar kullanılır.

Örnek

```
1. # 3 Adet liste oluşturalım.
2. birinci_liste = [1,2,3]
3. ikinci_liste = ['a','b','c']
4. üçüncü_liste = [40,50,60]
5. son_liste= [birinci_liste,ikinci_liste, üçüncü_liste]
6. print(son_liste[1][0],son_liste[2][1])7.
```

```
In [101]: print(son_liste[1][0],son_liste[2][1])
a 50
```

7.Listeler Üzerinde İterasyon İşlemi

Python'da karakter dizilerinde yapılan işlem gibi listeler üzerinde de iterasyon işlemi yapılabilir. Örnek olarak elimizde bir liste olduğunu ve içerisinde sayısal ifadeler (integer tanımlanmış) olduğunu düşünelim. Bu değeri toplayabilir, ortalamasını bulabilir ya da farklı bir listeye atayabilirsiniz. Kullanım şekli itibariyle karakter dizilerinde yapılan işlemlerin aynısı yapılabilir.

Örnek: İçerisinde sayısal değerler olan bir listedeki değerlerin karesini alarak başka bir listeye atayınız.

```
1. sayılar = [1,2,3,4,5]
2. kareler = []
3. for i in sayılar:
4.     kareler.append(i*i)
5. print(kareler)
```

```
In [113]: sayılar = [1,2,3,4,5]
     ...: kareler = []
     ...: for i in sayılar:
     ...:     kareler.append(i*i)
     ...:
     ...: print(kareler)
[1, 4, 9, 16, 25]
```

Örnek: Bir liste içerisinde bulunan değerlerin ortalamasını bulun.

```
sayılar = [1,2,3,4,5,6,7,8,9]
toplam=0
for i in sayılar:
    toplam=toplam+i
print("sayıların ortalaması: ",toplam/len(sayılar))
```

```
In [114]: sayılar = [1,2,3,4,5,6,7,8,9]
     ...: toplam=0
     ...: for i in sayılar:
     ...:     toplam=toplam+i
     ...:
     ...: print("sayıların ortalaması: ",toplam/len(sayılar))
sayıların ortalaması:  5.0
```

Örnek: Listedeki değerlerden 3'ün katları olan sayıları ekrana yazdırın.

```
sayılar = [5,8,12,17,25,36,41,49,60,72]

for i in sayılar:

    if i%3==0:

        print(i,end=",")
```

```
In [115]: sayılar = [5,8,12,17,25,36,41,49,60,72]
     ...: for i in sayılar:
     ...:     if i%3==0:
     ...:         print(i,end=",")
12,36,60,72,
```

Örnek: İç içe listeler üzerinde gezinme işlemi yapılabilir. [[3,4],[7,8],[10,11],[14,15]] şeklinde bir liste olduğunu varsayalım. Liste elemanları üzerinde klasik bir şekilde gezinme işlemi yapılmak istenirse:

```
liste = [[3,4],[7,8],[10,11],[14,15]]

for i in liste:

    print(i,end=",")
```

```
In [116]: liste = [[3,4],[7,8],[10,11],[14,15]]
     ...: for i in liste:
     ...:     print(i,end=",")
[3, 4],[7, 8],[10, 11],[14, 15],
```

Listeye erişildi ancak alt listelere erişilebilmesi için aşağıdaki gibi bir yapının kullanılması gerekmektedir.

Örnek:

```
liste = [[3,4],[7,8],[10,11],[14,15]]

for i,j in liste:

    print(i,end=",")

    print(j,end=",")
```

```
In [117]: liste = [[3,4],[7,8],[10,11],[14,15]]
     ...: for i,j in liste:
     ...:     print(i,end=",")
     ...:     print(j,end=",")
3,4,7,8,10,11,14,15,
```

Örnek: Ya da iç içe for döngüsü kullanarak aynı işlem gerçekleştirilebilir.

liste = [[3,4],[7,8],[10,11],[14,15]]

for i in liste:

    for j in i:

        print(j,end=",")

```
In [118]: liste = [[3,4],[7,8],[10,11],[14,15]]
     ...: for i in liste:
     ...:     for j in i:
     ...:         print(j,end=",")
3,4,7,8,10,11,14,15,
```

8.Genel Örnekler

Listelerde veri tipi dönüşümleri için, elemanlara yeni değer ataması yapıldığında string, int, float vb. veri tipleri arasında değer alabilir.

Örnek

liste=[1,2,3,4,5,'ankara']

print(liste)

```
In [104]: liste=[1,2,3,4,5,'ankara']
     ...: print(liste)
[1, 2, 3, 4, 5, 'ankara']
```

liste[0]=str("kocaeli")

liste[2]=float(1.5)

liste[5]=int(20)

print(liste)

```
In [105]: liste[0]=str("kocaeli")
     ...: liste[2]=float(1.5)
     ...: liste[5]=int(20)
     ...: print(liste)
['kocaeli', 2, 1.5, 4, 5, 20]
```

Örnekte liste adında bir liste örneği tanımlanmıştır. İçindeki elemanları 1,2,3,4,5,'ankara' 'dır. İndis numarasına göre liste [0] değeri önce 1 iken veri tipi dönüşümünden dolayı 'kocaeli' olmuştur. Liste [2]'in değeri ise ilk başta 3'tür. Daha sonra float veri tipinde 1.5 değerini almıştır. Son olarak liste[5]'indeğeri 'ankara'dır. Liste[5]'e int veri tipinde bir değer kaydedilerek 20 olmuştur.

Herhangi bir string türünde veriyi parçalayarak da liste oluşturulabilir. Örnekte string veri türünde adı meyve olan bir değişken olarak tanımlanmış ve veri olarak elmayı atanmıştır. Meyve adlı değişkenliste yardımıyla parçalanmıştır.

Örnek

```
meyve="elma"

liste=list(meyve)

print (liste)
```

```
In [106]: meyve="elma"
     ...: liste=list(meyve)
     ...: print (liste)
['e', 'l', 'm', 'a']
```

Aşağıdaki örnekte 1 ile 15 arasındaki sayılardan oluşan liste oluşturulmuştur. Listenin ögeleri ekrana listelenmiştir. Ekrana listeleme işleminde sort( ) metodu ile ögeler küçükten büyüğe, reverse ile büyükten küçüğe doğru sıralanmıştır.

Örnek

```
liste=list(range(1,15,2))

print(liste)
```

```
In [107]: liste=list(range(1,15,2))
     ...: print(liste)
[1, 3, 5, 7, 9, 11, 13]
```

```
liste.sort()

print(liste)
```

```
In [108]: liste.sort()
     ...: print(liste)
[1, 3, 5, 7, 9, 11, 13]
```

```
liste.reverse()

print(liste)
```

```
In [109]: liste.reverse()
     ...: print(liste)
[13, 11, 9, 7, 5, 3, 1]
```

split( ) metodu, listeyi belirtilen ayıracı kullanarak yeniden döndürmeye yarar. Yani split( ) karakter dizilerini istenen şekilde böler. -ayırıcı diye tanımladığımız ilk parametre, karakter dizisinin nereden bölüneceğini seçer. Eğer ayırıcı tanımlanmazsa karakter dizisi her boşluk gördüğünde ayırır. Örnekte bilgiler girildikçe listeye kaydedecektir. Listeye 4 adet öge girilmiş ve listeleme işlemi yapılmıştır.

Örnek

```
1. bilgi=input("bilgilerinizi araya virgül koyarak yazınız: ")
2. liste=bilgi.split(",")
3. print(liste)
```

```
In [110]: bilgi=input("bilgilerinizi araya virgül koyarak yazınız: ")
     ...: liste=bilgi.split(",")
     ...: print(liste)

bilgilerinizi araya virgül koyarak yazınız: umut,yamak,omü,istatistik
['umut', 'yamak', 'omü', 'istatistik']
```

Aşağıdaki örnekte ise cümle değişkenindeki kelimeler split metodu ile kelimeler listesine aktarılmıştır. len( ) metodu ile de kaç adet öge olduğu ekrana listelenmiştir.

Örnek

```
1. cümle ="23 nisan herkese mutlu olsun"
2. kelimeler=cümle.split(" ")
3. print("cümlenizde ",len(kelimeler), "adet kelime vardır")4.
```

```
In [112]: cümle="23 nisan herkese mutlu olsun"
     ...: kelimeler=cümle.split(" ")
     ...: print("cümlenizde ",len(kelimeler), "adet kelime vardır")
cümlenizde  5 adet kelime vardır
```

PYTHON'DA DEMETLER(TUPLES)

Demetler, tek bir değişkende birden çok öğeyi depolamak için kullanılır. Demetler Python'daki veri koleksiyonlarını depolamak için kullanılan 4 yerleşik veri türünden biridir, diğer 3'ü Listeler, Kümeler ve Sözlüklerdir, hepsi farklı niteliklere ve kullanıma sahiptir. Tuple, sıralanmış ve değiştirilemeyen verilerden oluşmuş bir veri koleksiyonudur.

Demetler, özellikle görünüş olarak listelere çok benzeyen bir veri tipidir. Bu veri tipi de, tıpkı listeler gibi, farklı veri tiplerini içinde barındıran kapsayıcı bir veri tipidir.

Fakat, demetler veri ekleme ve çıkarmanın yapılamadığı veri yapılarıdır. Demetler ekleme çıkarma gibi işlemlerle uğraşmadığı için daha hızlı çalışır.Eklenen verilerin program boyunca değiştirilmesinin istenmediği durumlarda kullanılmaktadır.

1. Demet Tanımlamak

Demet tanımlamanın birkaç farklı yolu vardır. Nasıl karakter dizilerinin ayırt edici özelliği tırnak işaretleri, listelerin ayırt edici özelliği ise köşeli parantez işaretleri, sözlüklerin küme parantezi ise, demetlerin ayırt edici özelliği de normal parantez işaretleridir. Dolayısıyla bir demet tanımlamak için normal parantez işaretlerinden yararlanacağız:

```
In [4]: demet = () # boş tuple/ demet yaratalım
   ...: type(demet)
Out[4]: tuple
```

```
In [6]: demet = ("ahmet", "mehmet", 2,3)
   ...: print(type(demet))
<class 'tuple'>
```

Görüldüğü gibi, karakter dizilerinin type() sorgusuna str, listelerin ise list cevabı vermesi gibi, demetler de type() sorgusuna tuple cevabı veriyor.

