Bilgisayar Programlama

Bilgisayar Programlama 1: Python ile Programlamaya Başlangıç

Açık Matematik

Açık Matematik — açık kaynaklı, reklamsız Türkçe matematik notları
acik-matematik.com
Bu çalışma Creative Commons BY-NC-SA 4.0 lisansıyla paylaşılmıştır: kaynak göstererek ve aynı lisansla, ticari olmayan amaçlarla kopyalayabilir, dağıtabilir ve uyarlayabilirsiniz.
Bu sürüm: 4 Kasım 2026

İçindekiler

Giriş 6

1 Hazırlık: Python ve Çalışma Ortamı 7
1.1 Program, Algoritma ve Programlama Dili 7
1.2 Python'u Çalıştırmanın Yolları 8
1.3 Defter, Kod Hücresi ve Metin Hücresi 8
1.4 Metin Hücreleri ve Markdown 9
1.5 İlk Program 10
1.6 Bu Notlardaki Kodları Okumak 11
1.7 Alıştırmalar 11

2 Değişkenler, Veri Tipleri ve Atama 14
2.1 Değişken ve Atama 14
2.2 Temel Veri Tipleri 14
2.3 Dinamik Tip Sistemi 15
2.4 Yorumlar, Çok Satırlı Metinler ve Noktalı Virgül 16
2.5 Tip Dönüşümleri 16
2.6 Atama Çeşitleri 18
2.7 Değişken Adlandırma Kuralları 19
2.8 Birleşik Atama Operatörleri 20
2.9 Alıştırmalar 21

3 Aritmetik İşlemler ve math Modülü 26
3.1 Aritmetik Operatörler 26
3.2 İşlem Önceliği 27
3.3 Ondalıklı Sayılardaki Yuvarlama Hatası 28
3.4 Hazır Matematik Fonksiyonları 29
3.5 math Modülü 30
3.6 Karmaşık Sayılarla İşlemler 34
3.7 Çeşitli Örnekler 35
3.8 Alıştırmalar 36

4 Girdi ve Çıktı İşlemleri 40
4.1 Kullanıcıdan Değer Almak 40
4.2 Hatalı Girdileri Yakalamak 42
4.3 Ekrana Yazdırmak: print 43
4.4 Biçimli Yazdırma 44
4.5 Kaçış Karakterleri 46

 CC BY-NC-SA 4.0

4.6 Hata Türleri . . . 47
4.7 İletişim Pencereleri . . . 49
4.8 Alıştırmalar . . . 49

5 Karşılaştırma, Mantık Operatörleri ve Koşullu İfadeler . . . 53
5.1 Karşılaştırma Operatörleri . . . 53
5.2 Mantık Operatörleri . . . 54
5.3 if Yapısı . . . 56
5.4 Koşullarda Mantık Operatörleri . . . 59
5.5 İç İçe Koşullar . . . 60
5.6 Tek Satırda Koşul . . . 60
5.7 pass İfadesi . . . 61
5.8 Çeşitli Örnekler . . . 61
5.9 Alıştırmalar . . . 63

6 Koleksiyonlar ve Listeler . . . 67
6.1 Dört Temel Koleksiyon . . . 67
6.2 Listelerde İndeksleme . . . 69
6.3 Dilimleme . . . 70
6.4 Eleman Ekleme ve Silme . . . 72
6.5 Diğer Liste İşlemleri . . . 73
6.6 Listelerle Kullanılan Hazır Fonksiyonlar . . . 75
6.7 range Fonksiyonu . . . 76
6.8 Alıştırmalar . . . 76

7 for Döngüsü ve Liste Üreteçleri . . . 81
7.1 for Döngüsünün Yapısı . . . 81
7.2 Döngüyle Toplam ve Çarpım . . . 83
7.3 break, continue ve pass . . . 85
7.4 enumerate ve zip . . . 87
7.5 Liste Üreteçleri . . . 88
7.6 Örnekler . . . 89
7.7 Alıştırmalar . . . 92

8 İç İçe Döngüler, while Döngüsü ve match-case . . . 97
8.1 İç İçe Döngüler . . . 97
8.2 while Döngüsü . . . 100
8.3 match-case Yapısı . . . 106
8.4 Çeşitli Örnekler . . . 107
8.5 Alıştırmalar . . . 110

9 Demetler . . . 114
9.1 Demet Oluşturmak . . . 114
9.2 Elemanlara Erişim ve Dilimleme . . . 115

 CC BY-NC-SA 4.0

9.3 Demetler Değiştirilemez . . . 116
9.4 Demetlerle İşlemler . . . 117
9.5 Üreteç İfadeleri . . . 118
9.6 Alıştırmalar . . . 119

10 Rastgele Sayılar: random Modülü . . . 123
10.1 Sözde Rastgele Sayılar . . . 123
10.2 Tohum Değeri . . . 123
10.3 Sayı Üretmek . . . 124
10.4 Rastgele Seçim ve Karıştırma . . . 125
10.5 Dağılımlardan Sayı Üretmek . . . 126
10.6 Örnekler . . . 127
10.7 Alıştırmalar . . . 129

11 Problemler: Tipler, İşlemler ve Koşullar . . . 133
11.1 Tipler ve Dönüşümler . . . 133
11.2 Yazdırma ve Biçimlendirme . . . 135
11.3 Atama ve Aritmetik . . . 137
11.4 Mantık ve Koşullar . . . 140
11.5 Döngülerle Kısa Sorular . . . 142

12 Problemler: Listeler, Döngüler ve Demetler . . . 146
12.1 Liste Dilimleme . . . 146
12.2 Liste Üreteçleri . . . 147
12.3 Döngüler ve Koşullar . . . 152
12.4 while Döngüleri . . . 155
12.5 Demetler . . . 158
12.6 Algoritmalar . . . 159
12.7 İç İçe Toplamlar . . . 163

 CC BY-NC-SA 4.0

Giriş

Bilgisayar Programlama, programlamayı hiç bilmeyen bir okura Python dilini baştan öğretir. İlk adım çalışma ortamını hazırlamak ve ilk programı çalıştırmaktır; ardından değişkenler, işlemler, girdi ve çıktı, karar yapıları, listeler, demetler ve döngüler sırayla gelir. Her kavram kısa kodlar üzerinden anlatılır ve her kodun altında, çalıştırıldığında ekrana yazdığı çıktı aynen verilir.

> **i** Not
>
> Notların ilk yarısı tamamlanmıştır; sonraki konular eklendikçe bu sayfadan bağlantılanacaktır. Python'ı matematiksel hesap için ileri düzeyde kullanmak isteyenler Python ile Bilimsel Hesaplama notlarına geçebilir.

 CC BY-NC-SA 4.0

1. Hazırlık: Python ve Çalışma Ortamı

Bilgisayar çok hızlı ama hiç düşünmeyen bir yardımcıdır: ne yapacağını ona adım adım ve hiç belirsizlik bırakmadan söylememiz gerekir. Programlama, bu adımları bilgisayarın anlayacağı bir dilde yazma işidir. Bu derste o dil **Python** olacak.
Kod yazmaya başlamadan önce iki şeyi hazırlamamız gerekiyor. Birincisi kavramsal: program, algoritma ve programlama dili ne demektir? İkincisi pratik: Python kodunu nereye yazıp nasıl çalıştıracağız? Bu bölümün sonunda tarayıcıda çalışan bir Python defteri açmış, ilk programımızı çalıştırmış ve bu notlardaki kodların nasıl okunacağını öğrenmiş olacağız.

1.1 Program, Algoritma ve Programlama Dili

Bilgisayara bir iş yaptırmanın ilk adımı, işin kendisini adımlara ayırmaktır; dil seçimi ondan sonra gelir.

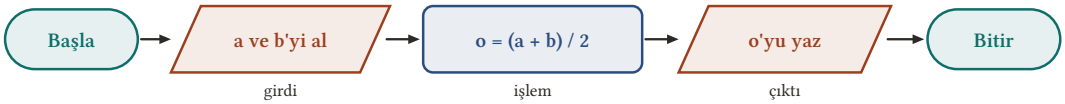
Tanım 1.1 (Algoritma). Bir problemi çözmek için izlenecek, sırası belli, her biri açık ve uygulanabilir adımlardan oluşan ve sonlu sayıda adımda biten talimatlar dizisine **algoritma** denir.

Yani bir algoritma, bir yemek tarifine benzer: malzemeler (girdiler) alınır, adımlar sırayla uygulanır ve sonunda bir sonuç (çıktı) elde edilir. Tarifteki "biraz tuz ekle" gibi belirsiz bir adım bir aşçı için yeterli olabilir, ama bir bilgisayar için değildir; algoritmanın her adımı tek bir anlama gelmelidir.

Örnek 1.1 (İki Sayının Ortalamasını Bulan Algoritma). İki sayının aritmetik ortalamasını bulan bir algoritma yazınız ve 8 ile 13 sayıları için uygulayınız.

Çözüm.
Algoritma dört adımdan oluşur:

1. Birinci sayıyı al ve ona a de.
2. İkinci sayıyı al ve ona b de.
3. a ile b'yi topla ve toplamı 2'ye böl; sonuca o de.
4. o'yu yaz ve dur.



Şekil 1.1: Ortalama algoritmasının akış şeması. Oval kutular başlangıcı ve bitişi, eğik kutular girdi ve çıktıyı, dikdörtgen kutu bir hesabı gösterir; oklar adımların sırasını verir.

$a = 8$ ve $b = 13$ için üçüncü adımda $o = (8 + 13)/2 = 21/2 = 10{,}5$ bulunur ve dördüncü adımda 10,5 yazılır. Adımlar sonlu sayıdadır, her biri tek anlamlıdır ve algoritma her iki sayı için aynı biçimde çalışır. ■

Algoritmayı bulmak işin düşünme kısmıdır. Onu bilgisayara anlatmak için bir dile ihtiyacımız var.

Tanım 1.2 (Program ve Programlama Dili). Bir algoritmanın, bilgisayarın çalıştırabileceği biçimde yazılmış hâline **program** denir. Programların yazıldığı, kelimeleri ve yazım kuralları kesin olarak belirlenmiş yapay dile **programlama dili** denir. Bir programı oluşturan metne **kaynak kod** ya da kısaca **kod** denir.

 CC BY-NC-SA 4.0

Yani bir programlama dili, insanın okuyup yazabileceği kadar anlaşılır, bilgisayarın da tek bir biçimde yorumlayabileceği kadar kesin bir dildir. Günlük dilde anlamı bağlamdan çıkarırız; programlama dilinde ise bir virgülün ya da bir parantezin eksik olması bile programın çalışmamasına yol açar.
Python, 1991'de ilk sürümü yayımlanan, okunaklı yazımıyla bilinen genel amaçlı bir programlama dilidir. Bu notlarda dilin 3. sürümü kullanılır; kodlar Python 3.12 ile çalıştırılmıştır.

Tanım 1.3 (Yorumlayıcı). Python kodunu satır satır okuyup çalıştıran programa **Python yorumlayıcısı** (interpreter) denir. Kodu bu şekilde, ayrı bir derleme adımı olmadan çalıştırılan dillere **yorumlanan diller** denir.

Yani Python'da bir kod yazdığımızda onu ayrıca makine diline çevirmekle uğraşmayız; yorumlayıcı kodu okur ve hemen çalıştırır. Bir hata varsa yorumlayıcı hatanın bulunduğu satırda durur ve bir hata mesajı yazar.

1.2 Python'u Çalıştırmanın Yolları

Python kodu birkaç farklı ortamda yazılıp çalıştırılabilir; dil hepsinde aynıdır, değişen yalnızca kodu yazdığımız pencere ve çıktıyı görme biçimidir.
Bu derste en çok kullanacağımız üç ortam şunlardır:

Ortam	Kurulum	Kod nereye yazılır	En uygun olduğu iş
Google Colab	gerekmez, tarayıcıda açılır	defterin kod hücrelerine	derste ve evde hızlı deneme
IDLE	python.org'dan Python ile gelir	etkileşimli kabuğa ya da `.py` dosyasına	kısa denemeler, internetsiz çalışma
VS Code	ayrıca kurulur, Python eklentisi eklenir	`.py` dosyasına ya da `.ipynb` defterine	uzun programlar ve projeler

Google Colab. Bir Google hesabıyla `colab.research.google.com` adresine girilir ve **Yeni not defteri** seçilir. Python Google'ın sunucularında çalışır; bilgisayara hiçbir şey kurmak gerekmez. Defterler Google Drive'a kaydedilir. Bu notlardaki bütün kodlar Colab'da olduğu gibi çalışır.
Python ve IDLE. `python.org` adresinden işletim sistemine uygun kurulum dosyası indirilir. Windows'ta kurulumun ilk ekranında **Add python.exe to PATH** kutusunu işaretlemek, Python'ın komut satırından da çalışmasını sağlar. Kurulumla birlikte **IDLE** adlı basit bir düzenleyici gelir. IDLE açıldığında `>>>` istemiyle beklenen bir **etkileşimli kabuk** (shell) görünür: buraya yazılan her satır Enter'a basıldığı anda çalışır. Uzun bir program için **File → New File** ile yeni bir dosya açılır, kod yazılır, `.py` uzantısıyla kaydedilir ve **Run → Run Module** (F5) ile çalıştırılır.
VS Code. Visual Studio Code, birçok dil için kullanılan ücretsiz bir kod düzenleyicisidir. Python ile kullanmak için önce Python kurulur, sonra VS Code'un eklentiler bölümünden **Python** eklentisi eklenir. `.py` uzantılı bir dosya açıldığında sağ üstteki çalıştır düğmesi programı çalıştırır ve çıktı alttaki terminal penceresinde görünür. **Jupyter** eklentisi eklenirse Colab'dakine benzer defterler (`.ipynb` dosyaları) VS Code içinde de açılabilir.

> **Hangisiyle başlamalı?**
>
> Hiç kurulum yapmadan hemen başlamak için Colab yeterlidir; bu bölümün geri kalanı da Colab üzerinden anlatılıyor. Kendi bilgisayarında çalışmak isteyen, Python'ı kurup IDLE ya da VS Code ile aynı kodları çalıştırabilir.

1.3 Defter, Kod Hücresi ve Metin Hücresi

Colab'da ve Jupyter'de çalışma, sayfalara bölünmüş bir defter üzerinde yapılır; bu yapıyı tanımak, kodun hangi sırayla çalıştığını anlamanın da anahtarıdır.

Tanım 1.4 (Defter). Kodun, açıklama metinlerinin ve kodun ürettiği çıktıların bir arada durduğu belgeye **defter** (notebook) denir. Jupyter ve Colab defterleri `.ipynb` uzantılı dosyalarda saklanır.

 CC BY-NC-SA 4.0

Yani bir defter, içinde çalışan kodlar bulunan bir ders notu gibidir. Bu derste kullanılan notlar da başlangıçta böyle defterler olarak hazırlanabilir.

Tanım 1.5 (Kod Hücresi ve Metin Hücresi). Bir defter alt alta dizilmiş **hücrelerden** (cell) oluşur. İçine Python kodu yazılan ve çalıştırıldığında çıktısı hemen altında görünen hücreye **kod hücresi**; içine açıklama, başlık ya da formül yazılan ve çalıştırıldığında biçimlendirilmiş metne dönüşen hücreye **metin hücresi** denir.

Yani kod hücreleri bilgisayar için, metin hücreleri okur için yazılır. Colab'da yeni bir kod hücresi **+ Kod**, yeni bir metin hücresi **+ Metin** düğmesiyle eklenir.
Bir kod hücresini çalıştırmanın üç yolu vardır:

- hücrenin solundaki çalıştır düğmesine tıklamak,
- **Ctrl + Enter**: hücreyi çalıştırır, imleç aynı hücrede kalır,
- **Shift + Enter**: hücreyi çalıştırır ve bir sonraki hücreye geçer.

Bir hücrede tanımlanan bir değişken, o hücre çalıştırıldıktan sonra defterin bütün hücrelerinde kullanılabilir. Önemli olan hücrelerin sayfadaki yeri değil, **çalıştırılma sırasıdır**. Colab her hücrenin solunda, o hücrenin oturumda kaçıncı sırada çalıştırıldığını gösteren küçük bir numara tutar.

Örnek 1.2 (Hücrelerin Çalıştırılma Sırası). Bir defterde alt alta iki kod hücresi olsun. Birinci hücrede `x = 5`, ikinci hücrede `print(x * 2)` yazıyor. Önce yalnız ikinci hücre çalıştırılırsa ne olur? Sonra birinci, ardından yine ikinci hücre çalıştırılırsa ne olur?