Python'da demet tanımlamanın birden fazla yolu vardır. Mesela yukarıdaki demeti şöyle de tanımlayabiliriz:

```
In [9]: demet = "ahmet", "mehmet", 23, 45
   ...: print(demet)
   ...: print(type(demet))
('ahmet', 'mehmet', 23, 45)
<class 'tuple'>
```

Görüldüğü gibi parantez işaretlerini kullanmadan, öğeleri yalnızca virgül işareti ile ayırdığımızda da elde ettiğimiz şey bir demet oluyor.

Demet oluşturmak için tuple() adlı bir fonksiyondan da yararlanabilirsiniz. Bu fonksiyon, liste oluşturan list() fonksiyonuna çok benzer:

```
tuple()
    Arguments
    tuple(iterable)
```

List fonksiyonunda da olduğu gibi tuple fonksiyonunun argümanının da iteratsyona müsait bir nesne olması gerekmektedir.

```
In [10]: demet = tuple("istatistik")
    ...: print(demet)
('i', 's', 't', 'a', 't', 'i', 's', 't', 'i', 'k')
```

Bu fonksiyonu kullanarak başka veri tiplerini demete dönüştürebilirsiniz:

```
In [11]: demet = tuple([1,2,3,4])
    ...: print(demet)
(1, 2, 3, 4)
```

Burada, listeyi tuple() fonksiyonu yardımıyla demete dönüştürdük.

2. Tek Öğeli bir Demet Tanımlamak

Önceki veri tiplerindeki gibi tek elemanlı demet tanımlamak bizleri yanlışa götürecektir:

```
In [19]: demet=(2)
    ...: print(demet)
    ...: type(demet)
2
Out[19]: int
```

```
In [20]: demet = ("python")
    ...: print(demet)
    ...: type(demet) ### bir
python
Out[20]: str
```

Görüldüğü gibi karakter dizisi ve tam sayı şeklinde veriler üretilmiş oldu. Bunu aşmak adına parantezlerin içine yazılan değişkenlerin yanına bir virgül ilave edilebilir.

```
In [22]: demet=(2,)
    ...: print(demet)
    ...: type(demet) ### bir tuple tanımlanmış oldu
(2,)
Out[22]: tuple
```

```
In [23]: demet = ("python",)
    ...: print(demet)
    ...: type(demet) ### bir tuple tanımlanmış oldu
('python',)
Out[23]: tuple
```

3. Demetlerin Öğelerine Erişmek

Eğer bir demet içinde yer alan herhangi bir öğeye erişmek isterseniz, karakter dizileri ve listelerden hatırladığınız yöntemi kullanabilirsiniz:

```
In [29]: demet = ('elma', 'armut', 'kiraz')
    ...: print("demetin ilk elemanını yazdırır:",demet[0])
    ...: print("demetin son elemanını yazdırır:",demet[-1])
    ...: print("demeti baştan 1. ve 2.elemanını yazdırır:",demet[:2])
    ...: print("demeti baştan sona kadar yazdırır:",demet[:])
demetin ilk elemanını yazdırır: elma
demetin son elemanını yazdırır: kiraz
demeti baştan 1. ve 2.elemanını yazdırır: ('elma', 'armut')
demeti baştan sona kadar yazdırır: ('elma', 'armut', 'kiraz')
```

Daha önce öğrendiğimiz indeksleme ve dilimleme kuralları aynen demetler için de geçerli olduğunu görmüş oluyoruz.

4. Demetlerle Listelerin Birbirinden Farkı

En başta da söylendiği gibi, demetlerle listeler birbirine çok benzer. Ama demetlerle listelerin birbirinden çok önemli bazı farkları da vardır. Bu iki veri tipi arasındaki en önemli fark şudur; listeler değiştirilebilir (*mutable*) bir veri tipi iken, demetler değiştirilemez (*immutable*) bir veri

tipidir. Yani tıpkı karakter dizileri gibi, demetler de bir kez tanımlandıktan sonra bunların üzerinde değişiklik yapmak mümkün değildir.

```
In [30]: demet = ('elma', 'armut', 'kiraz')
    ...: demet[0] = "muz"
Traceback (most recent call last):

  File "<ipython-input-30-fc94f92b45a6>", line 2, in <module>
    demet[0] = "muz"

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

Demetin herhangi bir öğesini değiştirmeye çalıştığımızda Python bize bir hata mesajı gösteriyor.Bu mesaja göre tuple/demet tipi nesnenin öğe atamasını desteklemediğini öğrenmiş oluyoruz.

Bu bakımdan, eğer programın akışı esnasında üzerinde değişiklik yapmayacağınız veya değişiklik yapılmasını istemediğiniz birtakım veriler varsa ve eğer siz bu verileri liste benzeri bir taşıyıcı içine yerleştirmek istiyorsanız, listeler yerine demetleri kullanabilirsiniz. Ayrıca demetler üzerinde işlem yapmak listelere kıyasla daha hızlıdır. Dolayısıyla, performans avantajı nedeniyle de listeler yerine demetleri kullanmak isteyebilirsiniz.

Demetlerin üzerinde direkt olarak bir değişimin yapılamaycağını gördük. Fakat önceden tanımlanmış bir demetin üzerinde değişiklik yapabilmek için, örneğin bir demetle başka bir demeti birleştirerek de demeti yeniden tanımlamak da mümkündür:

```
In [34]: demet = ('elma', 'armut', 'kiraz')
    ...: demet = demet +  ("muz",)
    ...: print(demet)
('elma', 'armut', 'kiraz', 'muz')
```

5. Demetlerin Kullanım Alanı

Demetleri ilk öğrendiğinizde bu veri tipi size son derece gereksizmiş gibi gelebilir. Ama aslında oldukça yaygın kullanılan bir veri tipidir. Özellikle programların ayar (*conf*) dosyalarında bu veri tipi sıklıkla kullanılır. Örneğin Python tabanlı bir web çatısı (*framework*) olan Django'nun settings.py adlı ayar dosyasında pek çok değer bir demet olarak saklanır. Mesela bir Django projesinde web sayfalarının şablonlarını (*template*) hangi dizin altında saklayacağınızı belirlediğiniz ayar şöyle görünür:

TEMPLATE_DIRS = ('/home/projects/djprojects/blog/templates',)

Burada, şablon dosyalarının hangi dizinde yer alacağını bir demet içinde gösteriyoruz. Bu demet içine birden fazla dizin adı yazabilirdik. Ama biz bütün şablon dosyalarını tek bir dizin altında tutmayı tercih ettiğimiz için tek öğeli bir demet tanımlamışız. Bu arada, daha önce de söylediğimiz gibi, demetlerle ilgili en sık yapacağınız hata, tek öğeli demet tanımlamaya çalışırken aslında yanlışlıkla bir karakter dizisi tanımlamak olacaktır. Örneğin yukarıdaki TEMPLATE_DIRS değişkenini şöyle yazsaydık:

TEMPLATE_DIRS = ('/home/projects/djprojects/blog/templates')

6. Demetlerin Metotları

Daha önce de bahsedildiği gibi, listeler ve demetler birbirine benzer. Aralarındaki en önemli fark, listelerin değiştirilebilir bir veri tipi iken, demetlerin değiştirilemez bir veri tipi olmasıdır. Elbette bu fark, iki veri tipinin metotlarında da kendini gösterir. Demetler üzerinde değişiklik yapamadığımız için, bu veri tipi değişiklik yapmaya yarayan metotlara sahip değildir.

Demetlerin hangi metotları olduğunu şu komutla görebilirsiniz:

dir(demet)

['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', **'count', 'index'**]

Buradan da anlaşılacağı üzere demetler üzerinde bizi ildilendirmekte olan sadece 2 tane metot vardır.

6.1. count metodu

Karakter dizileri, listeler ve demetlerin ortak metotlarından biri de count() metodudur. Tıpkı karakter dizileri ve listelerde olduğu gibi, demetlerin count() metodu da bir öğenin o veri tipi içinde kaç kez geçtiğini söylemektedir.

```
In [37]: demet = ('elma', 'armut', 'kiraz')
    ...: """ count metodu"""
    ...: demet.count("elma")
Out[37]: 1
```

Listelerin count() metoduyla ilgili söylediğimiz her şey demetlerin count() metodu için de geçerlidir.

6.2. index metodu

Listeler ve karakter dizileri konusunu anlatırken bu veri tiplerinin index() adlı bir metodu olduğundan söz etmiştik hatırlarsanız. İşte demet veri tipinin de index() adında ve listelerle karakter dizilerinin index() metoduyla aynı işi yapan bir metodu bulunur. Bu metot bir demet öğesinin demet içindeki konumunu söylemektedir.

```
In [38]: """ index metodu """
    ...: demet = ('elma', 'armut', 'kiraz')
    ...: demet.index("elma")
Out[38]: 0
```

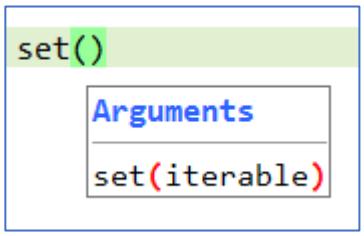
PYTHON'DA KÜMELER (SETS)

Bu bölümde, Pythondaki veri tipleri yolculuğumuza kümeler ve kümelerin özel bir formu olan dondurulmuş kümeleri inceleyeceğiz.

Tıpkı listeler, demetler, sözlükler gibi kümeler de Pythonda çok sayıda değer içeren veri tiplerinden biridir. Adından da az çok tahmin edebileceğiniz gibi kümeler, matematikten bildiğimiz "küme" kavramıyla sıkı sıkıya bağlantılıdır. Bu veri tipi, matematikteki kümelerin sahip olduğu bütün özellikleri taşır. Yani matematikteki kümelerden bildiğimiz kesişim, birleşim ve fark gibi özellikler Pythondaki kümeler için de geçerlidir.

1. Küme Tanımlamak

Listeler, demetler ve sözlüklerin aksine kümelerin ayırt edici bir işareti yoktur. Bununla beraber sözlük tanımlanırken faydalanılan "{ }" küme parantezleri de küme oluşturmak için istisnai bir durumda kullanılacaktır. Küme oluşturmak için set() adlı özel bir fonksiyondan yararlanılmaktadır.



Set fonksiyonun argümanının iterable yani iterasyona müsait yapıda bir nesne olduğunu görülmektedir.

Bu nedenle set fonksiyonunu kullanırken argümanına dikkat edilmelidir. Tıpkı listeler ve demetlerde de olduğu gibi yinelenilebilir yapıdaki nesneler aracılığıyla küme oluşturulmaktadır.