> **Çözüm.**
> Oturum yeni açıldıysa ve birinci hücre hiç çalıştırılmadıysa Python `x` adında bir şey tanımaz. İkinci hücre bu yüzden şu hata ile durur:
>
> ```
> NameError: name 'x' is not defined
> ```
>
> Birinci hücre çalıştırıldığında `x` değişkeni oluşur ve değeri 5 olur. Bundan sonra ikinci hücre çalıştırılırsa `x * 2` hesaplanır ve ekrana `10` yazılır. Hücreler sayfada aynı yerde dursa da sonucu çalıştırılma sırası belirlemiştir. ■

Hücreler karışık sırayla çalıştırıldıkça defterin durumu sayfada görünenden farklılaşabilir. Böyle durumlarda **Çalışma zamanı** (Runtime) menüsündeki **Oturumu yeniden başlat** (Restart session) komutu bütün değişkenleri siler; ardından hücreler yukarıdan aşağıya sırayla yeniden çalıştırılır. Defter **Dosya** menüsünden `.ipynb` ya da `.py` olarak indirilebilir ve yazdırılabilir.

1.4 Metin Hücreleri ve Markdown

Metin hücreleri **Markdown** adlı sade bir biçimlendirme diliyle yazılır; birkaç işaret öğrenmek, düzenli ve okunaklı defterler hazırlamaya yeter.

Yazılan	Görünen
`# Başlık`, `## Alt başlık`	büyük ve daha küçük başlık
`**kalın**`, `*eğik*`	**kalın**, *eğik*
`- madde`	madde işaretli liste
`` `print` ``	satır içinde kod: `print`
`$x^2 + 1$`	formül: $x^2 + 1$

Bir metin hücresinde, çalıştırılmayacak ama okura gösterilmek istenen birkaç satırlık bir kod parçası yazmak için kod, üçlü ters tırnak (```) satırları arasına alınır:

````
```
for eleman in veri_yapisi:
````

 CC BY-NC-SA 4.0
````

```
    eleman üzerinde yapılacak işlemler
```
```

Bu satırlar metin hücresinde eşit aralıklı bir yazı tipiyle, kutu içinde görünür ama çalıştırılmaz. Böyle bir gösterim, bir yapının genel kalıbını anlatmak için kullanışlıdır.

## 1.5 İlk Program

Artık ilk programımızı yazabiliriz; geleneğe uyarak ekrana bir selam yazdıracağız.

```
print("Merhaba, Dünya!")
```

Çıktı:

```
Merhaba, Dünya!
```

`print` ekrana bir şey yazdıran hazır bir **fonksiyondur**; yazılacak şey ondan sonra gelen parantezin içine konur. Tırnak içindeki `"Merhaba, Dünya!"` bir **metindir**: Python onu olduğu gibi ekrana yazar. Bir programda birden çok satır varsa satırlar yukarıdan aşağıya sırayla çalışır.

```
print("Açık Matematik")
print("Bilgisayar Programlama")
print(2 + 3)
print("2 + 3")
```

Çıktı:

```
Açık Matematik
Bilgisayar Programlama
5
2 + 3
```

Son iki satırın farkına dikkat edin: `2 + 3` tırnaksız yazıldığında Python toplamı hesaplar ve `5` yazar; tırnak içinde yazıldığında ise bunu bir metin olarak görür ve olduğu gibi yazar.

Tanım 1.6 (Yorum Satırı). Bir satırda `#` işaretinden satır sonuna kadar olan kısma **yorum** (comment) denir. Python yorumları okumaz ve çalıştırmaz.

Yani yorumlar bilgisayar için değil, kodu okuyan insan için yazılır: kodun ne yaptığını açıklamak, bir hesabın sonucunu not etmek ya da bir satırı geçici olarak devre dışı bırakmak için kullanılır.

```
# Bu program bir dairenin alanını yaklaşık olarak hesaplar.
r = 2                 # yarıçap
alan = 3.14 * r * r   # pi yerine 3.14 aldık
print(alan)
# print("Bu satır çalışmaz, çünkü başında # var.")
```

Çıktı:

```
12.56
```

Programda `r` ve `alan` adlı iki **değişken** kullandık: `=` işareti sağdaki değeri soldaki ada bağlar. Değişkenleri bir sonraki bölümde ayrıntılı olarak göreceğiz. Python'da ondalık ayırıcı virgül değil nokta olduğundan 3,14 sayısı `3.14` diye yazılır.
Birkaç satırı aynı anda yorum hâline getirmek için satırlar seçilip bir kısayola basılır. Colab'da **Ctrl + /** seçili satırların başına `#` koyar, ikinci basış kaldırır. VS Code'da da **Ctrl + /** çalışır; ayrıca **Ctrl + K** ardından **Ctrl + C** yorum ekler, **Ctrl + K** ardından **Ctrl + U** yorumu kaldırır.

1.5.1 Hata Mesajlarını Okumak

Kod yazarken hata yapmak işin doğal bir parçasıdır; önemli olan hata mesajını okuyup nedenini bulabilmektir.

 CC BY-NC-SA 4.0
```

```
print("Merhaba"
```

Çıktı:

```
SyntaxError: '(' was never closed
```

Python bu kodu çalıştırmaya hiç başlamadan durdu, çünkü açılan parantez kapanmamış. **SyntaxError** (yazım hatası), kodun dilin yazım kurallarına uymadığını söyler. Mesajın geri kalanı hatanın nedenini İngilizce olarak açıklar: "`(` hiç kapatılmadı". Colab ve IDLE, hatanın bulunduğu satırı da gösterir ve hatalı yerin altına `^` işareti koyar. Bu notlarda bir kod hata verdiğinde, hata mesajının yalnız son satırı gösterilecek; hatanın türünü ve nedenini bu satır söyler.

1.6 Bu Notlardaki Kodları Okumak

Notlarda kod, bir kutu içinde ve renklendirilmiş olarak verilir; hemen altında **Çıktı:** satırından sonra, kod çalıştırıldığında ekrana yazılan her şey aynen gösterilir. Kodlar kopyalanıp bir Colab hücresine yapıştırıldığında aynı çıktıyı verir.
İleride kullanıcıdan klavyeyle değer alan programlar yazacağız. Böyle programların çıktısında, kullanıcının yazdığı değer de sorunun hemen yanında görünür; Colab'da ve terminalde ekran tam olarak böyle görünür:

```
Adınızı girin: Zeynep
Merhaba, Zeynep
```

Burada `Adınızı girin:` yazısını program yazdı; `Zeynep` ise kullanıcının klavyeden girdiği değerdir.

1.7 Alıştırmalar

Aşağıdaki alıştırmaları bir Colab defterinde deneyerek çözün.

Alıştırma 1.1 (Üç Satırlık Tanıtım). Ekrana alt alta adınızı, bölümünüzü ve bu dersin adını yazdıran bir program yazınız.

> **Çözüm.**
> Her satır için ayrı bir `print` kullanırız; her `print` yazdığı metnin sonunda yeni satıra geçer. Adı ve bölümü örnek olarak Zeynep ve Matematik aldık:
>
> ```
> print("Zeynep")
> print("Matematik")
> print("Bilgisayar Programlama")
> ```
>
> Çıktı:
>
> ```
> Zeynep
> Matematik
> Bilgisayar Programlama
> ```
>
> Metinler tırnak içinde yazılmalıdır; tırnak unutulursa Python `Zeynep`'i bir değişken adı sanar ve `NameError` hatası verir. ■

Alıştırma 1.2 (Tırnaklı ve Tırnaksız Yazım). Aşağıdaki programın çıktısını çalıştırmadan bulunuz.

```
print("10 - 4")
print(10 - 4)
print("10" + "4")
```

> **Çözüm.**

 CC BY-NC-SA 4.0

Birinci satırda `10 - 4` tırnak içindedir; Python bunu bir metin olarak görür ve olduğu gibi yazar. İkinci satırda tırnak yoktur; Python çıkarmayı yapar ve `6` yazar. Üçüncü satırda iki metin `+` ile birleştirilir: `"10"` ile `"4"` yan yana eklenir ve `104` yazılır. Metinlerde `+` toplama değil birleştirme anlamına gelir; bunu ilerideki bölümlerde ayrıntılı göreceğiz.

```
print("10 - 4")
print(10 - 4)
print("10" + "4")
```

Çıktı:

```
10 - 4
6
104
```

■

Alıştırma 1.3 (Bir Yazım Hatasını Düzeltmek). Aşağıdaki program çalışmıyor. Hata mesajı nedir, hata nasıl düzeltilir?

```
print("Programlamaya hoş geldiniz!)
```

Çözüm.
Metin çift tırnakla açılmış ama kapatılmamış. Python satırın sonuna kadar kapanan tırnağı arar, bulamaz ve yazım hatası verir:

```
print("Programlamaya hoş geldiniz!)
```

Çıktı:

```
SyntaxError: unterminated string literal (detected at line 1)
```

Mesaj, "1. satırda bitirilmemiş bir metin" bulunduğunu söyler. Ünlem işaretinden sonra kapanan tırnağı ekleyince program çalışır:

```
print("Programlamaya hoş geldiniz!")
```

Çıktı:

```
Programlamaya hoş geldiniz!
```

■

Alıştırma 1.4 (Yorumla Devre Dışı Bırakmak). Aşağıdaki programın çıktısını bulunuz. Ardından programı, yalnız `B` ve `D` harflerini yazacak biçimde, hiçbir satırı silmeden değiştiriniz.

```
print("A")
print("B")
# print("C")
print("D")
```

Çözüm.

 CC BY-NC-SA 4.0

Üçüncü satır `#` ile başladığından yorumdur ve çalışmaz. Program `A`, `B` ve `D` harflerini alt alta yazar:

```
print("A")
print("B")
# print("C")
print("D")
```

Çıktı:

```
A
B
D
```

`A`'nın yazılmaması için birinci satırın başına da `#` koymak yeterlidir; satır silinmemiş olur ve gerektiğinde `#` kaldırılarak geri getirilebilir:

```
# print("A")
print("B")
# print("C")
print("D")
```

Çıktı:

```
B
D
```

■

Alıştırma 1.5 (Bir Metin Hücresi Yazmak). Bir metin hücresinde en üstte "Dairenin Alanı" başlığı, altında kalın yazılmış "Formül:" sözcüğü ve yanında $A = \pi r^2$ formülü görünecek biçimde Markdown kodunu yazınız.

Çözüm.
Başlık `#` ile, kalın yazı `**` işaretleri arasında, formül de iki dolar işareti arasında yazılır. Formülün içinde `\pi` yunan harfi π'yi, `r^2` de üssü verir:

```
# Dairenin Alanı

**Formül:** $A = \pi r^2$
```

Hücre çalıştırıldığında büyük bir "Dairenin Alanı" başlığı ve altında "**Formül:** $A = \pi r^2$" satırı görünür. Başlık ile sonraki satır arasındaki boş satır, ikisinin ayrı paragraflar olarak görünmesini sağlar. ■

Çalışma ortamımız hazır ve ilk programlarımızı çalıştırdık. Bir sonraki bölüm Değişkenler, Veri Tipleri ve Atama, programlarda verileri saklamak için kullanılan değişkenleri ve Python'daki temel veri tiplerini ele alıyor.

 CC BY-NC-SA 4.0

2. Değişkenler, Veri Tipleri ve Atama

Bir program çalışırken sayıları, metinleri ve ara sonuçları bir yerde tutmak zorundadır: bir öğrencinin adını, bir dairenin yarıçapını ya da bir toplamın o ana kadarki değerini. Bu değerleri adlandırıp saklamanın yolu **değişkenlerdir**.
Bu bölümde değişkenlerin nasıl oluşturulduğunu, Python'ın hangi temel veri tiplerini tanıdığını ve bir değeri bir tipten başka bir tipe nasıl çevirebileceğimizi göreceğiz. Bölümün sonunda değişkenlere değer vermenin kısa yollarını ve değişkenlere ad verirken uyulması gereken kuralları öğreneceğiz.

2.1 Değişken ve Atama

Bir değeri daha sonra kullanmak üzere saklamak için ona bir ad veririz; bu işlemin adı atamadır.

Tanım 2.1 (Değişken ve Atama). Bir değere verilen ada **değişken** (variable) denir. `ad = değer` biçimindeki komuta **atama** (assignment) denir: `=` işareti, sağ taraftaki değeri hesaplar ve sol taraftaki ada bağlar.

Yani `yas = 25` komutundan sonra `yas` adı 25 değerini temsil eder; programın geri kalanında `yas` yazdığımız her yerde Python 25'i kullanır. Atamada `=` işareti matematikteki eşitlik anlamına gelmez, "sağdakini soldakine ver" anlamına gelir. Bu yüzden `x = x + 1` matematikte çelişki olsa da Python'da anlamlı bir komuttur: `x`'in eski değerine 1 eklenir ve sonuç yeniden `x`'e verilir.

```
x = 10
print(x)
x = x + 1
print(x)
```

Çıktı:

```
10
11
```

Bir değişkene yeni bir değer atandığında eski değer unutulur; `print` her zaman değişkenin o anki değerini yazar.

2.2 Temel Veri Tipleri

Python her değerin bir **tipini** (type) bilir: bir tam sayı ile bir metin aynı biçimde saklanmaz ve aynı işlemlere girmez.

Tanım 2.2 (Tam Sayı: int). Ondalık kısmı olmayan sayılara **tam sayı** denir ve Python'da tipleri `int`'tir: `25`, `-7`, `0` gibi.

Yani `int` matematikteki $\mathbb{Z}$ kümesinin elemanlarını temsil eder. Python'da tam sayıların bir üst sınırı yoktur; bilgisayarın belleği yettiği sürece istediğimiz kadar büyük tam sayılarla hesap yapılabilir.

Tanım 2.3 (Ondalıklı Sayı: float). Ondalık kısmı olan sayılara **ondalıklı sayı** ya da **kayan noktalı sayı** (floating-point number) denir ve tipleri `float`'tur: `5.9`, `-0.25`, `3.0` gibi. Ondalık ayırıcı olarak nokta kullanılır.

Yani `3` bir `int`, `3.0` ise bir `float`'tur; ikisi aynı sayıyı gösterse de Python onları farklı tiplerde saklar.

Tanım 2.4 (Karmaşık Sayı: complex). Gerçel ve sanal kısmı olan sayılara **karmaşık sayı** denir ve tipleri `complex`'tir. Sanal birim i yerine `j` harfi kullanılır: $3 + 4i$ sayısı `3 + 4j` diye yazılır. Bir `z` karmaşık sayısının gerçel kısmı `z.real`, sanal kısmı `z.imag` ile alınır.

 CC BY-NC-SA 4.0

Yani karmaşık sayılarla Python'da doğrudan hesap yapılabilir; `j` harfi, önüne bir sayı yazıldığında sanal birim anlamına gelir.

```
z = 3 + 4j
print(z)
print(z.real)   # gerçel kısım
print(z.imag)   # sanal kısım
```

Çıktı:

```
(3+4j)
3.0
4.0
```

Gerçel ve sanal kısımlar, `z` tam sayılarla yazılmış olsa bile `float` olarak verilir.

Tanım 2.5 (Metin: str). Tek tırnak (`'...'`) ya da çift tırnak (`"..."`) arasına yazılan karakter dizilerine **metin** ya da **karakter dizisi** (string) denir ve tipleri `str`'dir.

Yani `"Ali"` ile `'Ali'` aynı metindir; tırnağın türü yalnız metnin nerede başlayıp nerede bittiğini gösterir. Metin içinde rakam bulunsa bile tırnak içindeki bir değer sayı değil metindir: `"25"` bir `str`, `25` bir `int`'tir.

Tanım 2.6 (Mantıksal Değer: bool). Yalnız `True` (doğru) ve `False` (yanlış) değerlerinden birini alan tipe **mantıksal tip** (Boolean) denir ve adı `bool`'dur.

Yani `bool` tipindeki bir değer bir sorunun yanıtı gibidir: "5, 3'ten küçük mü?" sorusunun yanıtı `5 < 3` ifadesinin değeridir ve bu değer `False`'tur. Mantıksal değerleri koşullu ifadelerde yoğun olarak kullanacağız. `True` ve `False` büyük harfle başlar; `true` yazmak hata verir.