```
In [2]: boş_küme =set()
   ...: type(boş_küme)
Out[2]: set
```

Set fonksiyonu aracılığıyla tanımlanmış boş kümenin hangi veri tipinde olduğunu belirlemek adına type fonksiyonunu kullarak küme oluşturmuş olduğunu teyit etmiş oluyoruz.

Yinelenebilir/tekralanabilir(iterable) bir değer üzerinden küme yaratalım. Bu amaçla bir karakter dizisi kullanılsın.

```
In [14]: küme = set("python") # karakter dizisinden küme yaratalım.
    ...: type(küme)
    ...: print(küme)
{'o', 'n', 'p', 'h', 't', 'y'}
```

Kümelerin indislerinin olup olmadığını teyit edelim.

```
In [16]: küme[-1] # kümenin indekslerinin olmadığını görelim.
Traceback (most recent call last):

  File "<ipython-input-16-ab7232e419be>", line 1, in <module>
    küme[-1] # kümenin indekslerinin olmadığını görelim.

TypeError: 'set' object does not support indexing
```

Görüldüğü üzere kümeler objesinin indislemeyi desteklemediğini teyit etmiş oluruz.

Liste üzerinden bir küme yaratılsın.

```
In [18]: küme = set(list("python")) # listeleştirilen karakter dizisinde bir
küme oluşturalım
    ...: type(küme)
Out[18]: set
```

```
In [19]: print(küme)
{'o', 'n', 'p', 'h', 't', 'y'}
```

```
In [22]: küme = set([1,2,3,4]) # listeden küme 2

In [23]: küme
Out[23]: {1, 2, 3, 4}
```

Demet üzerinden bir küme yaratılsın.

```
In [28]: demet = (1,2,3,4,5) # önce demet tanımlansın
    ...: küme = set(demet)
    ...: küme
Out[28]: {1, 2, 3, 4, 5}

In [29]: type(küme)
Out[29]: set
```

Sözlük üzerinden küme tanımlanması:

```
In [30]: sözlük = {"kitap":"book",
    ...:           "lecture":"programming",
    ...:           "university":"omü"
    ...:         }  # önce sözlük tanımlansın
    ...: küme = set(sözlük)
    ...: küme
Out[30]: {'kitap', 'lecture', 'university'}

In [31]: type(küme)
Out[31]: set
```

Sözlük üzerinden tanımlanan kümelerin yalnızca sözlüğün anahtar değerlerinden oluştuğunu görmüş olduk.

int,float gibi sayısal değerler üzerinden küme oluşturulması:

```
In [32]: sayı = 5
    ...: küme = set(sayı)
Traceback (most recent call last):

  File "<ipython-input-32-77f3ef167f08>", line 2, in <module>
    küme = set(sayı)

TypeError: 'int' object is not iterable
```

Bu örnek bizlere int/float gibi sayısal nesnelerin tekrarlanabilir veri formatına uymadığını göstermiş oldu.

Küme oluşturmanın bir yönteminden daha söz edelim. Konunu başında da bahsedildiği gibi, listeler, demetler, sözlükler ve karakter dizilerinin aksine kümelerin [ ], ( ), { } gibi ayırt edici bir işareti yoktur. Ama eğer istersek sözlükleri oluşturmak için kullandığımız özel işaretleri küme oluşturmak için de kullanabiliriz.

```
In [7]: boş_küme = {}
   ...: type(boş_küme) # böylece sözlük yaratılmış olduğunu gördük.
Out[7]: dict
```

Boş küme yaratabilmek adına kullanılan küme parantezlerinin sözlük yaratmış olduğunu ve küme parantezlerinin öncelikle sözlük tanımlanması üzerine algılandığını gösterilmektedir.

```
In [10]: küme = {1,2,3,4} # küme içerisine eleman tanımlanması
    ...: type(küme)
Out[10]: set
```

```
In [35]: küme
Out[35]: {1, 2, 3, 4}
```

Boş küme dışında küme tanımlamanın küme parantezleri aracılığıyla yapılabildiğini görmüş olduk.

2. Kümelerin Yapısı

Kümelerin yapısına bakıldığında daha öncede belirtmiş olduğumuz gibi bir veri koleksiyonudur. Kümeler sırası olmayan, indislenemeyen ve tekrarsız elemanlardan oluşan bir koleksiyondur. Bir önceki aşamada kümeyi tanımladıktan sonra onların şimdi kümelerin yapısını incelemeye çalışalım:

```
In [36]: karakter_dizisi = "Python Programlama Dili"
    ...: küme = set(karakter_dizisi)
    ...: print(küme)
{'o', 'g', 'n', 'a', 'm', 'l', 'D', 'P', ' ', 'h', 'r', 't', 'y', 'i'}
```

Bu örnekten yapacağımız yorum şu şekildedir. Küme elemanlarının tıpki matematikteki küme elemanları gibi bir kere yazılacağını keşfetmiş olduk. Bunu ayrıca count metodu üzerinden de göreceğiz. Ayrıca elemanlarının bir sırasının olmadığını da görmüş olduk.

Küme elemanlarını seçmek için indeks kullanımının mümkün olmadığını tekrar hatırlamış olalım.

```
In [37]: küme [-1]
Traceback (most recent call last):

  File "<ipython-input-37-c6bfbacb06a5>", line 1, in <module>
    küme [-1]

TypeError: 'set' object does not support indexing
```

Yine kümenin elemanlarını seçmeye çalıştığımızda hata almaktayız.

```
In [38]: küme ["P"]
Traceback (most recent call last):

  File "<ipython-input-38-35604a3838da>", line 1, in <module>
    küme ["P"]

TypeError: 'set' object is not subscriptable
```

Kümenin elemanlarını seçmek adına döngülerden faydalanabiliriz.

3. Kümelerin Metotları

Daha önceki veri tiplerinde olduğu gibi, kümelerin de metotları vardır. Artık biz bir veri tipinin metotlarını nasıl listeleyeceğimizi çok iyi biliyoruz. Nasıl liste için list(); demet için tuple(); sözlük için de dict() fonksiyonlarını kullanıyorsak, kümeler için de set() adlı fonksiyondan dir() fonksiyonu içerisinde yararlanacağız ve aşağıdaki metotlara erişmiş olacağız ve koyu renkte belirtilmiş temel metodları tanıyacağız.

['__and__','__class__','__contains__','__delattr__','__dir__','__doc__','__eq__','__format__','__ge__','__getattribute__','__gt__','__hash__','__iand__','__init__','__ior__','__isub__','__iter__','__ixor__',

'__le__','__len__','__lt__','__ne__','__new__','__or__','__rand__','__reduce__','__reduce_ex__','__repr__','__ror__','__rsub__','__rxor__','__setattr__','__sizeof__','__str__','__sub__','__subclasshook__','__xor__',**'add','clear','copy','difference','difference_update','discard','intersection','intersectio_update','isdisjoint','issubset','issuperset','pop','remove','symmetric_difference','symmetric_difference_update','union','update'**]

Metot	**Açıklaması**
add	Kümeye yeni bir eleman / öğe ekler.
clear	Kümedeki tüm elemanları siler.
copy	Kümenin kopyasını oluşturur.
difference	İki veya daha fazla küme arasındaki farkı içeren bir küme verir.
difference_update	Bu kümedeki, belirtilen başka bir kümede de bulunan öğeleri kaldırır.
discard	Belirtilen öğeyi kaldırır.
intersection	Diğer iki kümenin kesişim noktası olan bir küme verir.
intersection_update	Bu kümedeki diğer, belirtilen kümelerde bulunmayan öğeleri kaldırır
isdisjoint	İki kümede kesişme olup olmadığını döndürür.
issubset	Başka bir kümenin bu kümeyi içerip içermediğini döndürür.
issuperset	Bu kümenin başka bir küme içerip içermediğini döndürür.

pop	Kümeden bir öğe kaldırır / siler.
remove	Belirtilen öğeyi kaldırır.
symmetric_difference	İki kümenin simetrik farklarıyla bir küme verir.
symmetric_difference_update	Bu kümeden ve diğerinden simetrik farkları ekler.
union	Kümelerin birleşimini içeren bir küme döndürür.
update	Seti bu ve diğerlerinin birleşimiyle günceller.

3.1 add metodu

Geçtiğimiz haftalarda işlediğimiz veri tiplerinin bazılarının değiştirilebilir bazılarınınsa değişime müsait yapıda olmadıklarını görmüştük. Kümelerse, değişime müsait yapıda bir veri tipidir. Adından da anlaşılacağı gibi, bu metot yardımıyla kümelerimize yeni öğeler ilave edebileceğiz. Hemen bunun nasıl kullanıldığına bakalım:

```
In [50]: küme = {"elma","armut", "muz" }
    ...: type(küme)
Out[50]: set

In [51]: küme.add("karpuz") # karpuz meyvesini ekleyelim
    ...: print(küme)
{'armut', 'muz', 'karpuz', 'elma'}
```

Örnekte kümemize bir karakter dizisi olan "karpuz" meyvesi eklendi ve kümemiz bu şekilde güncellenmiş oldu. Geçtiğimiz haftalardan hatırlanacağı üzere karakter dizileri değiştirilemeyen veri tipidir.

Yani, değiştirilemeyen bir nesneyi kümemize rahatlıkla eklemiş olduk peki veri eklemek için değiştirilebilen yapıda olan listeyi deneyelim:

```
In [54]: küme.add(["çilek","kiraz"]) # kümemize bir liste eklemeyi deneyelim.
Traceback (most recent call last):

  File "<ipython-input-54-a579f8f928bd>", line 1, in <module>
    küme.add(["çilek","kiraz"]) # kümemize bir liste eklemeyi deneyelim.

TypeError: unhashable type: 'list'
```

Kodlarımızı çalıştırdığımızda kümeye bir liste eklemenin bizi hataya götürdüğü görülmüş oldu. Bunun nedeniyse "hashable" yani değiştirilemez bir veri tipinin kümeye eklenmemesidir. Liste değiştirilebilir bir nesne olması sebebiyle kümeye eklenememiştir. Bunu da uzun bir örnekle somutlaştıralım:

```
In [69]: küme = set()
    ...: küme.add("matematik") # kümeye string yapıda bir nesne eklensin
    ...: print("kümeye string ekleyip yazdıralım:",küme)
    ...:
    ...: küme.add(1) # kümeye int/float eklensin
    ...: print("kümeyi int/float ekleyip yazdıralım:",küme)
    ...:
    ...: küme.add(("fizik","kimya","biyoloji"))
    ...: print("kümeye demet ekleyip yazdıralım:",küme)
kümeye string ekleyip yazdıralım: {'matematik'}
kümeyi int/float ekleyip yazdıralım: {1, 'matematik'}
kümeye demet ekleyip yazdıralım: {1, ('fizik', 'kimya', 'biyoloji'),
'matematik'}
```

Bu örnekten int-float/string ve tuple tipindeki verilerin eklenebildiğini görmekteyiz. Öte yandan, kümeye liste,demet ve sözlük eklenmeye çalışıldığında hata almaktayız:

```
In [71]: küme.add(dict({1:"elma",2:"karpuz"})) # kümeye sözlük ekleme denemesi
Traceback (most recent call last):

  File "<ipython-input-71-a59d2619e56a>", line 1, in <module>
    küme.add(dict({1:"elma",2:"karpuz"})) # kümeye sözlük ekleme denemesi

TypeError: unhashable type: 'dict'
```

```
In [72]: küme.add(list("dersler")) # kümeye liste ekle denemesi
Traceback (most recent call last):

  File "<ipython-input-72-72b0cae1f2fb>", line 1, in <module>
    küme.add(list("dersler")) # kümeye liste ekle denemesi

TypeError: unhashable type: 'list'
```

```
In [73]: küme.add(set("dersler")) # kümeye küme ekleme denemesi
Traceback (most recent call last):

  File "<ipython-input-73-c9d700bfc480>", line 1, in <module>
    küme.add(set("dersler")) # kümeye küme ekleme denemesi

TypeError: unhashable type: 'set'
```

3.2. clear metodu

Bu metodu daha önce diğer veri tipleri üzerinde çalışırken de görmüştük. Bu metodun görevi uygulandığı kümenin içini boşaltmaktır. Örnekle bu metodu somutlaştıralım:

```
In [77]: küme = set("samsun")
    ...: print(küme)
    ...: küme.clear()
    ...: print(küme)
{'a', 'm', 'u', 's', 'n'}
set()
```

3.3. copy metodu

Bu metodu daha önce diğer veri tipleri üzerinde çalışırken de görmüştük. Bu metodun görevi uygulandığı kümenin bir kopyasını oluşturmaktır. Örnekle bu metodu somutlaştıralım:

```
In [79]: küme = set("samsun")
    ...: print(küme)
    ...: yedek = küme.copy()
    ...: print(yedek)
{'a', 'm', 'u', 's', 'n'}
{'a', 'm', 'u', 's', 'n'}
```

3.4. difference metodu

Bu metot iki kümenin farkını almamızı sağlar.

```
In [83]: küme1 = {1,2,3,4}
    ...: küme2 = {2,4,5,6}
    ...: küme1.difference(küme2) # küme 1 ' in küme 2'den farkı
Out[83]: {1, 3}

In [84]: küme2.difference(küme1) # küme 2'nin küme 1'den farkı
Out[84]: {5, 6}
```

Aynı işlevi "-" operatörü ile de gerçekleştirebiliriz.

```
In [87]: küme1-küme2
Out[87]: {1, 3}

In [88]: küme2-küme1
Out[88]: {5, 6}
```

Fakat "+" operatörü kümeler için birleştirme ya da toplama işlevi yapmaz.

```
In [89]: küme1+küme2
Traceback (most recent call last):

  File "<ipython-input-89-a16ca585d74e>", line 1, in <module>
    küme1+küme2

TypeError: unsupported operand type(s) for +: 'set' and 'set'
```

3.5. difference_update metodu

Bu metot, difference() metodundan elde edilen sonuca göre bir kümenin güncellenmesini sağlar.

```
In [91]: küme1 = {1,2,3,4}
    ...: küme2 = {2,4,5,6}
    ...: küme1.difference_update(küme2)
    ...: küme1
Out[91]: {1, 3}
```

```
In [92]: küme1 = {1,2,3,4}
    ...: küme2 = {2,4,5,6}
    ...: küme2.difference_update(küme1)
    ...: küme2
Out[92]: {5, 6}
```

3.6. discard metodu

Daha önce öğrendiğimiz add() metodu yardımıyla, önceden oluşturduğumuz bir kümeye yeni öğeler ekleyebiliyorduk. Bu bölümde öğreneceğimiz discard() metodu ise kümeden öğe silmemizi sağlar.

```
In [98]: küme = {"samsun","sinop","ordu","çorum"}
    ...: print(küme)
    ...: küme.discard("sinop")
    ...: print(küme)
{'ordu', 'samsun', 'sinop', 'çorum'}
{'ordu', 'samsun', 'çorum'}
```

Eğer küme içinde bulunmayan bir öğe silmeye çalışırsak hiç bir şey olmaz. Yani hata mesajı almayız.

```
In [99]: küme.discard("trabzon")
    ...: print(küme)
{'ordu', 'samsun', 'çorum'}
```

3.7. intersection metodu

intersection kelimesi Türkçe'de "kesişim" anlamına gelir. Adından da anladığımız gibi, intersection() metodu bize iki kümenin kesişim kümesini verir.

```
In [102]: küme1 = {1,2,3,4}
     ...: küme2 = {2,4,5,6}
     ...: kesişim = küme1.intersection(küme2)
     ...: kesişim
Out[102]: {2, 4}
```

Görüldüğü üzere, küme 1 ve 2 nin kesişimi vasıtasıyla yeni bir kesişim kümesi yaratılır.

difference() metodunu anlatırken, difference() kelimesi yerine "-" işaretini de kullanabileceğimiz, söylemiştik. Benzer bir durum intersection() metodu için de geçerlidir. İki kümenin kesişimini bulmak için "&" işaretinden yararlanabiliriz.

```
In [104]: küme1 = {1,2,3,4}
     ...: küme2 = {2,4,5,6}
     ...: küme3 = küme1&küme2
     ...: küme3
Out[104]: {2, 4}
```

3.8. intersection_update metodu

Hatırlarsanız difference_update() metodunu işlerken şöyle bir şey demiştik. Bu metot, difference() metodundan elde edilen sonuca göre bir kümenin güncellenmesini sağlar. İşte intersection_update metodu da buna çok benzer bir işlevi yerine getirir. Bu metodun görevi, intersection() metodundan elde edilen sonuca göre bir kümenin güncellenmesini sağlamaktır.

```
In [107]: küme1 = {1,2,3,4}
     ...: küme2 = {2,4,5,6}
     ...: küme1.intersection_update(küme2)
     ...: print(küme1)
     ...: küme2.intersection_update(küme1)
     ...: print(küme2)
{2, 4}
{2, 4}
```

Görüldüğü gibi, intersection_update() metodu küme1'in bütün öğelerini sildi ve yerlerine küme1 ve küme2'nin kesişim kümesinin elemanlarını koydu. Aynı işlemin sonucunda küme2'nin bütün öğelerini sildi ve yerlerine küme1 ve küme2'nin kesişim kümesinin elemanlarını koydu.

3.9. isdisjoint metodu

Bu metodun çok basit bir görevi vardır. isdisjoint() metodunu kullanarak iki kümenin kesişim kümesinin boş olup olmadığı sorgulayabilirsiniz. Hatırlarsanız aynı işi bir önceki bölümde gördüğümüz intersection() metodunu kullanarak da yapabiliyorduk. Ama eğer hayattan tek beklentiniz iki kümenin kesişim kümesinin boş olup olmadığını, yani bu iki kümenin ortak eleman içerip içermediğini öğrenmekse, basitçe isdisjoint() metodundan yararlanabiliriz.

```
In [109]: küme1 = {1,2,3,4}
     ...: küme2 = {2,4,5,6}
     ...: küme1.isdisjoint(küme2)
Out[109]: False
```

Görüldüğü gibi, 2 kümenin kesişimi boş olmadığı için, yani bu iki küme ortak en az bir öğe barındırdığı için, isdisjoint() metodu False çıktısı veriyor. Burada temel olarak şu soruyu sormuş oluyoruz: **küme1 ve küme2 ayrık kümeler midir?**

Python da bize cevap olarak, "*Hayır değil,*" anlamına gelen False çıktısını veriyor. Çünkü küme1 ve küme2 kümelerinin ortak olan 2 elemanı vardır (2 ve 4).

3.10. issubset metodu

Bu metot yardımıyla, bir kümenin bütün elemanlarının başka bir küme içinde yer alıp yer almadığını sorgulayabiliriz. Yani bir kümenin, başka bir kümenin alt kümesi olup olmadığını bu metot yardımıyla öğrenebiliriz. Eğer bir küme başka bir kümenin alt kümesi ise bu metot bize True değerini verecek; eğer değilse False çıktısını verecektir.

```
In [113]: a = set([1, 2, 3])
     ...: b = set([0, 1, 2, 3, 4, 5])
     ...: a.issubset(b)
Out[113]: True

In [114]: b.issubset(a)
Out[114]: False
```

Bu örnekte True çıktısını aldık, çünkü a kümesinin bütün öğeleri b kümesi içinde yer alıyor. Yani a, b'nin alt kümesidir.

3.11. issuperset metodu

Bu metot, bir önceki bölümde gördüğümüz issubset() metoduna benzer. Matematik derslerinde gördüğümüz "kümeler" konusunda hatırladığınız "b kümesi a kümesini kapsar" ifadesini bu metotla gösteriyoruz. Önceki örneğimizden bu metoda bakalım.