2.2.1 type Fonksiyonu

Bir değerin ya da değişkenin tipini öğrenmek için `type` fonksiyonu kullanılır.

```
yas = 25
boy = 1.75
ad = "Ali"
gecti_mi = 5 < 3
print(type(yas))
print(type(boy))
print(type(ad))
print(type(gecti_mi))
print(type(2 / 3))
print(gecti_mi)
```

Çıktı:

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
<class 'float'>
False
```

`type`'ın yazdığı `<class 'int'>` ifadesi "bu değer `int` sınıfındandır" anlamına gelir. `2 / 3` bölümünün tipi `float`'tur: Python'da `/` ile bölmenin sonucu her zaman ondalıklı bir sayıdır.

2.3 Dinamik Tip Sistemi

Birçok programlama dilinde bir değişkenin tipi baştan yazılır ve sonradan değiştirilemez; Python'da durum farklıdır.

 CC BY-NC-SA 4.0

Tanım 2.7 (Dinamik Tip Sistemi). Bir değişkenin tipinin, ona atanan değere göre program çalışırken belirlendiği ve değişkene sonradan başka tipte bir değer atanabildiği dillere **dinamik tipli** diller denir. Python dinamik tipli bir dildir.

Yani Python'da `int x;` gibi bir tip bildirimi yoktur. Değişkenin tipi, o anda taşıdığı değerin tipidir; değer değişirse tip de değişebilir.

```
x = 10        # x şimdi bir int
print(type(x))
x = "Merhaba" # x artık bir str
print(x)
print(type(x))
```

Çıktı:

```
<class 'int'>
Merhaba
<class 'str'>
```

Bu esneklik kolaylık sağlar ama dikkat de ister: aynı adı farklı işler için kullanmak, programın ilerleyen bir yerinde değişkenin beklenmedik bir tipte olmasına yol açabilir.

2.4 Yorumlar, Çok Satırlı Metinler ve Noktalı Virgül

Kodun okunabilirliğini artıran ve birkaç satırı bir arada yazmayı sağlayan birkaç kural vardır.
Hazırlık bölümünde gördüğümüz gibi `#` işaretinden sonraki kısım bir yorumdur (Tanım 1.6). Yorum bir satırın başına ya da bir komutun yanına yazılabilir. Birden çok satıra yayılan bir metin ise üçlü tırnakla (`"""..."""` ya da `'''...'''`) yazılır; satır sonları metnin bir parçası olur.

```
mesaj = """Bu metin
üç satıra
yayılıyor."""
print(mesaj)
```

Çıktı:

```
Bu metin
üç satıra
yayılıyor.
```

Python'da her komut genellikle kendi satırına yazılır ve satır sonuna noktalı virgül konmaz. Birkaç kısa komut aynı satıra yazılmak istenirse aralarına noktalı virgül (`;`) konur:

```
x = 10; y = 20; z = 30
print(x, y, z)
x = x + 2; y = y + 3; z = z + 4
print(x, y, z)
```

Çıktı:

```
10 20 30
12 23 34
```

`print`'e virgülle ayrılmış birkaç değer verildiğinde, değerler aralarında birer boşlukla aynı satıra yazılır.

2.5 Tip Dönüşümleri

Bir değeri bir tipten başka bir tipe çevirmek için tipin adıyla aynı adı taşıyan fonksiyonlar kullanılır.

Tanım 2.8 (Tip Dönüşüm Fonksiyonları). `int(x)`, `float(x)`, `complex(x)`, `str(x)` ve `bool(x)` fonksiyonları `x` değerini sırasıyla tam sayıya, ondalıklı sayıya, karmaşık sayıya, metne ve mantıksal değere çevirir. Bu işleme **tip dönüşümü** (type conversion) denir.

 CC BY-NC-SA 4.0

Yani tip dönüşümü değerin kendisini değil, onun hangi biçimde saklandığını değiştirir. Dönüşümlerin her biri kendi kuralına göre çalışır.

Sayısal dönüşümler. `int` bir ondalıklı sayının ondalık kısmını atar; yuvarlama yapmaz. `float` bir tam sayıyı ondalıklı biçime çevirir. `complex(a)` sanal kısmı sıfır olan bir karmaşık sayı, `complex(a, b)` ise $a + bi$ sayısını verir.

```
print(int(5.9))       # ondalık kısım atılır
print(int(-5.9))      # sıfıra doğru kesilir
print(float(10))
print(complex(5))
print(complex(3, 4))
```

Çıktı:

```
5
-5
10.0
(5+0j)
(3+4j)
```

`int(-5.9)` sonucunun `-6` değil `-5` olduğuna dikkat edin: `int` sayıyı her zaman sıfıra doğru keser. Bir karmaşık sayı ise `int` ya da `float` ile gerçel sayıya çevrilemez:

```
z = complex(3, 4)
print(float(z))
```

Çıktı:

```
TypeError: float() argument must be a string or a real number, not 'complex'
```

Metin dönüşümleri. `str` bir değeri metne çevirir. Rakamlardan oluşan bir metin de `int` ya da `float` ile sayıya çevrilebilir. Metinler `+` ile birleştirilir; bir metinle bir sayı ise `+` ile toplanamaz.

```
sayi_metni = str(100)
print(sayi_metni, type(sayi_metni))
print(sayi_metni + "2")      # iki metin yan yana eklenir
print(int(sayi_metni) + 2)   # önce sayıya çevrilir, sonra toplanır
print(float("2.5") * 2)
```

Çıktı:

```
100 <class 'str'>
1002
102
5.0
```

```
sayi_metni = str(100)
print(sayi_metni + 2)
```

Çıktı:

```
TypeError: can only concatenate str (not "int") to str
```

Hata mesajı, bir metnin yalnız başka bir metinle birleştirilebileceğini söyler. Bu hata türünü ve diğerlerini Girdi ve Çıktı İşlemleri bölümünde topluca göreceğiz.

Mantıksal dönüşüm. `bool` bir değeri `True` ya da `False`'a çevirir. Kural basittir: sayılar için `0` ve `0.0`, metinler için boş metin `""` `False` olur; bunların dışındaki her değer `True` olur.

```
print(bool(0), bool(0.0), bool(""))
print(bool(5), bool(-2.3), bool("0"), bool("Ali"))
```

Çıktı:

```
False False False
True True True True
```

 CC BY-NC-SA 4.0

`bool("0")` sonucunun `True` olduğuna dikkat edin: `"0"` boş olmayan bir metindir; içinde sıfır rakamının yazılı olması sonucu değiştirmez.

Örnek 2.1 (Bir Ondalıklı Sayının Dönüşümleri). `a = 2/3` için `int(a)`, `float(a)`, `complex(a)` ve `bool(a)` değerlerini bulunuz.

Çözüm.
$2/3 = 0{,}666\ldots$ olduğundan `int(a)` ondalık kısmı atar ve 0 verir. `a` zaten `float` olduğu için `float(a)` değeri değiştirmez. `complex(a)` sanal kısmı sıfır olan karmaşık sayıyı verir. `a` sıfırdan farklı olduğu için `bool(a)` `True`'dur.

```
a = 2 / 3
print(type(a))
print(int(a))
print(float(a))
print(complex(a))
print(bool(a))
```

Çıktı:

```
<class 'float'>
0
0.6666666666666666
(0.6666666666666666+0j)
True
```

`float(a)` çıktısında, sonsuza dek sürmesi gereken 6'lar on altıncı basamakta kesilmiştir: bilgisayar $2/3$'ü sonsuz basamakla değil, sınırlı bir duyarlıkla saklar. Bunun nedenlerine Aritmetik İşlemler ve math Modülü bölümünde değineceğiz. ■

2.6 Atama Çeşitleri

Basit atamanın yanında Python, birden çok değişkene aynı anda değer vermenin kısa yollarını da sunar.

Tanım 2.9 (Çoklu Atama). `a = b = c = 5` biçimindeki atama aynı değeri birden çok değişkene verir. `x, y, z = 1, 2, 3` biçimindeki atama ise sağdaki değerleri sırayla soldaki değişkenlere verir; buna **çoklu atama** (multiple assignment) denir. Çoklu atamada iki taraftaki eleman sayıları eşit olmalıdır.

Yani `x, y, z = 1, 2, 3` komutu `x = 1`, `y = 2` ve `z = 3` komutlarını tek satırda yapar.

```
a = b = c = 5
print(a, b, c)
x, y, z = 1, 2, 3
print(x, y, z)
print(2 + x - y + z)
```

Çıktı:

```
5 5 5
1 2 3
4
```

Çoklu atamanın en kullanışlı uygulaması iki değişkenin değerlerini değiş tokuş etmektir.

Tanım 2.10 (Takas). İki değişkenin değerlerinin karşılıklı olarak yer değiştirmesine **takas** (swap) denir. Python'da `a, b = b, a` komutu `a` ve `b`'nin değerlerini takas eder.

Yani sağ taraftaki `b, a` önce eski değerlerle hesaplanır, sonra bu değerler sırayla `a`'ya ve `b`'ye verilir. Birçok dilde takas için üçüncü bir geçici değişken gerekir; Python'da gerekmez.

 CC BY-NC-SA 4.0

```
a = 5
b = 10
a, b = b, a
print("a:", a, "- b:", b)
```

Çıktı:

```
a: 10 - b: 5
```

Örnek 2.2 (Ardışık Atamayla Takas Denemesi). `a = 5` ve `b = 10` iken önce `a = b`, ardından `b = a` komutları çalıştırılırsa `a` ve `b`'nin son değerleri ne olur? Takas neden gerçekleşmez?

> **Çözüm.**
> İlk komut `a = b`, `a`'ya 10 değerini verir; `a`'nın eski değeri olan 5 bu anda kaybolur. İkinci komut `b = a`, `b`'ye `a`'nın yeni değeri olan 10'u verir. Sonuçta iki değişken de 10 olur:
>
> ```
> a = 5
> b = 10
> a = b
> b = a
> print(a, b)
> ```
>
> Çıktı:
>
> ```
> 10 10
> ```
>
> Takasın olması için ya üçüncü bir değişken kullanılmalı (`gecici = a`, `a = b`, `b = gecici`) ya da `a, b = b, a` yazılmalıdır. Çoklu atamada sağ taraf tamamen hesaplandıktan sonra atama yapıldığından hiçbir değer kaybolmaz. ■

Bir değişkenin değeri başka bir değişkene de atanabilir. `b = a` komutundan sonra `b`, `a`'nın o anki değerini alır; sonra `a` değişse bile `b` eski değerini korur:

```
a = 10
b = a
a = 20
print("a:", a, "- b:", b)
```

Çıktı:

```
a: 20 - b: 10
```

2.7 Değişken Adlandırma Kuralları

Değişkenlere istediğimiz adı veremeyiz; Python bazı kurallar koyar, bazı alışkanlıklar da kodun okunmasını kolaylaştırır.

Tanım 2.11 (Geçerli Değişken Adı). Python'da geçerli bir değişken adı şu kurallara uyar:

- harfle (`a-z`, `A-Z`) ya da alt çizgiyle (_) başlar, rakamla başlamaz;
- yalnız harf, rakam ve alt çizgi içerir; boşluk ve `@`, `#`, `$`, `%`, `-` gibi özel karakterler içermez;
- Python'ın **anahtar kelimelerinden** (keyword) biri değildir.

Yani `ogrenci_adi`, `sayi2`, `_toplam` geçerli adlardır; `2sayi`, `ogrenci adi`, `fiyat$` ve `for` geçerli değildir. Python büyük ve küçük harfi ayırt eder: `myVar` ile `myvar` iki farklı değişkendir.
Anahtar kelimeler, Python'ın kendi komutlarında kullandığı ve başka bir anlama verilemeyen sözcüklerdir. Güncel listeleri `keyword` modülünden alınabilir:

 CC BY-NC-SA 4.0

```
import keyword
print(keyword.kwlist)
print(keyword.iskeyword("lambda"))
print(keyword.iskeyword("x"))
```

Çıktı:

```
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', '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']
True
False
```

`import keyword` komutu `keyword` adlı hazır modülü programa dahil eder; modülleri bir sonraki bölümde `math` modülüyle birlikte göreceğiz. Bir anahtar kelimeyi değişken adı olarak kullanmak yazım hatası verir:

```
lambda = 3
```

Çıktı:

```
SyntaxError: invalid syntax
```

Kurallara ek olarak iyi bir adın uyduğu bazı alışkanlıklar vardır:

- Ad, değişkenin ne tuttuğunu anlatmalıdır: `a` yerine `ogrenci_adi`, `t` yerine `toplam`.
- Birden çok sözcük alt çizgiyle bağlanır: `dogum_yili`.
- Ad çok uzun olmamalıdır; Python'ın yazım kılavuzu bir kod satırının 79 karakteri geçmemesini önerir.
- Python `ç, ğ, ı, ö, ş, ü` gibi Türkçe harfleri adlarda kabul eder, ama başka sistemlerle uyum için bunları kullanmamak daha iyidir.
- `print`, `input`, `str`, `int`, `float`, `list`, `len`, `type` gibi hazır fonksiyonların adları anahtar kelime değildir ve değişken adı olarak kullanılabilir; ama kullanılırsa o fonksiyon programın geri kalanında çalışmaz.

Örnek 2.3 (Hazır Bir Fonksiyonun Adını Kullanmak). Aşağıdaki program neden hata verir?

```
print = 5
print("Merhaba")
```

Çözüm.
Birinci satır anahtar kelime olmayan `print` adına 5 değerini atar; bu bir yazım hatası değildir ve Python itiraz etmez. Ama bu andan sonra `print` adı ekrana yazan fonksiyonu değil, 5 sayısını gösterir. İkinci satırda 5 sayısı bir fonksiyon gibi çağrılmaya çalışılır:

```
print = 5
print("Merhaba")
```

Çıktı:

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

Hata mesajı "`int` nesnesi çağrılabilir değildir" der. Colab'da bu durumda oturumu yeniden başlatmak `print`'i eski hâline getirir. ■

2.8 Birleşik Atama Operatörleri

Bir değişkenin değerini bir işlemle güncellemek çok sık yapılan bir iştir; Python bunun için kısa bir yazım sunar.

Tanım 2.12 (Birleşik Atama Operatörü). `x += a` komutu `x = x + a` anlamına gelir; benzer biçimde `x -= a`, `x *= a`, `x /= a`, `x //= a`, `x %= a` ve `x **= a` komutları sırasıyla çıkarma, çarpma, bölme, tam

bölme, kalan ve üs alma işlemini yapıp sonucu yeniden `x`'e atar. Bunlara **birleşik atama operatörleri** (compound assignment operators) denir.

Yani `x += 3` "x'i 3 artır", `x *= 2` "x'i iki katına çıkar" demektir. `/=` işleminin sonucu, `/` ile bölmede olduğu gibi her zaman `float`'tur.

```
x = 5
x += 3      # x = 5 + 3
print("x:", x)
x -= 4      # x = 8 - 4
print("x:", x)
x *= 6      # x = 4 * 6
print("x:", x)
x /= 2      # x = 24 / 2
print("x:", x)
```

Çıktı:

```
x: 8
x: 4
x: 24
x: 12.0
```

Tam bölme `//`, kalan `%` ve üs alma `**` işlemlerini bir sonraki bölümde ayrıntılı olarak göreceğiz.

Örnek 2.4 (Birleşik Atamaları Adım Adım İzlemek). Aşağıdaki programın çıktısını bulunuz.

```
i, j = 8, 2
i /= j
j *= i
print(i)
print(j)
```

Çözüm.
Komutları sırayla izleyelim:

- `i, j = 8, 2`: $i = 8, j = 2$.
- `i /= j`: $i = 8/2 = 4{,}0$. Bölme sonucu `float` olduğundan `i` artık `4.0`'dır.
- `j *= i`: $j = 2 \cdot 4{,}0 = 8{,}0$. Bir `int` ile bir `float` çarpıldığında sonuç `float` olur.

```
i, j = 8, 2
i /= j
j *= i
print(i)
print(j)
```

Çıktı:

```
4.0
8.0
```

■

2.9 Alıştırmalar

Aşağıdaki alıştırmalarda önce sonucu kâğıt üzerinde bulun, sonra kodu çalıştırarak kontrol edin.

Alıştırma 2.1 (Değerlerin Tipleri). `7`, `7.0`, `"7"`, `7 > 3` ve `7 + 0j` değerlerinin tiplerini bulunuz.

 CC BY-NC-SA 4.0

Çözüm.
`7` ondalık kısmı olmayan bir sayıdır, tipi `int`'tir. `7.0` ondalık noktası taşıdığı için `float`'tur. `"7"` tırnak içinde olduğundan metindir, `str`'dir. `7 > 3` bir karşılaştırmadır; değeri `True`'dur ve tipi `bool`'dur. `7 + 0j` sanal kısmı sıfır olsa da karmaşık sayı olarak yazılmıştır, `complex`'tir.

```
print(type(7))
print(type(7.0))
print(type("7"))
print(type(7 > 3))
print(type(7 + 0j))
```

Çıktı:

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
<class 'complex'>
```

■

Alıştırma 2.2 (int ile Kesmek). `int(7.99)`, `int(-7.99)` ve `int("7")` değerlerini bulunuz. `int("7.5")` neden hata verir?

Çözüm.
`int` ondalıklı bir sayının ondalık kısmını atar ve sayıyı sıfıra doğru keser: `int(7.99)` 7, `int(-7.99)` -7 olur. `int("7")` rakamdan oluşan metni tam sayıya çevirir ve 7 verir.

```
print(int(7.99))
print(int(-7.99))
print(int("7"))
```

Çıktı:

```
7
-7
7
```

`"7.5"` metni bir tam sayıyı değil, ondalıklı bir sayıyı gösterir; `int` metinden doğrudan yalnız tam sayı yazımlarını okuyabilir:

```
print(int("7.5"))
```

Çıktı:

```
ValueError: invalid literal for int() with base 10: '7.5'
```

Böyle bir metni tam sayıya çevirmek için önce `float`, sonra `int` uygulanır: `int(float("7.5"))` 7 verir.
■

Alıştırma 2.3 (Hangileri True?). `bool(1)`, `bool(-1)`, `bool(0.0)`, `bool(" ")` ve `bool("")` değerlerini bulunuz.

Çözüm.

 CC BY-NC-SA 4.0

Sayılarda yalnız sıfır `False` verir: `bool(1)` ve `bool(-1)` `True`, `bool(0.0)` `False`'tur. Metinlerde yalnız boş metin `False` verir. `" "` bir boşluk karakteri içerdiği için boş değildir ve `True` verir; `""` ise boştur ve `False` verir.

```
print(bool(1), bool(-1), bool(0.0))
print(bool(" "), bool(""))
```

Çıktı:

```
True True False
True False
```

■

Alıştırma 2.4 (Üç Değişkeni Döndürmek). `a, b, c = 1, 2, 3` olsun. Tek bir atamayla `a`'ya `b`'nin, `b`'ye `c`'nin, `c`'ye de `a`'nın değerini veriniz; sonuç `2 3 1` olmalıdır.

Çözüm.
Takastaki gibi çoklu atama kullanırız. Sağ taraf önce eski değerlerle hesaplanır, sonra sırayla sol taraftaki değişkenlere verilir. `a`'ya `b`, `b`'ye `c`, `c`'ye `a` gideceği için sağ taraf `b, c, a` olmalıdır:

```
a, b, c = 1, 2, 3
a, b, c = b, c, a
print(a, b, c)
```

Çıktı:

```
2 3 1
```

Bunu ardışık atamalarla (`a = b`, `b = c`, `c = a`) yapmaya çalışsaydık `c = a` adımında `a` zaten değişmiş olurdu ve `c` de 2 olurdu. ■

Alıştırma 2.5 (Geçerli Değişken Adları). `toplam_1`, `1toplam`, `_sayac`, `ders adi`, `class`, `Class`, `not%` adlarından hangileri geçerli değişken adıdır?

Çözüm.
Tanım 2.11 maddelerini tek tek uygulayalım:

- `toplam_1`: harfle başlıyor, yalnız harf, rakam ve alt çizgi içeriyor. Geçerli.
- `1toplam`: rakamla başlıyor. Geçersiz.
- `_sayac`: alt çizgiyle başlıyor. Geçerli.
- `ders adi`: boşluk içeriyor. Geçersiz.
- `class`: bir anahtar kelime. Geçersiz.
- `Class`: Python büyük harfi ayırt ettiği için bu ad `class` ile aynı değildir ve anahtar kelime değildir. Geçerli.
- `not%`: `%` özel karakterini içeriyor. Geçersiz.

`keyword.iskeyword` ile son iki kelime denetimini doğrulayabiliriz:

```
import keyword
print(keyword.iskeyword("class"), keyword.iskeyword("Class"))
```

Çıktı:

```
True False
```

■

Alıştırma 2.6 (Birleşik Atama Zinciri). Aşağıdaki programın çıktısını bulunuz.

```
x = 3
x **= 2
x -= 1
x //= 3
x *= x
print(x)
```

Çözüm.
Her satırdan sonra `x`'in değerini yazalım:

- `x = 3`: $x = 3$.
- `x **= 2`: $x = 3^2 = 9$.
- `x -= 1`: $x = 9 - 1 = 8$.
- `x //= 3`: 8'in 3'e bölümünün tam kısmı alınır, $x = 2$.
- `x *= x`: $x = 2 \cdot 2 = 4$.

```
x = 3
x **= 2
x -= 1
x //= 3
x *= x
print(x)
```

Çıktı:

```
4
```

Hiçbir adımda `/` kullanılmadığı için `x` baştan sona `int` olarak kalır. ■

Alıştırma 2.7 (Metin mi, Sayı mı?). `a = "4"` ve `b = 4` iken `a * 3`, `b * 3`, `a + str(b)` ve `int(a) + b` değerlerini bulunuz.

Çözüm.
Bir metin bir tam sayıyla çarpıldığında metin o kadar kez yan yana yazılır: `a * 3` `"444"` olur. `b * 3` sayıların çarpımıdır, 12'dir. `a + str(b)` iki metni birleştirir ve `"44"` verir. `int(a) + b` önce `a`'yı sayıya çevirir, sonra toplar ve 8 verir.

```
a = "4"
b = 4
print(a * 3)
print(b * 3)
print(a + str(b))
print(int(a) + b)
```

Çıktı:

```
444
12
44
8
```

■

Bu bölümde verileri değişkenlerde saklamayı, Python'ın temel veri tiplerini ve tipler arasında dönüşümü gördük. Bir sonraki bölüm Aritmetik İşlemler ve math Modülü, bu değerlerle yapılan hesapları, işlem önceliğini ve hazır matematik fonksiyonlarını ele alıyor.

 CC BY-NC-SA 4.0

3. Aritmetik İşlemler ve math Modülü

Bilgisayarın ilk ve en temel işi hesap yapmaktır. Python'ı açıp `2 + 3` yazdığımızda onu bir hesap makinesi gibi kullanmış oluruz; ama bu hesap makinesinin kendine özgü kuralları vardır. Bölmenin iki farklı türü vardır, işlemler belli bir öncelik sırasıyla yapılır ve ondalıklı sayılarla yapılan hesaplar her zaman tam olarak beklediğimiz sonucu vermez.
Bu bölümde önce Python'ın aritmetik operatörlerini ve işlem önceliğini göreceğiz. Ardından hazır gelen matematik fonksiyonlarını ve karekökten logaritmaya, faktöriyelden en büyük ortak bölene kadar pek çok fonksiyon içeren `math` modülünü tanıyacağız.

3.1 Aritmetik Operatörler

Python'da dört işlemin yanında tam bölme, kalan ve üs alma için de birer operatör vardır.

İşlem	Operatör	Örnek	Sonuç
toplama	`+`	`5 + 3`	`8`
çıkarma	`-`	`10 - 4`	`6`
çarpma	`*`	`7 * 3`	`21`
bölme	`/`	`15 / 2`	`7.5`
tam bölme	`//`	`15 // 2`	`7`
kalan (mod)	`%`	`15 % 4`	`3`
üs alma	`**`	`2 ** 3`	`8`

```
print(5 + 3, 10 - 4, 7 * 3)
print(15 / 2)
print(15 // 2)
print(15 % 4)
print(2 ** 3)
print(10 / 2)   # bölüm tam sayı olsa da sonuç float'tur
```

Çıktı:

```
8 6 21
7.5
7
3
8
5.0
```

`/` operatörü bölüm tam sayı olsa bile her zaman bir `float` verir; `10 / 2` sonucu `5` değil `5.0`'dır. Tam sayı sonucu isteniyorsa tam bölme kullanılır.

Tanım 3.1 (Tam Bölme ve Kalan). a ve $b \neq 0$ sayıları için a'nın b'ye bölümünün aşağı yuvarlanmış değerine **tam bölüm** denir ve Python'da `a // b` ile hesaplanır. **Kalan** `a % b` ise

$$a = (a \,//\, b) \cdot b + (a \,\%\, b)$$

eşitliğini sağlayan sayıdır.

Yani `15 // 4` işlemi $15/4 = 3{,}75$ sayısını aşağı yuvarlar ve 3 verir; kalan da $15 - 3 \cdot 4 = 3$ olur. Pozitif sayılarda bu, ilkokulda öğrendiğimiz bölme işleminin bölümü ve kalanıdır. Bir sayının çift olup olmadığını

 CC BY-NC-SA 4.0

anlamak için `n % 2` değerine, son basamağını bulmak için `n % 10` değerine bakmak gibi pek çok kullanımı vardır.
Negatif sayılarda "aşağı yuvarlama" sözü önemlidir: $-7/2 = -3{,}5$ sayısı aşağı yuvarlanınca -3 değil -4 olur. Kalan da tanımdaki eşitlikten bulunur ve Python'da kalan her zaman bölenle aynı işaretlidir.

```
print(-7 // 2, -7 % 2)    # -7 = (-4) * 2 + 1
print(-8 // 3, -8 % 3)    # -8 = (-3) * 3 + 1
print(7 // -2, 7 % -2)    # 7 = (-4) * (-2) + (-1)
print(24 // 2.5)          # float ile tam bölmenin sonucu float'tur
```

Çıktı:

```
-4 1
-3 1
-4 -1
9.0
```

Son satırda $24/2{,}5 = 9{,}6$ aşağı yuvarlanır ve `9.0` bulunur; işlemdeki sayılardan biri `float` olduğu için sonuç da `float`'tur.

> **i** Son sonucu gösteren alt çizgi
>
> Etkileşimli kabukta (IDLE ya da terminal) ve Colab'da, en son hesaplanıp ekranda gösterilen değer `_` adıyla tekrar kullanılabilir: `15 / 2` yazdıktan sonra `_ * 2` yazmak `15.0` verir. Bu özellik yalnız etkileşimli çalışmada geçerlidir; bir `.py` dosyasında `_` böyle bir anlam taşımaz, bu yüzden programlarda kullanılmamalıdır.

3.2 İşlem Önceliği

Birden çok operatör içeren bir ifadede Python, matematikteki sıraya benzer bir öncelik sırası izler.

Tanım 3.2 (İşlem Önceliği). Python bir ifadeyi şu sırayla hesaplar:

1. parantez içi `( )`,
2. üs alma `**`,
3. çarpma, bölme, tam bölme ve kalan `*`, `/`, `//`, `%`,
4. toplama ve çıkarma `+`, `-`.

Aynı öncelikteki işlemler soldan sağa yapılır; yalnız üs alma sağdan sola yapılır.

Yani `3 + 5 * 2` ifadesinde önce çarpma yapılır ve sonuç 13 olur; toplamanın önce yapılması isteniyorsa `(3 + 5) * 2` yazılır ve sonuç 16 olur. Kâğıt üzerinde kesir çizgisinin yaptığı gruplama işini Python'da parantezler yapar.

```
print(3 + 5 * 2)
print((3 + 5) * 2)
print(2 ** 3 ** 2)      # 2 ** (3 ** 2) = 2 ** 9
print((2 ** 3) ** 2)    # 8 ** 2
print(2 ** 2 * 2, 2 * 2 ** 2)
print(-2 ** 2)          # -(2 ** 2)
```

Çıktı:

```
13
16
512
64
8 8
-4
```

`2 ** 3 ** 2` ifadesi $2^{3^2} = 2^9 = 512$ olarak hesaplanır, çünkü üs alma sağdan sola yapılır. `-2 ** 2` ifadesinde de üs alma eksi işaretinden önce gelir: sonuç $(-2)^2 = 4$ değil, $-(2^2) = -4$'tür.

 CC BY-NC-SA 4.0

Örnek 3.1 (Önceliği Adım Adım Uygulamak). `5 - 6 / 2 * 3 - 1 + 7 * 6 / 3` ifadesinin değerini bulunuz.

Çözüm.
Önce çarpma ve bölmeler soldan sağa yapılır:

- `6 / 2 * 3`: önce $6/2 = 3{,}0$, sonra $3{,}0 \cdot 3 = 9{,}0$.
- `7 * 6 / 3`: önce $7 \cdot 6 = 42$, sonra $42/3 = 14{,}0$.

İfade $5 - 9{,}0 - 1 + 14{,}0$ olur. Toplama ve çıkarmalar soldan sağa yapılınca $5 - 9{,}0 = -4{,}0$, $-4{,}0 - 1 = -5{,}0$ ve $-5{,}0 + 14{,}0 = 9{,}0$ bulunur.

```
a = 5 - 6 / 2 * 3 - 1 + 7 * 6 / 3
print(a)
```

Çıktı:

```
9.0
```

Sonuç `9.0`'dır; ifadede `/` kullanıldığı için tam sayı değil ondalıklı sayı çıkar. ■

Örnek 3.2 (Kalan ve Üs İçeren Bir İfade). `Y = 4 * 8 % 4 + 2 ** (2 * 2 + 2)` komutundan sonra `Y`'nin değeri nedir?

Çözüm.
Önce parantez içi hesaplanır: $2 \cdot 2 + 2 = 6$. Sonra üs alınır: $2^6 = 64$. Ardından aynı öncelikteki `*` ve `%` soldan sağa yapılır: $4 \cdot 8 = 32$ ve 32'nin 4'e bölümünden kalan 0'dır. Son olarak toplama yapılır: $0 + 64 = 64$.

```
Y = 4 * 8 % 4 + 2 ** (2 * 2 + 2)
print(Y)
```

Çıktı:

```
64
```

■

3.3 Ondalıklı Sayılardaki Yuvarlama Hatası

Tam sayılarla yapılan hesaplar her zaman tamdır; ondalıklı sayılarda ise küçük ama önemli bir pürüz vardır.

```
a = 0.1 + 0.1
b = 0.1 + 0.2
print(a)
print(b)
print(b == 0.3)
```

Çıktı:

```
0.2
0.30000000000000004
False
```

$0{,}1 + 0{,}2$ toplamının $0{,}3$ değil $0{,}30000000000000004$ çıkmasının nedeni, bilgisayarın sayıları ikilik tabanda saklamasıdır. $1/3$ sayısının onluk tabanda $0{,}333\ldots$ diye sonsuza dek sürmesi gibi, $0{,}1$ sayısı da

 CC BY-NC-SA 4.0

ikilik tabanda sonsuza dek süren bir açılıma sahiptir. Bilgisayar bu açılımı belli bir basamakta keser; saklanan sayı 0,1'e çok yakın ama ona eşit olmayan bir sayıdır. Bu küçük farklar işlemler sırasında birikir ve bazen çıktıda görünür.

> ⚠ Ondalıklı sayıları == ile karşılaştırmayın
>
> Yukarıdaki örnekte `0.1 + 0.2 == 0.3` sorusunun yanıtı `False`'tur. İki ondalıklı sayının eşitliğini sınamak yerine farklarının çok küçük olup olmadığına bakılır: `abs(b - 0.3) < 1e-9` gibi. Burada `1e-9`, 10^{-9} sayısının Python'daki yazılışıdır.

3.4 Hazır Matematik Fonksiyonları

Python, hiçbir şey yüklemeden kullanılabilen birkaç hazır matematik fonksiyonuyla gelir.

Fonksiyon	Anlamı	Örnek	Sonuç
`min(a, b, ...)`	en küçük değer	`min(5, -23, 34)`	`-23`
`max(a, b, ...)`	en büyük değer	`max(5, -23, 34)`	`34`
`abs(x)`	mutlak değer	`abs(-4.78)`	`4.78`
`pow(x, y)`	x^y	`pow(2, 3)`	`8`
`round(x, n)`	`x`'i `n` ondalık basamağa yuvarlar	`round(5.1234, 2)`	`5.12`

```
print(min(5, -23, 34))
print(max(5, -23, 34))
print(abs(-4.78))
print(pow(2, 3), pow(2, -1))
```

Çıktı:

```
-23
34
4.78
8 0.5
```

`pow(2, 3)` işlemi `2 ** 3` ile aynıdır; tam sayılarla çağrıldığında sonuç da tam sayıdır. Üs negatifse sonuç ondalıklı olur: $2^{-1} = 0{,}5$.

Tanım 3.3 (round Fonksiyonu). `round(x, n)` fonksiyonu `x` sayısını virgülden sonra `n` basamağa yuvarlar; `n` verilmezse en yakın tam sayıya yuvarlar ve sonucu `int` olarak verir. Sayı iki tam sayının tam ortasındaysa (örneğin 2,5), `round` onu çift olan komşusuna yuvarlar.