```
In [116]: a = set([1, 2, 3])
     ...: b = set([0, 1, 2, 3, 4, 5])
     ...: b.issuperset(a)
Out[116]: True
```

Biz a kümesinin b kümesinin alt kümesi olduğunu görmüştük. Bu metod aracılığıyla da b kümesi a kümesini kapsamaktadır deriz.

3.12. pop metodu

Bu metot listenin bir öğesini silip ekrana basar. Peki bu metot hangi ölçüte göre kümeden öğe siliyor? Herhangi bir ölçüt yok. Bu metot, küme öğelerini tamamen rastgele silmektedir.

```
In [120]: b = set([0, 1, 2, 3, 4, 5])
     ...: b.pop()
Out[120]: 0

In [121]: print(b)
{1, 2, 3, 4, 5}
```

3.13. remove metodu

Bu metot aracılığıyla listenin tanımlanmış bir öğesini silmemizi sağlar.pop metodundan farklı yanı bu metot aracılığıyla tanımlanmış bir eleman silinir.

```
In [124]: b = set([0, 1, 2, 3, 4, 5])
     ...: b.remove(0)
     ...: print(b)
{1, 2, 3, 4, 5}
```

3.14. symetric_difference metodu

difference() metodunu kullanarak iki küme arasındaki farklı öğeleri bulmayı öğrenmiştik. Peki ya kümelerin sadece birinde bulunan öğeleri nasıl alacağız? İşte bu noktada yardımımıza symmetric_difference() adlı metot yetişecek. Böylece iki kümenin ortak olarak sahip olmadığı öğeleri tanımlayabiliriz.

```
In [130]: a = set([1, 2, 3])
     ...: b = set([0, 1, 2, 3, 4, 5])
     ...: a.symmetric_difference(b)
Out[130]: {0, 4, 5}
```

3.15. symmetric_difference_update metodu

Daha önce kullanılan difference ve difference_update metotları arasındaki ilişkli symmetric_difference ve symmetric_difference_update arasında vardır.

```
In [134]: a = set([1, 2, 3])
     ...: b = set([0, 1, 2, 3, 4, 5])
     ...: a.symmetric_difference_update(b)
     ...: a
Out[134]: {0, 4, 5}
```

İki kümenin kesişiminde olmayan elemanlar tespit edilip belirtilmek istenen başka bir kümenin içerisine yazılır.

3.16. union metodu

Union kelimesi birleşim demektir. Yani kümelerin birleşimini elde etmek amacıyla bu metottan faydalanılır.

```
In [135]: küme1 = {0,2,4,6}
     ...: küme2 = {1,2,3,5}
     ...: küme1.union(küme2)
Out[135]: {0, 1, 2, 3, 4, 5, 6}
```

Bazı metotlarda olduğu gibi, union() metodu da bir kısayola sahiptir. union() metodu yerine "|" işaretini de kullanabiliriz.

```
In [136]: küme1|küme2
Out[136]: {0, 1, 2, 3, 4, 5, 6}
```

3.17. update metodu

```
In [145]: küme1 = {0,2,4,6}
     ...: küme2 = {1,2,3,5}
     ...: küme1.update(küme2)

In [146]: küme1
Out[146]: {0, 1, 2, 3, 4, 5, 6}
```

Add metodu aracılığıyla kümelere eleman eklemeyi görmüştük. Fakat bu metot tek eleman eklemeye yaramaktadır. Kümelere küme yada daha fazla sayıda eleman eklemek yani başka bir küme eklemek için update metodundan faydalanılabilir.

4. Dondurulmuş Kümeler (Frozenset)

Daha önce de söylediğimiz gibi, kümeler üzerinde değişiklik yapabiliyoruz. Zaten kümelerin add() ve remove() gibi metotlarının olması bu durumu teyit emektedir. Ancak bazı durumlarda, öğeleri üzerinde değişiklik yapılamayan kümelere de ihtiyaç duyabiliriz. Önceki haftalarda işlediğimiz listeler ve demetler arasında da buna benzer bir ilişki vardı. Demetler çoğu zaman, üzerinde değişiklik yapılamayan bir liste gibi davranır. İşte Python aynı imkanı bize kümelerde de sağlar. Eğer öğeleri üzerinde değişiklik yapılamayan bir küme oluşturmak istenirse set() yerine frozenset() fonksiyonunu kullanabilmekteyiz.

Dir fonksiyonu aracılığıyla dondurulmuş kümelerin metotlarını yazdırırsak kümelerden daha az sayıda metodun olduğunu keşfederiz.

['__and__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__ne__', '__new__', '__or__', '__rand__', '__reduce__', '__reduce_ex__', '__repr__', '__ror__', '__rsub__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__xor__', **'copy', 'difference', 'intersection', 'isdisjoint', 'issubset', 'issuperset', 'symmetric_difference', 'union']**

Bu komutlar/metotlar arasında add,remove ve update gibi değişikliğe yönelik metotlarının olmadığı gözümüze çarpmaktadır. Bu da dondurulmuş kümelerin değiştirilemedeğinin bir ipucu olarak karşımıza çıkmaktadır. Dondurulmuş kümeler ile normal kümeler arasında işlev olarak hiçbir fark yoktur. Bu ikisi arasındaki fark, listeler ile demetler arasındaki fark gibidir.

PYTHON'DA SÖZLÜKLER

Sözlükler, gerçek hayattaki sözlükler gibi davranan bir veri tipi olarak bilinmektedir. İçindeki her bir eleman indeks olarak değil, anahtar(key) ve değer (value) olarak tutulur. Anahtar değer çiftleri virgülle ayrılarak bir öğeyi oluştururlar. Değer kısmı bütün veri türünü içerebilir fakat anahtar kısmı sadece string ve int tipinde olabilir. Sözlükler sırasız bir öge koleksiyonudur. Anahtar ve değer yapısında veri kümeleri oluştururlar. Değerler herhangi bir veri türüne ait olabilir ve tekrarlanabilir fakat anahtarlar kesinlikle benzersiz olmalıdır.

1. Sözlük Oluşturma

Sözlük oluştururken dikkat edilmesi gerekenler:

- İki noktayı ve virgülü nereye koyulduğuna
- Ögelerimizi tanımlarken ayraç parantez () yerine Küme Parantezi {} kullanıldığına
- Anahtar-değer ilişkisine,
- Anahtar değeri tanımlarken tırnak işareti kullanıldığına dikkat ediniz.

Aşağıdaki örnekte, süslü parantez ve iki nokta ile anahtar değerlerinizi yerleştirerek kişi bilgilerini içeren bir sözlük oluşturulmuştur.

```
kişisel_bilgiler={"ad":"ali","telefon":"05554442211","e_posta":"ali@abc.com",
                  "ülke":"Türkiye","il":"Samsun","ilçe":"Atakum"}
print(kişisel_bilgiler)
```

```
In [4]: kişisel_bilgiler={"ad":"ali","telefon":"05554442211","e_posta":"ali@abc.com",
   ...:                   "ülke":"Türkiye","il":"Samsun","ilçe":"Atakum"}
   ...: print(kişisel_bilgiler)
{'il': 'Samsun', 'e_posta': 'ali@abc.com', 'ad': 'ali', 'ülke': 'Türkiye', 'ilçe': 'Atakum',
'telefon': '05554442211'}
```

Aşağıdaki örnekte çeşitli sözlük oluşturma yöntemleri örneklendirilmiştir.

```
In [5]: sözlük1 = {}
   ...: print(sözlük1)
{}
```

Boş Sözlük oluşturma

```
In [6]: sözlük2= {1: 'adana', 2: 'adıyaman'}
   ...: print(sözlük2)
{1: 'adana', 2: 'adıyaman'}
```

Farklı tipte Anahtar -Değer ikilileri

```
In [7]: sözlük3 = {'isim': 'ali', 1: [5, 4, 3]}
   ...: print(sözlük3)
{'isim': 'ali', 1: [5, 4, 3]}
```

Farklı tipte Anahtar -Değer ikilileri

```
In [8]: sözlük4 = dict({1:'erik', 2:'ayva'})
   ...: print(sözlük4)
{1: 'erik', 2: 'ayva'}
```

dict fonksiyonu ile sözlük oluşturma

Sözlükler anahtar-değer ikililerinden oluşmuş oldukları için içerdikleri öğe sayısı kullanıcıyı zorlayabilir. Bu durumu çözümlemek adına len() fonksiyonundan faydalanılabilir.

```
In [16]: print(sözlük4)
    ...: #sözlükteki eleman sayısı / ya da uzunluğu
    ...: print(len(sözlük4))
{1: 'erik', 2: 'ayva'}
2
```

Görüldüğü üzere en son oluşturulan 4. Sözlükteki öğe sayısı 2 olarak bizlere sunulmuştur. Yani 1 ile erik birlikte tek bir öğeyi oluşturur. Aynı şekilde 2 ile ayva kelimeleri de beraberce tek bir eleman olarak görülür.

Python programlama dilinde doğru kod yazmak kadar okunaklı kod yazmak da çok önemlidir. Mesela bir sözlüğü şöyle tanımladığımızda kodlarımızın pek okunaklı olmayacağını söyleyebiliriz:

sözlük = {"kitap": "book", "bilgisayar": "computer", "programlama": "programming","dil": "language", "defter": "notebook"}

```
In [17]: sözlük = {"kitap": "book", "bilgisayar": "computer", "programlama":
"programming","dil": "language", "defter": "notebook"}
    ...: print(sözlük)
{'programlama': 'programming', 'kitap': 'book', 'bilgisayar': 'computer', 'dil':
'language', 'defter': 'notebook'}
```

Teknik olarak bakıldığında bu kodlarda hiçbir problem yoktur. Ancak sözlükleri böyle sağa doğru uzayacak şekilde tanımladığımızda okunaklılığı azaltmış oluyoruz. Bu yüzden yukarıdaki sözlüğü şöyle yazmayı tercih edebiliriz:

sözlük = {"kitap" : "book",

"bilgisayar" : "computer",

"programlama": "programming",

"dil" : "language",

"defter" : "notebook"}

```
In [18]: sözlük = {"kitap"      : "book",
    ...:           "bilgisayar" : "computer",
    ...:           "programlama": "programming",
    ...:           "dil"        : "language",
    ...:           "defter"     : "notebook"}
    ...: print(sözlük)
{'programlama': 'programming', 'kitap': 'book', 'bilgisayar': 'computer', 'dil':
'language', 'defter': 'notebook'}
```

Sözlük içinde listelere de yer verilebilmektedir. Aşağıdaki örnekle bu durumu somutlaştıralım.