Yani `round(5.7)` 6, `round(5.1234, 2)` 5,12 olur. Tam ortadaki sayılar için kullanılan "çift olana yuvarla" kuralı, çok sayıda yuvarlama yapıldığında hataların hep aynı yöne birikmesini önler; bu yüzden `round(2.5)` 2, `round(3.5)` ise 4'tür.

```
print(round(5.7))
print(round(5.1234, 2))
print(round(0.127885534, 4))
print(round(0.127985534, 4))
print(round(-0.127985534, 5))
print(round(2.5), round(3.5))
```

Çıktı:

```
6
5.12
0.1279
```

 CC BY-NC-SA 4.0

```
0.128
-0.12799
2 4
```

Dördüncü satırda 0,127985534 dört basamağa yuvarlanınca 0,1280 olur; Python sondaki gereksiz sıfırı yazmaz ve `0.128` gösterir.
Bir sayıyı tam sayıya indirmenin üç farklı yolu vardır ve negatif sayılarda farklı sonuç verirler: `int` sıfıra doğru keser, `round` en yakına yuvarlar, ileride göreceğimiz `math.floor` ise aşağı yuvarlar.

3.5 math Modülü

Karekök, trigonometrik fonksiyonlar ya da logaritma gibi daha ileri işlemler Python'ın `math` modülündedir.

Tanım 3.4 (Modül ve import). Hazır fonksiyonları ve sabitleri bir arada tutan Python dosyasına **modül** (module) denir. Bir modülü programda kullanabilmek için önce `import modul_adi` komutuyla programa dahil etmek gerekir; ardından modüldeki bir fonksiyon `modul_adi.fonksiyon_adi(...)` biçiminde çağrılır.

Yani `import math` komutundan sonra karekök fonksiyonu `math.sqrt` adıyla kullanılır. Bir modülün içinde neler bulunduğu `dir` fonksiyonuyla listelenebilir:

```
import math
print(dir(math))
```

Çıktı:

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh',
'asin', 'asinh', 'atan', 'atan2', 'atanh', 'cbrt', 'ceil', 'comb', 'copysign', 'cos',
'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'exp2', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf',
'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'lcm', 'ldexp', 'lgamma', 'log',
'log10', 'log1p', 'log2', 'modf', 'nan', 'nextafter', 'perm', 'pi', 'pow', 'prod',
'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'sumprod', 'tan', 'tanh', 'tau',
'trunc', 'ulp']
```

Listenin başındaki `__doc__` gibi alt çizgili adlar Python'ın iç işleyişi içindir; geri kalanlar kullanabileceğimiz fonksiyon ve sabitlerdir. Bir fonksiyonun ne yaptığını öğrenmek için `help(math.sqrt)` yazılabilir. Modüldeki adları başlarına `math.` yazmadan kullanmak için `from math import sqrt, pi` gibi bir komutla yalnız istenen adlar dahil edilebilir. `from math import *` komutu modüldeki bütün adları dahil eder; ama programdaki başka adlarla çakışabileceği için büyük programlarda önerilmez.

```
from math import sqrt, pi
print(sqrt(16))
print(pi)
```

Çıktı:

```
4.0
3.141592653589793
```

3.5.1 Sabitler

`math` modülü sık kullanılan matematik sabitlerini hazır olarak verir.

Sabit	Anlamı	Değeri
`math.pi`	π	`3.141592653589793`
`math.e`	Euler sayısı e	`2.718281828459045`
`math.tau`	$\tau = 2\pi$	`6.283185307179586`

 CC BY-NC-SA 4.0

Sabit	Anlamı	Değeri
`math.inf`	pozitif sonsuz	`inf`
`math.nan`	sayı olmayan değer	`nan`

`math.inf` her sayıdan büyük kabul edilir; bir en küçük değer aramasına başlarken ilk değer olarak kullanışlıdır. `math.nan` (not a number) ise tanımsız bir sonucu temsil eder.

3.5.2 Kök, Üs ve Logaritma

Kök, üs ve logaritma fonksiyonlarının sonucu her zaman `float`'tur.

Fonksiyon	Anlamı
`math.sqrt(x)`	$\sqrt{x}$
`math.pow(x, y)`	x^y, sonucu her zaman `float`
`math.exp(x)`	e^x
`math.log(x)`	$\ln x$
`math.log(x, b)`	$\log_b x$
`math.log10(x)`, `math.log2(x)`	$\log_{10} x$, $\log_2 x$

```
import math
print(math.sqrt(16))
print(math.pow(2, 3))
print(math.exp(1), math.exp(0))
print(math.log(100, 10), math.log2(8), math.log10(1000))
print(math.log(math.e))
```

Çıktı:

```
4.0
8.0
2.718281828459045 1.0
2.0 3.0 3.0
1.0
```

`math.pow(2, 3)` ile `2 ** 3` aynı sayıyı verir, ama `math.pow` sonucu her zaman `float` olarak verir. Negatif bir sayının karekökü gerçel sayı olmadığından `math.sqrt(-4)` hata verir:

```
import math
print(math.sqrt(-4))
```

Çıktı:

```
ValueError: math domain error
```

"math domain error" hatası, fonksiyona tanım kümesinin dışında bir değer verildiğini söyler.

3.5.3 Yuvarlama Fonksiyonları

`math` modülünde bir sayıyı tam sayıya indiren üç fonksiyon vardır.

Tanım 3.5 (Tavan, Taban ve Kesme). `math.ceil(x)` x'ten büyük ya da ona eşit en küçük tam sayıyı (**tavan**, $\lceil x \rceil$), `math.floor(x)` x'ten küçük ya da ona eşit en büyük tam sayıyı (**taban**, $\lfloor x \rfloor$), `math.trunc(x)` ise x'in ondalık kısmını atarak elde edilen tam sayıyı verir.

Yani `ceil` yukarı, `floor` aşağı yuvarlar; `trunc` ise sayıyı sıfıra doğru keser ve `int` ile aynı işi yapar. Pozitif sayılarda `floor` ile `trunc` aynı sonucu verir; fark negatif sayılarda ortaya çıkar.

 CC BY-NC-SA 4.0

```
import math
x = 3.7
print(math.ceil(x), math.floor(x), math.trunc(x))
y = -3.7
print(math.ceil(y), math.floor(y), math.trunc(y))
```

Çıktı:

```
4 3 3
-3 -4 -3
```

$-3{,}7$ sayısının tabanı -4'tür, çünkü $-4 \le -3{,}7$ ve aradaki tam sayı -3, $-3{,}7$'den büyüktür. Kesme ise ondalık kısmı atar ve -3 verir. Tam bölme `//` da sonucu `floor` gibi aşağı yuvarlar.

3.5.4 Trigonometrik Fonksiyonlar

`math.sin(x)`, `math.cos(x)` ve `math.tan(x)` fonksiyonları x'i **radyan** cinsinden kabul eder.
Derece cinsinden bir açıyla çalışırken önce radyana çevirmek gerekir. Bunun için ya $x \cdot \pi/180$ hesaplanır ya da `math.radians(x)` kullanılır; ters yöndeki dönüşümü `math.degrees(x)` yapar. Modülde kotanjant fonksiyonu yoktur; $\cot x = 1/\tan x$ olarak hesaplanır.

```
import math
print(math.sin(30))                  # 30 radyanın sinüsü
print(math.sin(30 * math.pi / 180))  # 30 derecenin sinüsü
print(math.sin(math.radians(90)))
print(math.cos(math.radians(60)))
print(math.degrees(math.pi / 2))
print(1 / math.tan(math.radians(45)))   # cot 45°
```

Çıktı:

```
-0.9880316240928618
0.49999999999999994
1.0
0.5000000000000001
90.0
1.0000000000000002
```

Birinci satırdaki `-0.988...` sonucu yanlış değildir; yalnız 30 derecenin değil 30 radyanın sinüsüdür. $\sin 30^\circ = 0{,}5$ olmasına karşın çıktı `0.49999999999999994` olur: bu, π sayısının bilgisayarda sınırlı duyarlıkla saklanmasından kaynaklanan küçük bir yuvarlama hatasıdır.

Örnek 3.3 (Radyana Çevirmeyi Unutmak). Aşağıdaki programın son satırı neden 1 yazmaz?

```
import math
aci = math.degrees(math.pi / 2)
print(aci)
print(math.sin(aci))
```

> **Çözüm.**
> `math.degrees(math.pi / 2)` radyanı dereceye çevirir ve `aci` değişkenine 90,0 değerini verir. Ama `math.sin` her zaman radyan bekler; `math.sin(aci)` bu yüzden 90°'nin değil, 90 radyanın sinüsünü hesaplar:
>
> ```
> import math
> aci = math.degrees(math.pi / 2)
> print(aci)
> print(math.sin(aci))
> ```
>
> Çıktı:

 CC BY-NC-SA 4.0

```
90.0
0.8939966636005579
```

Doğru sonuç için açı tekrar radyana çevrilmelidir: `math.sin(math.radians(aci))` 1,0 verir. ■

3.5.5 Kalanlar ve Mutlak Değer

`math` modülünde kalan hesaplayan iki fonksiyon daha vardır; `%` operatöründen yalnız negatif sayılarda ve en yakın tam sayıyı kullanma biçiminde ayrılırlar.

İfade	Kural	$x=7, y=3$	$x=20, y=3$	$x=-7, y=3$
`x % y`	kalan bölenle aynı işaretli	`1`	`2`	`2`
`math.fmod(x, y)`	kalan bölünenle aynı işaretli	`1.0`	`2.0`	`-1.0`
`math.remainder(x, y)`	$x - ny$, n: x/y'ye en yakın tam sayı	`1.0`	`-1.0`	`-1.0`

`math.remainder` için $20/3 = 6{,}67$ sayısına en yakın tam sayı $n = 7$'dir ve kalan $20 - 7 \cdot 3 = -1$ olur. $-7/3 = -2{,}33$ için $n = -2$ ve kalan $-7 - (-2) \cdot 3 = -1$ bulunur. `math.fabs(x)` ise $\mid x \mid$ değerini her zaman `float` olarak verir.

```
import math
print(7 % 3, math.fmod(7, 3), math.remainder(7, 3))
print(20 % 3, math.fmod(20, 3), math.remainder(20, 3))
print(-7 % 3, math.fmod(-7, 3), math.remainder(-7, 3))
print(math.fabs(-7.5), math.fabs(-7))
```

Çıktı:

```
1 1.0 1.0
2 2.0 -1.0
2 -1.0 -1.0
7.5 7.0
```

3.5.6 Sayma ve Bölünebilme Fonksiyonları

Kombinatorik ve sayılar teorisinde sık kullanılan fonksiyonlar tam sayılarla çalışır ve tam sayı verir.

Fonksiyon	Anlamı	Örnek	Sonuç
`math.factorial(n)`	$n! = 1 \cdot 2 \cdots n$	`math.factorial(5)`	`120`
`math.perm(n, k)`	n elemandan k'lı sıralama sayısı $\frac{n!}{(n-k)!}$	`math.perm(5, 3)`	`60`
`math.comb(n, k)`	n elemandan k'lı seçim sayısı $\binom{n}{k}$	`math.comb(5, 3)`	`10`
`math.gcd(a, b)`	en büyük ortak bölen	`math.gcd(48, 18)`	`6`
`math.lcm(a, b)`	en küçük ortak kat	`math.lcm(4, 6)`	`12`

```
import math
print(math.factorial(5))
print(math.perm(5, 3), math.comb(5, 3))
print(math.gcd(48, 18), math.lcm(4, 6))
```

Çıktı:

 CC BY-NC-SA 4.0

```
120
60 10
6 12
```

5 kişiden 3'ünü bir sıraya dizmenin $5 \cdot 4 \cdot 3 = 60$ yolu vardır; sıranın önemli olmadığı seçimlerde bu sayı $3! = 6$'ya bölünür ve $\binom{5}{3} = 10$ bulunur.

3.5.7 Toplam ve Çarpım

`math.fsum` ve `math.prod`, köşeli parantez içinde yazılmış bir sayı listesinin elemanlarını toplar ve çarpar. Listeleri ileride ayrıntılı göreceğiz; şimdilik `[1.5, 2.5, 3.5]` yazımının üç sayıyı bir arada tuttuğunu bilmek yeter.
`math.fsum`, sayıları tek tek `+` ile toplamaya göre daha duyarlı bir yöntem kullanır: ara toplamlardaki yuvarlama hatalarını takip eder ve sonucu olabildiğince doğru verir.

```
import math
print(math.fsum([1.5, 2.5, 3.5]))
print(math.prod([1, 2, 3, 4]))
print(0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1)
print(math.fsum([0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]))
```

Çıktı:

```
7.5
24
0.9999999999999999
1.0
```

On tane 0,1'in toplamı 1'dir. `+` ile soldan sağa toplarken her adımdaki küçük hata bir sonrakine aktarılır ve sonuç `0.9999999999999999` çıkar; `math.fsum` ise doğru sonucu verir. Python 3.12'den beri hazır `sum` fonksiyonu da ondalıklı sayıları benzer bir duyarlı yöntemle toplar.

3.6 Karmaşık Sayılarla İşlemler

Karmaşık sayılarla dört işlem, matematikteki kurallarla doğrudan yapılır.
$z_1 = 2 + 3i$ ve $z_2 = 1 + 4i$ için çarpım

$$(2 + 3i)(1 + 4i) = 2 + 8i + 3i + 12i^2 = -10 + 11i$$

olur. `abs(z)` fonksiyonu karmaşık sayının modülünü, yani $|\, a + bi \,| = \sqrt{a^2 + b^2}$ değerini verir; `z.conjugate()` ise eşleniği $\overline{z} = a - bi$ değerini verir.

```
z1 = 2 + 3j
z2 = 1 + 4j
print(z1 + z2)
print(z1 - z2)
print(z1 * z2)
print(z1 / z2)
print(abs(z2))
print(z2.conjugate())
print(z2 * z2)
```

Çıktı:

```
(3+7j)
(1-1j)
(-10+11j)
(0.8235294117647058-0.29411764705882354j)
4.123105625617661
(1-4j)
(-15+8j)
```

$|\, z_2 \,| = \sqrt{1^2 + 4^2} = \sqrt{17} \approx 4{,}1231$'dir. Bölme de eşlenikle genişletilerek doğrulanabilir:

 CC BY-NC-SA 4.0

$$\frac{2+3i}{1+4i} = \frac{(2+3i)(1-4i)}{(1+4i)(1-4i)} = \frac{14-5i}{17} \approx 0{,}8235 - 0{,}2941\,i.$$

3.7 Çeşitli Örnekler

Bu bölümde öğrendiklerimizi birkaç hesap üzerinde bir arada kullanalım.

Örnek 3.4 (Toplamın Dörtte Biri ve Kalanı). `t = 12 + 25 + 15` ve `x = t / 4 + t % 4` olmak üzere `x`'in değerini bulunuz.

Çözüm.
$t = 52$'dir. $52/4 = 13{,}0$ ve $52 = 13 \cdot 4 + 0$ olduğundan `t % 4` 0'dır. Böylece $x = 13{,}0 + 0 = 13{,}0$ bulunur.

```
t = 12 + 25 + 15
x = t / 4 + t % 4
print(x)
```

Çıktı:

```
13.0
```

■

Örnek 3.5 (Bütün Operatörler Bir Arada). `a = 8` ve `b = 3` iken `a ** 2 - b * 4 + a // b - b % a` ifadesinin değerini bulunuz.

Çözüm.
Önce üs alınır: $8^2 = 64$. Sonra çarpma, tam bölme ve kalan hesaplanır: $3 \cdot 4 = 12$, $8 \,//\, 3 = 2$ ve $3 \,\%\, 8 = 3$; 3, 8'den küçük olduğu için $3 = 0 \cdot 8 + 3$ ve kalan 3'ün kendisidir. Son olarak soldan sağa $64 - 12 + 2 - 3 = 51$ bulunur.

```
a = 8
b = 3
sonuc = a ** 2 - b * 4 + a // b - b % a
print(sonuc)
```

Çıktı:

```
51
```

■

Örnek 3.6 (Parantezli Bir Tam Bölme). `x = 5` ve `y = 2` iken `(x + y) ** 2 // (x - y) + x * y % 3` ifadesinin değerini bulunuz.

Çözüm.
Parantezler önce hesaplanır: $x + y = 7$ ve $x - y = 3$. Sonra üs alınır: $7^2 = 49$. Ardından aynı öncelikteki işlemler soldan sağa yapılır: $49 \,//\, 3 = 16$, ayrıca $x \cdot y = 10$ ve $10 \,\%\, 3 = 1$. Sonuç $16 + 1 = 17$'dir.

```
x = 5
y = 2
z = (x + y) ** 2 // (x - y) + x * y % 3
print(z)
```

 CC BY-NC-SA 4.0

Çıktı:

```
17
```

■

Örnek 3.7 (İndirimli ve Vergili Fiyat). Fiyatı 149,99 TL olan bir üründen 7 adet alınıyor. Toplam tutara %20 indirim, indirimli tutara da %18 vergi uygulanıyor. Ödenecek tutarı iki ondalık basamağa yuvarlayarak bulunuz.

Çözüm.
Toplam $149{,}99 \cdot 7 = 1049{,}93$ TL'dir. İndirim tutarı toplamın %20'si, yani $1049{,}93 \cdot 20/100 = 209{,}986$ TL'dir; indirimli tutar $1049{,}93 - 209{,}986 = 839{,}944$ TL olur. Vergi $839{,}944 \cdot 18/100 = 151{,}18992$ TL'dir ve ödenecek tutar

$$839{,}944 + 151{,}18992 = 991{,}13392 \approx 991{,}13$$

TL bulunur.

```
fiyat = 149.99
adet = 7
toplam = fiyat * adet
indirimli = toplam - toplam * 20 / 100
vergi = indirimli * 18 / 100
son_fiyat = indirimli + vergi
print(round(son_fiyat, 2))
```

Çıktı:

```
991.13
```

İndirimi hesaplarken `/` yerine `//` yazılsaydı `toplam * 20 // 100` işlemi 209,986'yı aşağı yuvarlayıp 209,0 yapardı ve sonuç bir lira kadar farklı çıkardı. Para hesaplarında tam bölme ancak bilerek kullanılmalıdır. ■

3.8 Alıştırmalar

Her alıştırmada önce sonucu elle bulun, sonra kodu çalıştırarak doğrulayın.

Alıştırma 3.1 (Saniyeyi Saat, Dakika ve Saniyeye Ayırmak). 10 000 saniyenin kaç saat, kaç dakika ve kaç saniye olduğunu `//` ve `%` kullanarak bulunuz.

Çözüm.
Bir saatte 3600 saniye vardır. $10000 \,//\, 3600 = 2$ saat eder ve geriye $10000 \,\%\, 3600 = 2800$ saniye kalır. Bu kalanda $2800 \,//\, 60 = 46$ dakika vardır ve geriye $2800 \,\%\, 60 = 40$ saniye kalır. Sağlama:

$$2 \cdot 3600 + 46 \cdot 60 + 40 = 7200 + 2760 + 40 = 10000$$

.

```
toplam_saniye = 10000
saat = toplam_saniye // 3600
kalan = toplam_saniye % 3600
dakika = kalan // 60
saniye = kalan % 60
print(saat, "saat", dakika, "dakika", saniye, "saniye")
```

 CC BY-NC-SA 4.0

Çıktı:

```
2 saat 46 dakika 40 saniye
```

■

Alıştırma 3.2 (Üç Basamaklı Sayının Rakamları). `n = 472` sayısının yüzler, onlar ve birler basamağını `//` ve `%` ile bulup rakamlarının toplamını hesaplayınız.

Çözüm.
Birler basamağı 10'a bölümden kalandır: $472 \,\%\, 10 = 2$. Onlar basamağı için önce sayı 10'a tam bölünür, $472 \,//\, 10 = 47$, sonra bunun birler basamağı alınır: $47 \,\%\, 10 = 7$. Yüzler basamağı $472 \,//\, 100 = 4$'tür. Rakamlar toplamı $4 + 7 + 2 = 13$'tür.

```
n = 472
yuzler = n // 100
onlar = (n // 10) % 10
birler = n % 10
print(yuzler, onlar, birler)
print(yuzler + onlar + birler)
```

Çıktı:

```
4 7 2
13
```

■

Alıştırma 3.3 (Hipotenüs). Dik kenarları 5 ve 12 olan dik üçgenin hipotenüsünü `math.sqrt` ile bulunuz.

Çözüm.
Pisagor teoremine göre hipotenüs

$$c = \sqrt{a^2 + b^2} = \sqrt{25 + 144} = \sqrt{169} = 13$$

'tür.

```
import math
a = 5
b = 12
c = math.sqrt(a ** 2 + b ** 2)
print(c)
```

Çıktı:

```
13.0
```

`math.sqrt` sonucu her zaman `float` verdiği için çıktı `13` değil `13.0`'dır. ■

Alıştırma 3.4 (Faktöriyelle Kombinasyon). $\binom{10}{3} = \dfrac{10!}{3!\,7!}$ değerini `math.factorial` ile hesaplayıp `math.comb(10, 3)` ile karşılaştırınız.

 CC BY-NC-SA 4.0

Çözüm.
$10!/7! = 10 \cdot 9 \cdot 8 = 720$ ve $3! = 6$ olduğundan $\binom{10}{3} = 720/6 = 120$'dir. Python'da bölmeyi `//` ile yaparız; payın paydaya tam bölündüğünü bildiğimiz için sonuç kayıpsız bir tam sayı olur.

```
import math
pay = math.factorial(10)
payda = math.factorial(3) * math.factorial(7)
print(pay // payda)
print(math.comb(10, 3))
```

Çıktı:

```
120
120
```

`/` kullanılsaydı sonuç `120.0` olurdu. Büyük sayılarda `/` ile bölme, sonuç tam sayı olsa bile `float`'a geçtiği için basamak kaybına yol açabilir. ■

Alıştırma 3.5 (Negatif Sayıyı Tam Sayıya İndirmek). $x = -2{,}5$ için `int(x)`, `round(x)`, `math.floor(x)`, `math.ceil(x)` ve `x // 1` değerlerini bulunuz.

Çözüm.
Kuralları tek tek uygulayalım:

- `int(x)` sıfıra doğru keser: -2.
- `round(x)`: $-2{,}5$, -3 ile -2'nin tam ortasındadır; çift olan komşu -2'dir.
- `math.floor(x)` aşağı yuvarlar: -3.
- `math.ceil(x)` yukarı yuvarlar: -2.
- `x // 1` tam bölmedir ve aşağı yuvarlar; `x` `float` olduğu için sonuç `-3.0`'dır.

```
import math
x = -2.5
print(int(x), round(x), math.floor(x), math.ceil(x), x // 1)
```

Çıktı:

```
-2 -2 -3 -2 -3.0
```

■

Alıştırma 3.6 (Bölme Eşitliğini Doğrulamak). $a = -17$ ve $b = 5$ için `a // b` ve `a % b` değerlerini bulunuz ve $a = (a \,//\, b) \cdot b + (a \,\%\, b)$ eşitliğini doğrulayınız.

Çözüm.
$-17/5 = -3{,}4$ sayısı aşağı yuvarlanınca -4 olur; bu yüzden `a // b` -4'tür. Kalan eşitlikten bulunur:

$$-17 - (-4) \cdot 5 = -17 + 20 = 3$$

. Kalan, bölen 5 gibi pozitiftir. Doğrulama: $(-4) \cdot 5 + 3 = -20 + 3 = -17$.

```
a = -17
b = 5
print(a // b, a % b)
print((a // b) * b + a % b)
```

 CC BY-NC-SA 4.0

Çıktı:

```
-4 3
-17
```

■

Alıştırma 3.7 (Kalan, Çarpma ve Parantez). `Y = 4 % 8 * 4 + 2 * (2 ** 2 + 2)` komutundan sonra `Y`'nin değerini bulunuz.

Çözüm.
Önce parantez içi: $2^2 + 2 = 6$. Sonra aynı öncelikteki `%` ve `*` soldan sağa yapılır: $4 \,\%\, 8 = 4$, çünkü $4 = 0 \cdot 8 + 4$; ardından $4 \cdot 4 = 16$. Sağ tarafta $2 \cdot 6 = 12$'dir. Toplam $16 + 12 = 28$'dir.

```
Y = 4 % 8 * 4 + 2 * (2 ** 2 + 2)
print(Y)
```

Çıktı:

```
28
```

■

Alıştırma 3.8 (Karmaşık Sayının Modülü ve Tersi). $z = 3 - 4i$ için $\mid z \mid$ ve $1/z$ değerlerini bulunuz.

Çözüm.
$\mid z \mid = \sqrt{3^2 + (-4)^2} = \sqrt{25} = 5$'tir. Ters, eşlenikle genişletilerek bulunur:

$$\frac{1}{3 - 4i} = \frac{3 + 4i}{(3 - 4i)(3 + 4i)} = \frac{3 + 4i}{25} = 0{,}12 + 0{,}16\,i.$$

```
z = 3 - 4j
print(abs(z))
print(1 / z)
```

Çıktı:

```
5.0
(0.12+0.16j)
```

■

Bu bölümde Python'ı bir hesap makinesi olarak kullanmayı öğrendik: operatörler, işlem önceliği, yuvarlama ve `math` modülü artık elimizde. Şimdiye kadar bütün değerleri programın içine kendimiz yazdık. Bir sonraki bölüm Girdi ve Çıktı İşlemleri, kullanıcıdan değer almayı ve sonuçları düzenli bir biçimde ekrana yazmayı ele alıyor.

 CC BY-NC-SA 4.0

4. Girdi ve Çıktı İşlemleri

Şimdiye kadar yazdığımız programlar hep aynı sayılarla çalıştı: bir dairenin alanını hesaplarken yarıçapı programın içine kendimiz yazdık. Gerçek bir program ise değerleri kullanıcıdan alır, onlarla hesap yapar ve sonucu okunaklı bir biçimde kullanıcıya gösterir. Bir programın dış dünyayla bu alışverişine **girdi** (input) ve **çıktı** (output) denir.
Bu bölümde klavyeden değer almayı, alınan metni sayıya çevirmeyi ve kullanıcının yanlış bir değer girmesi hâlinde programın çökmesini önlemeyi göreceğiz. Ardından `print` fonksiyonunun inceliklerini, sonuçları istediğimiz biçimde yazdırmanın yollarını ve Python'ın hata mesajlarını okumayı öğreneceğiz.

4.1 Kullanıcıdan Değer Almak

Python'da kullanıcıdan bir değer almanın en yaygın yolu `input` fonksiyonudur.

Tanım 4.1 (input Fonksiyonu). `input(mesaj)` fonksiyonu ekrana `mesaj` metnini yazar, kullanıcının bir şeyler yazıp Enter'a basmasını bekler ve yazılanı **her zaman bir metin** (`str`) olarak verir.

Yani `input`'un içine yazılan metin kullanıcıya bir soru ya da talimattır; fonksiyonun verdiği değer ise kullanıcının yanıtıdır. Bu yanıt bir değişkene atanarak saklanır. Aşağıdaki çıktıda `Ulaş`, kullanıcının klavyeden yazdığı değerdir.

```
isim = input("Lütfen adınızı girin: ")
print("Merhaba, " + isim)
```

Çıktı:

```
Lütfen adınızı girin: Ulaş
Merhaba, Ulaş
```

Programda iki metni `+` ile birleştirdik.

Tanım 4.2 (Metin Birleştirme). İki metin arasındaki `+` operatörü, ikinci metni birincinin sonuna ekleyerek yeni bir metin oluşturur; buna **metin birleştirme** (concatenation) denir. `+` operatörünün iki yanındaki değerler metinse birleştirme, sayıysa toplama yapılır; bir metin ile bir sayı `+` ile birleştirilemez.

Yani `"Hız Sınırı " + "80"` işlemi `"Hız Sınırı 80"` metnini verir; burada `"80"` bir sayı değil, iki karakterlik bir metindir. `+` metinlerin arasına kendiliğinden boşluk koymaz; boşluk isteniyorsa metinlerden birinin içine yazılmalıdır.

```
birinci = "Hız Sınırı "
ikinci = "80"
print(birinci + ikinci)
print(birinci + ikinci + "!")
print(birinci + ikinci + "10")
```

Çıktı:

```
Hız Sınırı 80
Hız Sınırı 80!
Hız Sınırı 8010
```

Son satırda `"80" + "10"` toplanmadı, yan yana yazıldı ve `8010` oluştu.

 CC BY-NC-SA 4.0

4.1.1 Sayı Olarak Değer Almak

`input` her zaman metin verdiği için, kullanıcının girdiği bir sayıyla hesap yapmadan önce onu sayıya çevirmek gerekir.

```
a = input("Kenar uzunluğu: ")
alan = a * a
print("Karenin alanı =", alan)
```

Çıktı:

```
Kenar uzunluğu: 5
TypeError: can't multiply sequence by non-int of type 'str'
```

Kullanıcı `5` yazdı ama `a` değişkeni `5` sayısını değil, `"5"` metnini tutuyor; iki metin çarpılamadığı için program `TypeError` hatasıyla durdu. Çözüm, `input`'un verdiği metni Değişkenler, Veri Tipleri ve Atama bölümündeki tip dönüşüm fonksiyonlarıyla (Tanım 2.8) sayıya çevirmektir. Bunun için `input` çağrısı doğrudan `int` ya da `float` içine yazılır:

```
yas = int(input("Lütfen yaşınızı girin: "))
boy = float(input("Lütfen boyunuzu girin (örnek: 1.75): "))
print("Yaşınız:", yas)
print("Boyunuz:", boy)
print("Gelecek yıl", yas + 1, "yaşında olacaksınız.")
```

Çıktı:

```
Lütfen yaşınızı girin: 19
Lütfen boyunuzu girin (örnek: 1.75): 1.56
Yaşınız: 19
Boyunuz: 1.56
Gelecek yıl 20 yaşında olacaksınız.
```

`int(input(...))` yazımında önce içteki `input` çalışır ve bir metin verir; sonra dıştaki `int` bu metni tam sayıya çevirir. Ondalıklı sayı girilirken ayırıcı olarak virgül değil nokta kullanılmalıdır: `1,56` yazılırsa `float` onu okuyamaz.

Örnek 4.1 (İki Sayının Toplamı). Kullanıcıdan iki sayı alıp toplamlarını yazan bir program yazınız. Sayılar `int`'e çevrilmezse ne olur?

Çözüm.
Önce dönüşüm yapmadan deneyelim. Kullanıcı `5` ve `6` girsin:

```
A = input("1. sayıyı gir: ")
B = input("2. sayıyı gir: ")
T = A + B
print("Toplam =", T)
```

Çıktı:

```
1. sayıyı gir: 5
2. sayıyı gir: 6
Toplam = 56
```

`A` ve `B` metin olduğundan `+` toplama değil birleştirme yaptı ve `"5"` ile `"6"` yan yana yazılıp `56` oluştu. Hata mesajı çıkmadığı için bu tür bir yanlışlık fark edilmeden geçebilir. Değerleri `int`'e çevirince toplama yapılır:

```
A = int(input("1. sayıyı gir: "))
B = int(input("2. sayıyı gir: "))
```

 CC BY-NC-SA 4.0

```
T = A + B
print("Toplam =", T)
```

Çıktı:

```
1. sayıyı gir: 5
2. sayıyı gir: 6
Toplam = 11
```

Ondalıklı sayılarla da çalışması isteniyorsa `int` yerine `float` kullanılır. ■

4.2 Hatalı Girdileri Yakalamak

Kullanıcı bir sayı yerine harf ya da boş bir değer girerse dönüşüm başarısız olur ve program durur.

```
yas = int(input("Lütfen yaşınızı girin: "))
print("Yaşınız:", yas)
```

Çıktı:

```
Lütfen yaşınızı girin: 45.5
ValueError: invalid literal for int() with base 10: '45.5'
```

`45.5` bir tam sayı yazımı olmadığından `int` onu okuyamadı ve `ValueError` hatası oluştu. Programın böyle durumlarda çökmesi yerine kullanıcıyı uyarması için hatayı yakalamak gerekir.

Tanım 4.3 (try-except Yapısı). Hata çıkabilecek komutlar `try:` satırının altına, bir hata oluşursa çalışacak komutlar da `except HataTuru:` satırının altına girintili olarak yazılır. `try` bloğundaki komutlarda `HataTuru` türünde bir hata oluşursa Python o komutta durur, programı sonlandırmaz ve `except` bloğuna geçer; hata oluşmazsa `except` bloğu atlanır.

Yani `try-except`, "şunu dene; olmazsa şunu yap" demenin yoludur. Satır başındaki boşluklara **girinti** (indentation) denir ve Python hangi komutların `try`'a, hangilerinin `except`'e ait olduğunu bu girintilerden anlar. Girinti için genellikle dört boşluk kullanılır; Colab ve VS Code, iki noktayla biten bir satırdan sonra Enter'a basıldığında girintiyi kendiliğinden ekler.