```
In [21]: sözlük = {"Ahmet" : ["İstanbul", "Öğretmen", 34],
    ...:           "Mehmet" : ["Adana", "Mühendis", 40],
    ...:           "Sedat" : ["İskenderun", "Doktor", 30]}

In [22]: print(sözlük)
{'Sedat': ['İskenderun', 'Doktor', 30], 'Mehmet': ['Adana', 'Mühendis', 40],
'Ahmet': ['İstanbul', 'Öğretmen', 34]}

In [23]: print(sözlük["Ahmet"])
['İstanbul', 'Öğretmen', 34]
```

2. Sözlük Öğelerine Erişmek

Yukarıdaki örneklerden bir sözlüğün en basit şekilde nasıl tanımlanacağı görülmüştür. Peki tanımladığımız bir sözlüğün öğelerine nasıl erişim sağlayabilirirz ? Bunu yapabilmek adına tanımlanan sözlükte köşeli parantezler içerise anahtar(key) yazılırsa bu anahtara ait değere erişebiliriz. Aşağıdaki örnekle bu durumu somutlaştıralım.

```
In [19]: sözlük = {"kitap"      : "book",
    ...:           "bilgisayar" : "computer",
    ...:           "programlama": "programming",
    ...:           "dil"        : "language",
    ...:           "defter"     : "notebook"}
    ...: print(sözlük["kitap"])
book
```

Kitap ile belirtilmiş anahtarı karşılık gelen değer book olarak yazdırılmıştır. Sözlüklerde anahtar kelime üzerinden bu seçim yapılmakla beraber daha önce gördüğümüz karakter dizileri ve listelerde sıralı bir şekilde seçim yapabilmek mümkün değildir.

```
In [29]: sözlük[1]
Traceback (most recent call last):

  File "<ipython-input-29-0073a541569f>", line 1, in <module>
    sözlük[1]

KeyError: 1

Traceback (most recent call last):

  File "<ipython-input-29-0073a541569f>", line 1, in <module>
    sözlük[1]

KeyError: 1
```

Buradan anlaşıldığı üzere sözlüklerde sıra kavramı bulunmamaktadır.

3. Sözlüğe Öğe Eklemek

Tıpkı listeler gibi, sözlükler de büyüyüp küçülebilen bir veri tipidir. Yani bir sözlüğü ilk kez tanımladıktan sonra istediğimiz zaman bu sözlüğe yeni öğeler ekleyebilir veya varolan öğeleri çıkarabiliriz.

Bir sözlüpe öğe eklemek için şöyle bir formül kullanılacaktır:

>>> sözlük[anahtar] = değer

Bu yöntemi bir de örnek ile somutlaştıralım.

```
In [34]: sözlük = {}
    ...: sözlük["1"] = "omü"
    ...: sözlük[2] = "odtü"

In [35]: sözlük
Out[35]: {2: 'odtü', '1': 'omü'}
```

Görüldüğü gibi eklemek istediğimiz öğeler sözlüğe eklenmiş. Ancak ikinci ilave edilen öğenin sözlüğün en sonuna değil, sözlük içine rastgele bir şekilde yerleştirildiğini gözlemliyoruz. Çünkü, dediğimiz gibi, sözlükler sırasız bir veri tipidir.

```
In [36]: sözlük = {}

In [37]: sözlük = {'a': 1}

In [38]: sözlük
Out[38]: {'a': 1}

In [39]: sözlük = {'a': (1,2,3)}

In [40]: sözlük
Out[40]: {'a': (1, 2, 3)}

In [41]: sözlük = {'a': 'alfabe'}

In [42]: sözlük
Out[42]: {'a': 'alfabe'}

In [43]: sözlük = {'a': [1,2,3]}

In [44]: sözlük
Out[44]: {'a': [1, 2, 3]}
```

Bir sözlüğe değer olarak bütün veri tiplerini verebiliriz.

Boş bir sözlüğe değer atanırken sırasıyla sayısal,demet,string,liste tipinde veri tipleri atanmıştır.

Ama durum sözlük anahtarları açısından böyle değildir. Yani sözlüklere anahtar olarak her veri tipini atayamayız. Bir değerin ‘anahtar’ olabilmesi için, o öğenin değiştirilemeyen (*immutable*) bir veri tipi olması gerekir.

Bizler şu ana dek karakter dizilerinin(string) ve sayısal ifadelerin değiştirilemeyen olduklarını gördük. Bu özelliğe sahip demetler ise bir sonraki konumuzdur.

Değiştirilebilen veri tiplerine ise diziler ve sözlükleri söyleyebiliriz.

Şimdi bu konuştuklarımızı örneklerle somut hale getirelim.

```
In [45]: sözlük = {}
    ...: #Bu sözlüğe anahtar olarak bir demet ekleyelim:
    ...: anahtar = (1,2,3)
    ...: sözlük[anahtar] = 'demet'
    ...: sözlük
Out[45]: {(1, 2, 3): 'demet'}
```

Anahtar olarak demet tanımlandı ve sözlüğümüz oluşturuldu.

```
In [46]: anahtar = 45
    ...: sözlük[anahtar] = 'sayı'
    ...: sözlük
Out[46]: {45: 'sayı', (1, 2, 3): 'demet'}
```

Anahtar olarak bir sayı seçildi ve sözlüğümüz tanımlandı.

```
In [47]: anahtar = 'omü'
    ...: sözlük[anahtar] = 'karakter dizisi'
    ...: sözlük
Out[47]: {'omü': 'karakter dizisi', 45: 'sayı', (1, 2, 3): 'demet'}
```

Anahtar olarak karakter dizisi seçildi ve sözlüğümüze ilave edildi.

```
In [48]: anahtar = [1,2,3]
    ...: sözlük[anahtar] = 'liste'
Traceback (most recent call last):

  File "<ipython-input-48-95c5fb7dacb5>", line 2, in <module>
    sözlük[anahtar] = 'liste'

TypeError: unhashable type: 'list'

Traceback (most recent call last):
```

Anahtar olarak liste seçildiğinde ise hata aldık çünkü değiştirilebilen bir veri tipi olan liste sözlük anahtarı olarak atanamamaktadır.

Aşağıda da sözlük seklinde bir anahtar tanımlanmak istenmiştir ve o denememizde de hatayla karşılaşmaktayız.

```
In [49]: anahtar = {"a": 1, "b": 2, "c": 3}
    ...: sözlük[anahtar] = 'anahtar sözlük'
Traceback (most recent call last):

  File "<ipython-input-49-3d49a5468e69>", line 2, in <module>
    sözlük[anahtar] = 'anahtar sözlük'

TypeError: unhashable type: 'dict'
```

4. Sözlük Öğeleri Üzerinde Değişiklik Yapmak

Sözlükler değiştirilebilir veri tipleridir. Dolayısıyla sözlükler üzerinde rahatlıkla istediğimiz değişikliği yapabiliriz. Sözlükler üzerinde değişiklik yapma işlemi, biraz önce öğrendiğimiz, sözlüklere yeni öğe ekleme işlemiyle aynıdır. Örnekle somutlaştıralım.

```
In [52]: notlar = {"ali":45,
    ...:           "veli":56,
    ...:           "erhan":80,}
    ...: notlar
Out[52]: {'ali': 45, 'erhan': 80, 'veli': 56}
```

Şimdi bu sözlükteki 'ali' adlı kişinin 45 olan notunu 65 olarak değiştirelim:

```
In [53]: notlar["ali"]=65
    ...: notlar
Out[53]: {'ali': 65, 'erhan': 80, 'veli': 56}
```

5. Sözlüklerin Metotları

Tıpkı öteki veri tiplerinde olduğu gibi, sözlüklerin de birtakım metotları bulunur. Bu bölümde sözlüklerin şu metotlarını inceleyeceğiz:

- keys()
- values()
- items()
- get()
- clear()
- copy()
- fromkeys()
- pop()
- popitem()
- setdefault()
- update()

İlk olarak keys() metoduyla başlayalım.

5.1. keys()

Sözlükleri tarif ederken, sözlüklerin anahtar-değer çiftlerinden oluşan bir veri tipi olduğu ifade edilmişti. Bir sözlüğü normal yollardan ekrana yazdırırsak bize hem anahtarları hem de bunlara karşılık gelen değerleri verecektir. Ama eğer bir sözlüğün sadece anahtarlarını almak isterseniz keys() metodundan yararlanabilirsiniz:

```
In [55]: notlar.keys()
Out[55]: dict_keys(['veli', 'erhan', 'ali'])
```

Gördüğünüz gibi, notlar.keys() komutu bize bir dict_keys nesnesi veriyor. Bu nesneyi programınızda kullanabilmek için isterseniz, bunu listeye, demete veya karakter dizisine dönüştürebilirsiniz:

```
In [56]: liste = list(notlar.keys())
    ...: liste
Out[56]: ['veli', 'erhan', 'ali']
```

5.2. values()

keys() metodu bir sözlüğün anahtarlarını veriyor. Bir sözlüğün değerlerini ise values() metodu verir:

```
In [57]: notlar.values()
Out[57]: dict_values([56, 80, 65])
```

Yine keys metodunda lduğu gibi elde edilen değerleri bir liste / demet ya da karakter dizileri haline getirebilmek mümkündür.

5.3. items()

Bu metot, bir sözlüğün hem anahtarlarını hem de değerlerini aynı anda almamızı sağlar:

```
In [58]: notlar.items()
Out[58]: dict_items([('veli', 56), ('erhan', 80), ('ali', 65)])
```

5.4. get()

Anahtarın karşılığındaki değeri sorgulamamızı sağlar. Sözlüklerin get() adlı metodu, parantez içinde iki adet argüman alır. Birinci argüman sorgulamak istediğimiz sözlük öğesidir. İkinci argüman ise bu öğenin sözlükte bulunmadığı durumda kullanıcıya hangi mesajın gösterileceğini belirtir.