```
try:
    yas = int(input("Lütfen yaşınızı girin: "))
    print("Yaşınız:", yas)
except ValueError:
    print("Geçersiz giriş! Lütfen bir tam sayı girin.")
```

Çıktı:

```
Lütfen yaşınızı girin: 45.5
Geçersiz giriş! Lütfen bir tam sayı girin.
```

Bir `try` bloğundan sonra farklı hata türleri için birden çok `except` bloğu yazılabilir. Aşağıdaki programda kullanıcı sıfır girerse sıfıra bölme hatası (`ZeroDivisionError`), sayı olmayan bir şey girerse `ValueError` yakalanır.

```
try:
    sayi = int(input("Bir sayı girin: "))
    sonuc = 10 / sayi
    print("Sonuç:", sonuc)
except ZeroDivisionError:
    print("Hata: Bir sayıyı sıfıra bölemezsiniz!")
except ValueError:
    print("Hata: Lütfen geçerli bir tam sayı girin!")
```

 CC BY-NC-SA 4.0

Çıktı:

```
Bir sayı girin: a
Hata: Lütfen geçerli bir tam sayı girin!
```

Aynı program kullanıcı `0` girdiğinde ise şöyle çalışır:

```
try:
    sayi = int(input("Bir sayı girin: "))
    sonuc = 10 / sayi
    print("Sonuç:", sonuc)
except ZeroDivisionError:
    print("Hata: Bir sayıyı sıfıra bölemezsiniz!")
except ValueError:
    print("Hata: Lütfen geçerli bir tam sayı girin!")
```

Çıktı:

```
Bir sayı girin: 0
Hata: Bir sayıyı sıfıra bölemezsiniz!
```

Kullanıcı yanlış girdikçe soruyu yeniden soran bir program için döngülere ihtiyaç var; bunu İç İçe Döngüler, while Döngüsü ve match-case bölümünde yapacağız.

4.2.1 eval Fonksiyonu

`input` ile alınan metni sayıya çevirmenin bir başka yolu `eval` fonksiyonudur.

Tanım 4.4 (eval Fonksiyonu). `eval(metin)` fonksiyonu, kendisine verilen metni bir Python ifadesi olarak hesaplar ve sonucunu verir.

Yani `eval("5.6")` `5.6` sayısını, `eval("2 + 3")` ise `5` sayısını verir. Kullanıcı tam sayı girerse `int`, ondalıklı sayı girerse `float` elde edilir; hangi tipte bir sayı geleceğini önceden bilmek gerekmez.

```
a = eval(input("Kenar uzunluğu: "))
alan = a * a
print("Karenin alanı =", alan)
print(type(a))
```

Çıktı:

```
Kenar uzunluğu: 5.6
Karenin alanı = 31.359999999999996
<class 'float'>
```

$5{,}6^2 = 31{,}36$ olması gerekirken çıktıda görülen küçük fark, ondalıklı sayıların bilgisayarda yaklaşık saklanmasından kaynaklanır.

⚠ eval tehlikelidir

`eval` kendisine verilen metni Python kodu olarak çalıştırır. Kullanıcı bir sayı yerine bilgisayardaki dosyaları silen bir komut yazarsa `eval` onu da çalıştırır. Bu yüzden kullanıcıdan gelen girdiyi `eval` ile değil, `int` ya da `float` ile çevirmek her zaman daha güvenlidir.

4.3 Ekrana Yazdırmak: print

Sonuçları ekrana yazmak için kullandığımız `print` fonksiyonu, ilk bakışta göründüğünden daha esnektir. `print`'e virgülle ayrılmış birden çok değer verilebilir; bu değerler metin, sayı ya da ifade olabilir. `print` her değeri yazar, değerlerin arasına birer boşluk koyar ve en sonda yeni bir satıra geçer.

```
isim = "Ahmet"
yas = 30
```

 CC BY-NC-SA 4.0

```
a = 5
b = 10
print("İsim:", isim)
print("Yaş:", yas)
print(a, "+", b, "=", a + b)
print("Merhaba" + "Arkadaşlar")
print("Merhaba", "Arkadaşlar")
```

Çıktı:

```
İsim: Ahmet
Yaş: 30
5 + 10 = 15
MerhabaArkadaşlar
Merhaba Arkadaşlar
```

Son iki satırın farkına dikkat edin: `+` metinleri boşluksuz birleştirir, virgül ise `print`'in araya bir boşluk koymasını sağlar.

Tanım 4.5 (sep ve end Parametreleri). `print` fonksiyonunun `sep` parametresi yazılan değerlerin arasına konacak metni, `end` parametresi de yazdırmanın sonuna eklenecek metni belirler. Varsayılan olarak `sep` bir boşluk (`" "`), `end` ise yeni satır karakteri (`"\n"`) olur.

Yani `print("a", "b", sep="-")` değerlerin arasına boşluk yerine `-` koyar ve `a-b` yazar. `end` değiştirildiğinde `print` yazdıktan sonra yeni satıra geçmez; bir sonraki `print` aynı satırdan devam eder.

```
print("Python", "programlama", "dili")
print("Python", "programlama", "dili", sep="-")
print("Python", "programlama", "dili", sep="$")
print("Python", "programlama", "dili", sep="")
print("Merhaba", end=", ")
print("Dünya!")
print("Python kullanmayı biliyor musun", end="?")
```

Çıktı:

```
Python programlama dili
Python-programlama-dili
Python$programlama$dili
Pythonprogramlamadili
Merhaba, Dünya!
Python kullanmayı biliyor musun?
```

İlk `print("Merhaba", end=", ")` satır sonunda yeni satıra geçmek yerine `, ` yazdı; bu yüzden `Dünya!` aynı satıra geldi. `sep` yalnız birden çok değer yazdırılırken etkilidir: `print("Python", sep="&&")` tek bir değer olduğu için yalnız `Python` yazar.

4.4 Biçimli Yazdırma

Bir sonucu metnin içine yerleştirmek ya da ondalıklı bir sayının kaç basamağının yazılacağını belirlemek için biçimli yazdırma kullanılır; Python bunun için üç yol sunar.

4.4.1 f-string

En yeni ve en okunaklı yol f-string'dir.

Tanım 4.6 (f-string). Tırnağın önüne `f` harfi konarak yazılan metne **f-string** denir. Bir f-string'in içinde süslü parantez `{ }` arasına yazılan her ifade hesaplanır ve değeri metnin o yerine yerleştirilir.

Yani `f"{isim} {yas} yaşındadır."` metninde `{isim}` ve `{yas}` yerine değişkenlerin değerleri yazılır. `f` unutulursa süslü parantezler olduğu gibi yazılır.

 CC BY-NC-SA 4.0

```
isim = "Ali"
yas = 25
print("{isim} {yas} yaşındadır.")
print(f"{isim} {yas} yaşındadır.")
print(f"{isim} gelecek yıl {yas + 1} yaşında olacak.")
a = 5
b = 10
print(f"{a}+{b}=", a + b)
print(f"{a}+{b}={a + b}")
print(f"{a}+{b}=,a+b")
```

Çıktı:

```
{isim} {yas} yaşındadır.
Ali 25 yaşındadır.
Ali gelecek yıl 26 yaşında olacak.
5+10= 15
5+10=15
5+10=,a+b
```

Son satırda `a+b` süslü parantez içinde olmadığı için hesaplanmadı ve metin olarak yazıldı.
Süslü parantezin içinde ifadeden sonra iki nokta ve bir **biçim belirteci** yazılabilir. En sık kullanılanı `:.nf` belirtecidir: değeri virgülden sonra `n` basamakla yazar.

```
a = 10 / 3
b = 2.5 * 4.8
c = 7.123456
print(f"a = {a:.3f}")
print(f"b = {b:.3f}")
print(f"c = {c:.3f}")
print(f"a = {round(a, 3)}")
print(f"b = {round(b, 3)}")
```

Çıktı:

```
a = 3.333
b = 12.000
c = 7.123
a = 3.333
b = 12.0
```

`:.3f` ile `round(..., 3)` arasındaki farka dikkat edin: `round` bir sayı verir ve sondaki sıfırlar yazılmaz (`12.0`); `:.3f` ise sayıyı her zaman tam üç basamakla yazar (`12.000`). Tablo gibi hizalı çıktılarda `:.3f` tercih edilir.

4.4.2 format Metodu

f-string'den önce yaygın olan yol, metnin `format` metodudur.
`"... {} ... {} ...".format(d1, d2)` yazımında metindeki her `{}` sırayla `format`'a verilen değerlerle doldurulur. Süslü parantezin içine sıra numarası yazılarak değerlerin sırası değiştirilebilir; sıra numaraları 0'dan başlar. Biçim belirteci yine iki noktadan sonra yazılır.

```
import math
isim = "Ali"
yas = 30
print("{}, {} yaşındadır.".format(isim, yas))
print("{}, {} yaşındadır.".format(yas, isim))
metin = "Girilen {}, {} ve {} sayılarının toplamı {}."
print(metin.format(5, 6, 9, 5 + 6 + 9))
print("PI = {0:.4f}".format(math.pi))
print("{1} {0} {2:.2f}".format(10, "Python", 20))
```

Çıktı:

 CC BY-NC-SA 4.0

```
Ali, 30 yaşındadır.
30, Ali yaşındadır.
Girilen 5, 6 ve 9 sayılarının toplamı 20.
PI = 3.1416
Python 10 20.00
```

İkinci satırda değerlerin sırası karıştırıldığı için anlamsız bir cümle çıktı: `format` değerleri adlarına göre değil, sırasına göre yerleştirir. Son satırda `{1}` ikinci değeri (`"Python"`), `{0}` birinci değeri (`10`), `{2:.2f}` üçüncü değeri iki basamakla (`20.00`) yazdı.

4.4.3 Yüzde İşaretiyle Biçimlendirme

Python'ın en eski biçimlendirme yolu, C dilinden gelen `%` yazımıdır; eski kodlarda hâlâ sık görülür. Metnin içinde `%d` bir tam sayının, `%f` bir ondalıklı sayının, `%s` bir metnin yerini tutar; değerler metinden sonra `%` işaretiyle ve parantez içinde sırayla verilir. `%.2f` yazımı ondalıklı sayıyı iki basamakla yazar; yalnız `%f` yazılırsa altı basamak kullanılır.

```
isim = "Ali"
yas = 30
print("%s %d yaşındadır." % (isim, yas))
print("%d yaşında olan %s okula geldi." % (yas, isim))
print("%d ile %d sayılarının toplamı %f" % (5, 3, 5 + 3))
print("Ortalama: %.2f" % (17 / 3))
```

Çıktı:

```
Ali 30 yaşındadır.
30 yaşında olan Ali okula geldi.
5 ile 3 sayılarının toplamı 8.000000
Ortalama: 5.67
```

Bu üç yol aynı işi yapar. Yeni kod yazarken en okunaklı olan f-string önerilir.

4.5 Kaçış Karakterleri

Bir metnin içine yeni satır ya da sekme gibi klavyeden doğrudan yazılamayan karakterler koymak için ters eğik çizgi kullanılır.

Tanım 4.7 (Kaçış Karakteri). Metin içinde ters eğik çizgiyle (`\`) başlayan ve tek bir özel karakteri temsil eden ikiliye **kaçış karakteri** (escape character) denir. En sık kullanılanlar şunlardır:

Yazım	Anlamı
`\n`	yeni satır
`\t`	sekme (tab) boşluğu
`\\`	ters eğik çizginin kendisi
`\'`	tek tırnak
`\"`	çift tırnak

Yani `"Can\nEvi"` metni ekrana iki satır olarak yazılır. Ters eğik çizginin kendisini yazdırmak için onu iki kez yazmak gerekir, çünkü tek bir `\` sonraki karakterle birlikte bir kaçış karakteri olarak okunur.

```
print("Matematik\nİstatistik")
print("Matematik\tİstatistik")
print("Matematik \\ İstatistik")
print("p", "i", "t", "o", "n", sep="\n")
```

Çıktı:

 CC BY-NC-SA 4.0

```
Matematik
İstatistik
Matematik   İstatistik
Matematik \ İstatistik
p
i
t
o
n
```

Tırnaklar için kural şudur: metin çift tırnakla yazılmışsa içinde tek tırnak serbestçe kullanılabilir, tek tırnakla yazılmışsa çift tırnak serbestçe kullanılabilir. Metni açan tırnakla aynı türden bir tırnak metnin içinde kullanılacaksa önüne `\` konur; yoksa Python metnin orada bittiğini sanır.

```
print("Bursa'nın iskenderi meşhurdur.")
print('Bursa\'nın iskenderi meşhurdur.')
print('Merhaba, "Python" kullanıyorum.')
print("Merhaba, \"Python\" kullanıyorum.")
```

Çıktı:

```
Bursa'nın iskenderi meşhurdur.
Bursa'nın iskenderi meşhurdur.
Merhaba, "Python" kullanıyorum.
Merhaba, "Python" kullanıyorum.
```

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

Çıktı:

```
SyntaxError: unterminated string literal (detected at line 1)
```

Bu satırda metin `'Bursa'` ile bitmiş sayılır; Python ardından gelen `nın iskenderi meşhurdur.'` kısmını anlamlandıramaz ve yazım hatası verir.

4.6 Hata Türleri

Python bir hatayla karşılaştığında programı durdurur ve hatanın türünü ve nedenini söyleyen bir mesaj yazar; bu mesajı okuyabilmek, hatayı bulmanın en kısa yoludur.
Colab ve IDLE hata mesajında hatanın bulunduğu satırı da gösterir ve yazım hatalarında hatalı yerin altına `^` işareti koyar. Mesajın son satırı hatanın türünü ve kısa bir açıklamasını içerir. Sık karşılaşılan hata türleri şunlardır:

Hata türü	Ne zaman oluşur
`SyntaxError`	kod Python'ın yazım kurallarına uymuyor: kapanmamış parantez ya da tırnak, eksik iki nokta
`IndentationError`	girintiler kurala uymuyor
`NameError`	tanımlanmamış bir ad kullanılıyor
`TypeError`	bir işlem uygun olmayan tipte bir değere uygulanıyor: metin ile sayıyı toplamak
`ValueError`	tipi doğru ama değeri uygunsuz bir argüman veriliyor: `int("abc")`
`ZeroDivisionError`	bir sayı sıfıra bölünüyor
`IndexError`	bir listede ya da metinde var olmayan bir sıra numarasına erişiliyor
`KeyError`	bir sözlükte var olmayan bir anahtar kullanılıyor

 CC BY-NC-SA 4.0

Hata türü	Ne zaman oluşur
`AttributeError`	bir değerin sahip olmadığı bir özelliği ya da metodu kullanılıyor
`ModuleNotFoundError`	`import` edilen modül bulunamıyor; `ImportError` türünün özel bir hâlidir

Listeleri ve sözlükleri ilerideki bölümlerde göreceğiz. Aşağıdaki kısa programlar bu hatalardan bazılarını gösteriyor:

```
print(sayi)
```

Çıktı:

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

```
print(10 / 0)
```

Çıktı:

```
ZeroDivisionError: division by zero
```

```
print(int("abc"))
```

Çıktı:

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

```
sayi = 5
    print(sayi)
```

Çıktı:

```
IndentationError: unexpected indent
```

```
import matematik
```

Çıktı:

```
ModuleNotFoundError: No module named 'matematik'
```

```
isim = "Ali"
print(isim[3])
```

Çıktı:

```
IndexError: string index out of range
```

Son örnekte `isim[3]`, `"Ali"` metninin dördüncü karakterini ister; metinde üç karakter olduğu için `IndexError` oluştu. Sıra numaralarının neden 0'dan başladığını listelerle birlikte göreceğiz.

> **Hata ayıklamak üç adımda**
>
> 1. Hata mesajının son satırını okuyun: hata türü sorunun ne olduğunu söyler.
> 2. Mesajda gösterilen satıra gidin; yazım hatalarında `^` işaretinin hemen önüne de bakın.
> 3. Neden hâlâ belli değilse, hatadan önceki satırlara `print` ekleyerek değişkenlerin o andaki değerlerini ve tiplerini yazdırın.

Örnek 4.2 (Bir Hatayı Adım Adım Bulmak). Aşağıdaki program kullanıcıdan bir doğum yılı alıp yaşı hesaplamak istiyor ama hata veriyor. Hatayı bulunuz ve düzeltiniz.

 CC BY-NC-SA 4.0