```
In [63]: notlar.get("ali")
Out[63]: 65

In [64]: notlar.get("umut","bu öğrenci sınava girmemiştir")
Out[64]: 'bu öğrenci sınava girmemiştir'
```

5.5. clear()

Bu kelime İngilizce'de "temizlemek" anlamına gelir. Görevi sözlükteki öğeleri temizlemektir. Yani içi dolu bir sözlüğü bu metot yardımıyla tamamen boşaltabiliriz:

```
In [65]: notlar.clear()

In [66]: notlar
Out[66]: {}
```

Gördüğünüz gibi artık adlı sözlüğümüz bomboş. clear() metodunu kullanarak bu sözlüğün bütün öğelerini sildik. Ama tabii ki bu şekilde sözlüğü silmiş olmadık. Boş da olsa bellekte hâlâ sözlük duruyor. Eğer siz sözlüğü ortadan kaldırmak isterseniz del adlı bir parçacıktan yararlanmanız gerekir:

5.6. copy()

Sözlüğü kopyalamak adına faydalanılır.

```
In [73]: yedek_notlar = notlar.copy()
    ...: yedek_notlar
Out[73]: {'ali': 65, 'erhan': 80, 'veli': 56}
```

5.7. fromkeys()

Bu metot öteki metotlardan biraz farklıdır. Bu metot mevcut sözlük üzerinde işlem yapmaz. fromkeys()'in görevi yeni bir sözlük oluşturmaktır. Bu metot yeni bir sözlük oluştururken listeler veya demetlerden yararlanır.

```
In [74]: elemanlar = "Ahmet", "Mehmet", "Can"
    ...:
    ...: adresler = dict.fromkeys(elemanlar, "Kadıköy")
    ...:
    ...: adresler
Out[74]: {'Ahmet': 'Kadıköy', 'Can': 'Kadıköy', 'Mehmet': 'Kadıköy'}
```

Gördüğünüz gibi öncelikle "elemanlar" adlı bir demet tanımladık. Daha sonra da "adresler" adlı bir sözlük tanımlayarak, fromkeys() metodu yardımıyla anahtar olarak "elemanlar" demetindeki öğelerden oluşan, değer olarak ise "Kadıköy"ü içeren bir sözlük meydana getirdik.

En başta tanımladığımız "elemanlar" demeti liste de olabilirdi. Hatta tek başına bir karakter dizisi dahi yazabilirdik.

5.8. pop()

Bu metodu listelerden hatırlıyoruz. Bu metot listelerle birlikte kullanıldığında, listenin en son öğesini silip, silinen öğeyi de ekrana basıyordu. Eğer bu metodu bir sıra numarası ile birlikte kullanırsak, listede o sıra numarasına karşılık gelen öğe siliniyor ve silinen bu öğe ekrana basılıyordu. Bu metodun sözlüklerdeki kullanımı da az çok buna benzer. Ama burada farklı olarak, pop metodunu argümansız bir şekilde kullanamıyoruz. Yani pop metodunun parantezi içinde mutlaka bir sözlük öğesi belirtmeliyiz:

```
In [75]: sepet = {"meyveler": ("elma", "armut"),
    ...:          "sebzeler": ("pırasa", "fasulye"),}
    ...:
    ...: sepet.pop("meyveler")
Out[75]: ('elma', 'armut')

In [76]: sepet
Out[76]: {'sebzeler': ('pırasa', 'fasulye')}
```

Bu komut, sözlükteki "meyveler" anahtarını silecek ve sildiği bu öğenin değerini ekrana basacaktır. Eğer silmeye çalıştığımız anahtar sözlükte yoksa Python bize bir hata mesajı gösterecektir:

```
In [77]: sepet.pop("tatlılar")
Traceback (most recent call last):

  File "<ipython-input-77-d557ba93ca8c>", line 1, in <module>
    sepet.pop("tatlılar")

KeyError: 'tatlılar'
```

5.9. popitem()

popitem() metodu da bir önceki bölümde öğrendiğimiz pop() metoduna benzer. Bu iki metodun görevleri hemen hemen aynıdır. Ancak pop() metodu parantez içinde bir parametre alırken, popitem() metodunun parantezi boş, yani parametresiz olarak kullanılır.

```
In [81]: sepet
Out[81]: {'meyveler': ('elma', 'armut'), 'sebzeler': ('pırasa', 'fasulye')}

In [82]: sepet.popitem()
Out[82]: ('meyveler', ('elma', 'armut'))

In [83]: sepet
Out[83]: {'sebzeler': ('pırasa', 'fasulye')}
```

5.10. setdefault()

```
In [84]: sepet = {"meyveler": ("elma", "armut"),
    ...:          "sebzeler": ("pırasa", "fasulye"),}
    ...:
    ...: sepet.setdefault("içecekler", ("su", "kola"))
    ...:
    ...: sepet
Out[84]:
{'içecekler': ('su', 'kola'),
 'meyveler': ('elma', 'armut'),
 'sebzeler': ('pırasa', 'fasulye')}
```

Bu komut yardımıyla sözlüğümüz içinde "içecekler" adlı bir anahtar oluşturduk. Bu anahtarın değeri ise ("su", "kola") oldu. Bir de elimizde varolan bir anahtar-değer ikilisine göz atalım.

```
In [85]: sepet.setdefault("meyveler", ("erik", "çilek"))
    ...:
    ...: sepet
Out[85]:
{'içecekler': ('su', 'kola'),
 'meyveler': ('elma', 'armut'),
 'sebzeler': ('pırasa', 'fasulye')}
```

Gördüğünüz gibi, sözlükte zaten "meyveler" adlı bir anahtar bulunduğu için, Python aynı adı taşıyan ama değerleri farklı olan yeni bir "meyveler" anahtarı oluşturmadı. Demek ki bu metot yardımıyla bir sözlük içinde arama yapabiliyor, eğer aradığımız anahtar sözlükte yoksa, setdefault() metodu içinde belirttiğimiz özellikleri taşıyan yeni bir anahtar-değer çifti oluşturabiliyoruz.

5.11. update()

İnceleyeceğimiz son metot update() metodu.Bu metot yardımıyla oluşturduğumuz sözlükleri yeni verilerle güncelleyeceğiz.

```
In [86]: notlar
Out[86]: {'ali': 65, 'erhan': 80, 'veli': 56}

In [87]: düzeltilmiş_notlar = {"erhan":90,
    ...:                       "mert":77,
    ...:                       "efe":90
    ...:         }

In [88]: notlar.update(düzeltilmiş_notlar)

In [89]: notlar
Out[89]: {'ali': 65, 'efe': 90, 'erhan': 90, 'mert': 77, 'veli': 56}
```

Python'da String Metodları

Aslında biz string (karakter dizisi) kavramının ne olduğunu geçtiğimiz haftalardan biliyoruz. Çok kaba bir şekilde ifade etmek gerekirse, karakter dizileri, adından da anlaşılacağı gibi, karakterlerin bir araya gelmesiyle oluşan bir dizidir. Karakter dizileri; tek, çift veya üç tırnak içinde gösterilen, öteki veri tiplerinden de bu tırnaklar aracılığıyla ayırt edilen özel bir veri tipidir. Teknik olarak ifade etmek gerekirse, bir nesneyi type() fonksiyonu yardımıyla sorguladığımızda, eğer <class 'str'> çıktısı alıyorsak bu nesne bir karakter dizisidir.

1.Kullanımlarına Göre String Metotları

1.1. Büyük-Küçük Harf Değiştirme

Python'da karakter dizilerinin büyük-küçük harf durumlarını değiştirmek için kullanabileceğimiz beş farklı metot var. Bu metotlar şöyle sıralanabilir:

capitalize() metodu

Karakter dizilerinin ilk harfini büyütür.

```
In [15]:
    ...: print("samsun".capitalize())
Samsun

In [16]:
    ...: şehir = 'samsun'
    ...: şehir.capitalize()
Out[16]: 'Samsun'
```

title() metodu

Karakter dizisi içinde geçen her bir kelimenin ilk harfini büyütür.

```
In [27]: print ("pythonda string metotları".title())
Pythonda String Metotları

In [28]: metin = "pythonda string metotları"
    ...: print (metin.title())
Pythonda String Metotları
```

upper() metodu

Küçük harflerden oluşan bir karakter dizisi içindeki bütün harfleri büyütür.

```
In [31]:
    ...: print ("pythonda string metotları".upper())
PYTHONDA STRING METOTLARI

In [32]:
    ...: print (metin.upper())
PYTHONDA STRING METOTLARI
```

lower() metodu

Büyük harflerden oluşan bir karakter dizisi içindeki bütün harfleri küçültür.

```
In [39]:
    ...: print ("PYTHONDA STRING METOTLARI".lower())
pythonda string metotlari

In [40]:
    ...: metin = "PYTHONDA STRING METOTLARI"
    ...: print (metin.lower())
pythonda string metotlari
```

swapcase() metodu

Karakter dizisi içindeki büyük harfleri küçük, küçük harfleri büyük hale getirir.

```
In [41]:
    ...: print ("ŞakŞuKa".swapcase())
şAKşUkA

In [42]:
    ...: yemek = "ŞakŞuKa"
    ...: print (yemek.swapcase())
şAKşUkA
```

1.2. Büyük-Küçük Harf ve Boşluk Sorgulama

Python bize aynı zamanda karakter dizilerinin büyük-küçük harf durumlarını sorgulama imkanı da veriyor. Bu sorgulama işlemleri için de aşağıdakiler gibi birtakım metotlardan yararlanılır:

islower()

Karakter dizisinin tamamen küçük harflerden oluşup oluşmadığını sorgular.

```
    ...: print ("samsun".islower())
    ...:
    ...: print ("samsuN".islower())
True
False
```

isupper()

Karakter dizisinin tamamen büyük harflerden oluşup oluşmadığını sorgular.

```
    ...: print ("samsun".isupper())
    ...: print ("SAMSUN".isupper())
False
True
```

istitle()

Karakter dizisi içindeki kelimelerin ilk harflerinin büyük olup olmadığını sorgular.

```
In [53]: print ("pythonda string metotları".istitle())
False

In [54]: print ("Pythonda String Metotları".istitle())
True
```

isspace()

Karakter dizisinin tamamen boşluk karakterlerinden oluşup oluşmadığını sorgular.

```
    ...: print ("pythonda string metotları".isspace())
    ...: print ("  df".isspace())
    ...: print ("  ".isspace())
False
False
True
```

isnumeric()

Bu metot bir karakter dizisinin nümerik olup olmadığını denetler. Yani bu metot yardımıyla bir karakter dizisinin tamamının sayı değerli olup olmadığını denetleyebiliriz:

```
    ...: "12".isnumeric()
Out[59]: True

In [60]:
    ...: "dasd".isnumeric()
Out[60]: False

In [61]: "dsf232".isnumeric()
Out[61]: False
```

1.2. Stringlerde Karakter Değiştirme

replace()

Bu metot yardımıyla 2 ya da 3 parametre belirleyerek karakter değiştirebilir veya silebiliriz.

```
replace(old: str, new: str, count: int=...) -> str
```

Metodun parametrelerine bakınca soldan sağa sırasıyla değişecek değer, yeni değer ve kaç defa bu işlemin yapılacağını görebilmekteyiz. Aşağıdaki örneklerle bu durumları test edelim.

```
In [70]: kelime = "gazıantep" # bir string tanımlansın

In [71]: kelime.replace("ı","i") # ı harfini i ile yer
değiştirelim
Out[71]: 'gaziantep'

In [72]: kelime.replace("ı"," ") # boşluk ile yer değiştirme
Out[72]: 'gaz antep'

In [73]: kelime.replace("ı","") # hiçlik ile yer değiştirme
Out[73]: 'gazantep'

In [74]: kelime.replace("a","") # tüm a harflerini hiçlik ile
yer değiştirme
Out[74]: 'gzıntep'

In [75]: kelime.replace("a","",1) # 1 kere a harfini hiçlik
ile yer değiştirme
Out[75]: 'gzıantep'

In [76]: kelime
Out[76]: 'gazıantep'
```

1.3.String Verilerde Başlangıç ve Bitiş Değerleri Sorgulama

1.3.1 startswith()

String verinin belirli bir değer ile başlayıp başlamadığını sorgular. Bunun için aşağıdaki 3 parametre girilebilir. Bunlar soldan sağa doğru sorgulanacak karakter dizisi,hangi indisten başladığı ve nerede bittiğidir.

```
startswith(prefix: Union[Text, Tuple[Text, ...]], start:
           Optional[int]=..., end: Optional[int]=...) ->
           bool
```

```
In [97]: kelime.startswith("g") # kelime g ile başlıyor mu
Out[97]: True

In [98]: kelime.startswith("g",0) # g harfi kelimenin 0.
indeksinde mi
Out[98]: True

In [99]: kelime.startswith("g",0,5) # g harfi kelimenin 0.
indeksinden başlayıp 5.indekse kadar gidiyor mu?
Out[99]: True

In [100]: kelime.startswith("g",1,5) # g harfi kelimenin 1.
indeksinden başlayıp 5.indekse kadar gidiyor mu?
Out[100]: False

In [101]: kelime.startswith("a",1) #a harfi kelimenin 1.
indeksinden başlayıp 5.indekse kadar gidiyor mu?
Out[101]: True

In [102]: kelime.startswith("a",1,3) #a harfi kelimenin 1.
indeksinden başlayıp 3.indekse kadar gidiyor mu?
Out[102]: True

In [103]: kelime.startswith("azı",1,6) #azı karakter dizisi
kelimenin 1. indeksinden başlayıp 5.indekse kadar gidiyor mu?
Out[103]: True
```

1.3.2. endswith()

String verinin belirli bir değer ile bitip bitmediğini sorgular. Tıpki startwith metodunda olduğu gibi aşağıdaki 3 parametre girilebilir. Bunlar soldan sağa doğru sorgulanacak karakter dizisi,hangi indisten başladığı ve nerede bittiğidir.

```
endswith(suffix: Union[Text, Tuple[Text, ...]], start:
         Optional[int]=..., end: Optional[int]=...) -> bool
```

```
In [137]: kelime.endswith("p") # kelime p ile mi bitiyor?
Out[137]: True

In [138]: kelime.endswith("p",0) # p harfi kelimenin 0.
indeksinde mi
Out[138]: True

In [139]: kelime.endswith("p",0,5) # p harfi kelimenin 0.
indeksinden başlayıp 5.indekse kadar gidiyor mu?
Out[139]: False

In [140]: kelime.endswith("p",1,5) # g harfi kelimenin 1.
indeksinden başlayıp 5.indekse kadar gidiyor mu?
Out[140]: False

In [141]: kelime.endswith("antep",1,6) #azı karakter dizisi
kelimenin 1. indeksinden başlayıp 5.indekse kadar gidiyor mu?
Out[141]: False
```

1.4. String Yapılarda Karakter Konumu Bulma

1.4.1. find()/rfind() Metodu:

Bir karakterin String bir verinin içinde nerede yani verinin hangi konumunda (index) bulunduğunu bulmaya yarar. Aşağıdadan da görüleceği üzere bu metot için de taranacak karakter dizisi , başlangıç ve bitiş noktası girilir.

```
find(sub: Text, start: Optional[int]=..., end:
     Optional[int]=..., /) -> int
```

```
rfind(sub: Text, start: Optional[int]=..., end:
      Optional[int]=..., /) -> int
```

```
In [145]: text = "sakarya"

In [146]: print(text.find("a"))
1

In [147]: print(text.rfind("a"))
6
```

```
In [149]: print(text.find("a",1,6))
1

In [150]: print(text.find("a",1,1))
-1
```

1.4.2. join() Metodu:

Bir String verinin içerisindeki herbir karakterden sonra eklemek istediğimiz karakteri birleştimeye yarar.Parametre olarak iterasyona müsait bir yapı yazılır.

```
join(iterable: Iterable[str]) -> str
```

```
In [157]: #Formatı: "eklemek istenilen string ya da
karakter değer".join(değişkenin kendisi = elimizde olan
string veri)

In [158]: a = "Linux"

In [159]: print (".".join(a))
L.i.n.u.x
```

1.5. String Yapılarda Karakter Kırpma

1.5.1. strip() Metodu:

Bir String verinin başında (solunda) ve sonunda (sağında) yer alan boşluk ve yeni satır (\n) gibi karakterleri silemeye yarar.

```
In [172]: # strip / rstrip ve lstrip

In [173]: metin = "   soldan ve sağdan 3 sekme boşluk   "

In [174]: metin.strip() #boşluklar yok oldu
Out[174]: 'soldan ve sağdan 3 sekme boşluk'
```

1.5.2. rstrip() Metodu:

Bir String verinin sadece sonunda (sağında) yer alan boşluk ve yeni satır (\n) gibi karakterleri silemeye yarar.

```
In [181]: metin = "sağdan 3 sekme boşluk   "

In [182]: metin.rstrip() #sağdaki boşluklar yok olacak
Out[182]: 'sağdan 3 sekme boşluk'
```

1.5.3. lstrip() Metodu:

Bir String verinin sadece başında (solunda) yer alan boşluk ve yeni satır (\n) gibi karakterleri silemeye yarar.

```
In [177]: metin = "   soldan 3 sekme boşluk"

In [178]: metin.lstrip() #soldaki boşluklar yok olacak
Out[178]: 'soldan 3 sekme boşluk'
```

1.6. String Yapılarda Karakterleri Bölme

1.6.1 split() Metodu:

Bir String verinin içerisindeki karakterleri soldan sağa doğru okuyarak belirlediğimiz bir karakter kriterine String veriyi bölüp liste haline getirmeye yarar. Bu metodun 2 parametresi vardır. Birinci parametre ayıraç olarak kullanılacak nesne, ikinci parametre olarak da maksimum kaç defa ayırma işlemi yapılacağı girilir ve bu ayrılan nesneleri listenin içerisine atar.

```
split(sep: Optional[str]=..., maxsplit: int=...) ->
      List[str]
```

```
In [183]: ##### split

In [184]: ofis = "Word, Excel, PowerPoint, Access"

In [185]: print (ofis.split(","))
['Word', ' Excel', ' PowerPoint', ' Access']
```

```
In [191]: ofis.split(",",0)
Out[191]: ['Word, Excel, PowerPoint, Access']
```

```
In [196]: ofis.split(",",1)
Out[196]: ['Word', ' Excel, PowerPoint, Access']
```

```
In [197]: ofis.split(",",2)
Out[197]: ['Word', ' Excel', ' PowerPoint, Access']
```

Yukarıdaki örneklerde görüldüğü gibi virgül görüldüğü zaman çeşitli ayırmalar yapıldı ve ikinci parametre ile bu ayırma işleminin kaç defa tekrarlanılacağı belirtildi.

1.6.2 rsplit() Metodu:

Bir String verinin içerisindeki karakterleri sağdan sola doğru okuyarak belirlediğimiz bir karakter kriterine String veriyi bölüp liste haline getirmeye yarar.

```
In [203]: #rsplit

In [204]: ofis.rsplit(",") # metnin sağından başlayarak
dilimler.
Out[204]: ['Word', ' Excel', ' PowerPoint', ' Access']

In [205]: ofis.rsplit(",",1) # metnin sağıdan başayarak 1
kere dilimleme yapar.
Out[205]: ['Word, Excel, PowerPoint', ' Access']

In [206]: ofis.rsplit(",",2) # # metnin sağıdan başayarak 2
kere dilimleme yapar.
Out[206]: ['Word, Excel', ' PowerPoint', ' Access']
```

1.7. String Yapılarda Karakter Sayımı

1.7.1. count() Metodu:

Bir String verinin içerisindeki belirli bir karakterden kaç adet bulunduğunu belirlemeye yarar.

```
count(x: Text, start: Optional[int]=..., end:
      Optional[int]=..., /) -> int
```

Count metodu için girilebilecek 3 parametre vardır. Bu parametreler sırasıyla belitilen karakter dizisinin, hangi indeksten başlayarak durulacak indekse kadar sayılmasını sağlarlar.

```
In [210]: # count

In [211]: kelime
Out[211]: 'gazıantep'

In [212]: kelime.count("a")
Out[212]: 2
```

```
In [229]: text = """Ondokuz Mayıs University is a major
state university founded in 1975
     ...: in Samsun, Turkey. The university bears the name
“19 May”, which marks the date
     ...: when Mustafa Kemal Atatürk, founder of the
Republic of Turkey, came to Samsun
     ...: in order to start the War of Independence."""

In [230]: text.count("o")
Out[230]: 10

In [231]: text[0:4] # belli bir kısımdaki metne bakalım.
Out[231]: 'Ondo'

In [232]: text.count("o",0,4) # belli bir kısımdaki metinde
kaç tane o var bakalım
Out[232]: 1
```