```
dogum_yili = input("Doğum yılınız: ")
yas = 2026 - dogum_yili
print("Yaşınız:", yas)
```

Çözüm.
Birinci adım. Programı çalıştırıp mesajın son satırını okuyalım:

```
dogum_yili = input("Doğum yılınız: ")
yas = 2026 - dogum_yili
print("Yaşınız:", yas)
```

Çıktı:

```
Doğum yılınız: 2004
TypeError: unsupported operand type(s) for -: 'int' and 'str'
```

Mesaj, bir `int` ile bir `str` arasında `-` işleminin yapılamadığını söylüyor.
İkinci adım. Mesajda gösterilen satır `yas = 2026 - dogum_yili` satırıdır. Bu satırda `2026` bir `int` olduğuna göre `str` olan `dogum_yili` olmalıdır.
Üçüncü adım. Tahminimizi `print(type(dogum_yili))` ekleyerek doğrulayabiliriz; `input` her zaman metin verdiği için `<class 'str'>` yazılır. Düzeltme, değeri `int`'e çevirmektir:

```
dogum_yili = int(input("Doğum yılınız: "))
yas = 2026 - dogum_yili
print("Yaşınız:", yas)
```

Çıktı:

```
Doğum yılınız: 2004
Yaşınız: 22
```

■

4.7 İletişim Pencereleri

Bu derste girdi ve çıktı için hep metin satırlarını kullanıyoruz; Python'da düğmeli ve kutulu pencerelerle çalışmak da mümkündür.
Python kurulumuyla birlikte gelen `tkinter` modülüyle pencereler, düğmeler ve giriş kutuları oluşturulabilir; `turtle` modülü de bir pencerede çizim yapmayı sağlar. `easygui` gibi kurulumla gelmeyen ve ayrıca yüklenen paketler de vardır. Bu modüller bir ekran penceresi açmaya ihtiyaç duyduğundan Colab'da doğrudan çalışmaz; IDLE ya da VS Code ile kendi bilgisayarımızda kullanılabilirler. Matematiksel hesap yapan programlarda pek tercih edilmezler.

4.8 Alıştırmalar

Girdi alan alıştırmalarda çıktıdaki kullanıcı girdileri örnek olarak seçilmiştir; farklı değerlerle de deneyin.

Alıştırma 4.1 (Santigrattan Fahrenhayta). Kullanıcıdan santigrat cinsinden bir sıcaklık alıp $F = \frac{9}{5}C + 32$ formülüyle fahrenhayta çeviren bir program yazınız. Kullanıcı 36,6 girdiğinde ne yazılır?

Çözüm.
Sıcaklık ondalıklı olabileceğinden `float` ile çevirir. $C = 36{,}6$ için

$$F = 1{,}8 \cdot 36{,}6 + 32 = 65{,}88 + 32 = 97{,}88$$

'dir. Sonucu f-string ile iki basamakla yazdıralım:

 CC BY-NC-SA 4.0

```
c = float(input("Sıcaklık (°C): "))
f = 9 / 5 * c + 32
print(f"{c} °C = {f:.2f} °F")
```

Çıktı:

```
Sıcaklık (°C): 36.6
36.6 °C = 97.88 °F
```

■

Alıştırma 4.2 (Dairenin Alanı ve Çevresi). Kullanıcıdan bir yarıçap alıp dairenin alanını ve çevresini virgülden sonra üç basamakla yazan bir program yazınız. Kullanıcı $2{,}5$ girdiğinde ne yazılır?

Çözüm.
Alan πr^2, çevre $2\pi r$'dir. $r = 2{,}5$ için alan $6{,}25\pi \approx 19{,}635$, çevre $5\pi \approx 15{,}708$'dir. π için `math.pi` kullanırız:

```
import math
r = float(input("Yarıçap: "))
alan = math.pi * r ** 2
cevre = 2 * math.pi * r
print(f"Alan: {alan:.3f}")
print(f"Çevre: {cevre:.3f}")
```

Çıktı:

```
Yarıçap: 2.5
Alan: 19.635
Çevre: 15.708
```

■

Alıştırma 4.3 (Tarihi Noktalarla Yazmak). `gun = 4`, `ay = 11` ve `yil = 2026` değişkenlerini kullanarak ekrana `4.11.2026` yazdırınız: önce `sep` ile, sonra bir f-string ile.

Çözüm.
`print`'e üç değer verip ayırıcı olarak nokta seçersek değerler aralarında nokta ile yazılır. f-string'de ise noktaları metnin içine kendimiz yazarız:

```
gun = 4
ay = 11
yil = 2026
print(gun, ay, yil, sep=".")
print(f"{gun}.{ay}.{yil}")
```

Çıktı:

```
4.11.2026
4.11.2026
```

■

 CC BY-NC-SA 4.0

Alıştırma 4.4 (Kaçış Karakterlerinin Çıktısı). Aşağıdaki programın çıktısını bulunuz.

```
print("ab\'c")
print("ab\"c")
print("ab\\c")
print("a\tb\nc")
```

Çözüm.
`\'` tek tırnağı, `\"` çift tırnağı, `\\` ters eğik çizgiyi temsil eder; bu yüzden ilk üç satırda `ab'c`, `ab"c` ve `ab\c` yazılır. Dördüncü satırda `a` ile `b` arasına bir sekme boşluğu girer, `\n` ise `c`'yi bir alt satıra geçirir.

```
print("ab\'c")
print("ab\"c")
print("ab\\c")
print("a\tb\nc")
```

Çıktı:

```
ab'c
ab"c
ab\c
a   b
c
```

■

Alıştırma 4.5 (Güvenli Bölme). Kullanıcıdan iki sayı alıp birinciyi ikinciye bölen bir program yazınız. Kullanıcı sayı olmayan bir değer girerse ya da ikinci sayı sıfırsa program çökmeden uygun bir mesaj yazmalıdır. Kullanıcı `7` ve `0` girdiğinde ne yazılır?

Çözüm.
İki hata türü olabilir: dönüşüm başarısız olursa `ValueError`, bölen sıfırsa `ZeroDivisionError`. İkisi için ayrı `except` blokları yazarız:

```
try:
    a = float(input("Bölünen: "))
    b = float(input("Bölen: "))
    print(f"{a} / {b} = {a / b}")
except ValueError:
    print("Lütfen geçerli bir sayı girin.")
except ZeroDivisionError:
    print("Bölen sıfır olamaz.")
```

Çıktı:

```
Bölünen: 7
Bölen: 0
Bölen sıfır olamaz.
```

`b` sıfır olduğu için `a / b` işleminde `ZeroDivisionError` oluştu; `print` çalışmadan ikinci `except` bloğuna geçildi. ■

Alıştırma 4.6 (format ile Sıra Numaraları). Aşağıdaki programın çıktısını bulunuz.

 CC BY-NC-SA 4.0

```
print("{2}-{0}-{1}".format("a", "b", "c"))
print("{0}{0}{1}".format("ha", "!"))
print("{:.1f} ve {:.3f}".format(2 / 3, 2 / 3))
```

Çözüm.
`{2}` üçüncü değeri, `{0}` birinciyi, `{1}` ikinciyi gösterir; birinci satırda `c-a-b` yazılır. İkinci satırda birinci değer iki kez kullanılır ve `haha!` yazılır. Üçüncü satırda $2/3 = 0{,}666\ldots$ önce bir, sonra üç basamakla yuvarlanarak yazılır: `0.7` ve `0.667`.

```
print("{2}-{0}-{1}".format("a", "b", "c"))
print("{0}{0}{1}".format("ha", "!"))
print("{:.1f} ve {:.3f}".format(2 / 3, 2 / 3))
```

Çıktı:

```
c-a-b
haha!
0.7 ve 0.667
```

■

Alıştırma 4.7 (Kullanıcı Bilgi Kartı). Kullanıcıdan adını, yaşını ve boyunu alıp bunları başlıklı bir bilgi kartı olarak yazan bir program yazınız. Boy, virgülden sonra iki basamakla yazılsın.

Çözüm.
Ad metin olarak kalır, yaş `int`'e, boy `float`'a çevrilir. Kartın başlığından önce bir boş satır bırakmak için metnin başına `\n` koyarız:

```
isim = input("Adınızı girin: ")
yas = int(input("Yaşınızı girin: "))
boy = float(input("Boyunuzu girin (örnek: 1.75): "))
print("\nKullanıcı Bilgileri")
print("Ad  :", isim)
print("Yaş :", yas)
print(f"Boy : {boy:.2f} m")
```

Çıktı:

```
Adınızı girin: Zeynep
Yaşınızı girin: 19
Boyunuzu girin (örnek: 1.75): 1.68

Kullanıcı Bilgileri
Ad  : Zeynep
Yaş : 19
Boy : 1.68 m
```

■

Artık programlarımız kullanıcıdan değer alıp sonuçları düzenli bir biçimde gösterebiliyor ve hatalı girdilere karşı kendini koruyabiliyor. Ama her girdiye hep aynı şekilde davranıyorlar. Bir sonraki bölüm Karşılaştırma, Mantık Operatörleri ve Koşullu İfadeler, programın bir koşula göre farklı yollardan ilerlemesini sağlayan karar yapılarını ele alıyor.

 CC BY-NC-SA 4.0

5. Karşılaştırma, Mantık Operatörleri ve Koşullu İfadeler

Şimdiye kadar yazdığımız programlar her çalıştırıldığında aynı komutları aynı sırayla yürüttü. Oysa çoğu problem bir karar gerektirir: girilen sayı negatifse işaretini değiştir, not 50'nin altındaysa "kaldı" yaz, yaş 18'den küçükse "reşit değil" de. Programın bir koşula bağlı olarak farklı yollardan ilerlemesini sağlayan yapılara **koşullu ifadeler** ya da **karar yapıları** denir.
Bir karar vermek için önce bir soru sormak gerekir. Bu bölümde soruları karşılaştırma ve mantık operatörleriyle nasıl soracağımızı, sonra `if`, `else` ve `elif` ile yanıtlara göre nasıl farklı işler yapacağımızı göreceğiz.

5.1 Karşılaştırma Operatörleri

Bir koşul, değeri `True` ya da `False` olan bir ifadedir; bu tür ifadelerin en basitleri iki değeri karşılaştırarak elde edilir.
Değişkenler, Veri Tipleri ve Atama bölümünde mantıksal tip `bool`'u tanımıştık (Tanım 2.6): yalnız `True` ve `False` değerlerini alır ve bir önermenin doğru mu yanlış mı olduğunu kodlar.

Tanım 5.1 (Karşılaştırma Operatörleri). İki değeri karşılaştırıp sonucu `True` ya da `False` olarak veren operatörlere **karşılaştırma operatörleri** denir:

Operatör	Anlamı	Örnek	Sonuç
`<`	küçük mü?	`2 < 3`	`True`
`>`	büyük mü?	`2 > 3`	`False`
`<=`	küçük ya da eşit mi?	`3 <= 3`	`True`
`>=`	büyük ya da eşit mi?	`2 >= 3`	`False`
`==`	eşit mi?	`2 == 3`	`False`
`!=`	eşit değil mi?	`2 != 3`	`True`

Yani `a < b` yazmak Python'a "a, b'den küçük mü?" diye sormaktır; yanıt bir `bool` değeridir ve bir değişkende saklanabilir.

```
a = 2
b = 3
print("a < b: " + str(a < b))
print("a > b: " + str(a > b))
print("a >= b: " + str(a >= b))
print("a <= b: " + str(a <= b))
print("a == b: " + str(a == b))
print("a != b: " + str(a != b))
c = 5 < 3
print(c, type(c))
```

Çıktı:

```
a < b: True
a > b: False
a >= b: False
```

 CC BY-NC-SA 4.0

```
a <= b: True
a == b: False
a != b: True
False <class 'bool'>
```

Bir `bool` değeri `+` ile bir metne eklenemeyeceği için önce `str` ile metne çevirdik; virgülle `print("a < b:", a < b)` yazmak da aynı sonucu verirdi.

> ⚠ `=` ile `==` farklıdır
>
> `=` bir **atama** operatörüdür: sağdaki değeri soldaki değişkene verir. `==` ise bir **karşılaştırma** operatörüdür: iki değerin eşit olup olmadığını sorar ve `True` ya da `False` verir. `if x = 5:` yazmak yazım hatasıdır; doğrusu `if x == 5:`'tir.

Eşitlik operatörü değerlerin tipini de hesaba katar: metin olan `"2"` ile tam sayı olan `2` eşit değildir. Buna karşılık `2 == 2.0` `True`'dur, çünkü iki sayı aynı değeri gösterir.

```
print("2" == 2)
print("2" != 2)
print(2 == 2.0)
print("Evet" == "Evet", "Evet" == "evet")
```

Çıktı:

```
False
True
True
True False
```

Metinler karakter karakter karşılaştırılır; büyük ve küçük harf farklı karakterlerdir, bu yüzden `"Evet"` ile `"evet"` eşit değildir.

Tanım 5.2 (Zincirleme Karşılaştırma). `a < x < b` biçimindeki yazım `a < x and x < b` anlamına gelir; buna **zincirleme karşılaştırma** denir. Diğer karşılaştırma operatörleri de aynı biçimde zincirlenebilir.

Yani matematikteki $13 \leq y < 20$ yazımı Python'da aynen `13 <= y < 20` olarak yazılabilir.

5.2 Mantık Operatörleri

Birden fazla koşulu birleştirmek ya da bir koşulu tersine çevirmek için mantık operatörleri kullanılır.

Tanım 5.3 (and Operatörü). `p and q` ifadesi, `p` ve `q`'nun **ikisi de** `True` ise `True`, en az biri `False` ise `False` olur.

Yani `and` "ve" bağlacıdır: "sayı 5'ten büyük **ve** 10'dan küçük" koşulu, iki koşulun ikisi de sağlandığında doğrudur.

Tanım 5.4 (or Operatörü). `p or q` ifadesi, `p` ve `q`'dan **en az biri** `True` ise `True`, ikisi de `False` ise `False` olur.

Yani `or` "veya" bağlacıdır: "sayı 10 **veya** 20" koşulu, sayı bu iki değerden birine eşitse doğrudur.

Tanım 5.5 (not Operatörü). `not p` ifadesi, `p` `True` ise `False`, `False` ise `True` olur.

Yani `not` bir koşulu tersine çevirir: `not x < 5` ifadesi "x, 5'ten küçük değil" demektir ve `x >= 5` ile aynı anlama gelir.
Üç operatörün bütün durumları bir **doğruluk tablosunda** özetlenir:

 CC BY-NC-SA 4.0

p	q	p and q	p or q	not p
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

```
a = True
b = False
print(True and True, True and False, False and False)
print(1 == 1 and 1 < 2)
print(1 < 2 and 3 < 1)
print(a or b, 1 < 2 or 3 < 1)
print(not True, not 1 < 2, not b)
```

Çıktı:

```
True False False
True
False
True True
False False True
```

Mantık operatörlerinin kendi aralarında da bir öncelik sırası vardır: önce `not`, sonra `and`, en son `or` uygulanır; karşılaştırmalar ise üçünden de önce yapılır. Bu yüzden `not 1 < 2` ifadesi `not (1 < 2)` olarak okunur. Öncelikten emin olunmayan yerlerde parantez kullanmak hem hatayı önler hem de okumayı kolaylaştırır.

Örnek 5.1 (Mantık Operatörlerinin Önceliği). `True or False and False` ifadesinin değeri nedir? `(True or False) and False` ile aynı mıdır?

> **Çözüm.**
> `and`, `or`'dan önce uygulanır; ifade `True or (False and False)` olarak okunur. Parantez içi `False`'tur ve `True or False` sonucu `True` olur.
> İkinci ifadede parantez önce `or`'u yaptırır: `True or False` `True` olur ve `True and False` sonucu `False`'tur. İki ifade farklıdır.
>
> ```
> print(True or False and False)
> print((True or False) and False)
> ```
>
> Çıktı:
>
> ```
> True
> False
> ```
>
> ■

5.2.1 Kısa Devre ve Mantıksal Olmayan Değerler

`and` ve `or` yalnız `True` ve `False` ile değil, her türlü değerle kullanılabilir; bu durumda verdikleri sonuç biraz şaşırtıcı olabilir.

Tanım 5.6 (Kısa Devre Değerlendirmesi). `p or q` ifadesinde Python önce `p`'ye bakar: `p` doğru sayılan bir değerse sonucu `p` olarak verir ve `q`'ya hiç bakmaz; değilse sonucu `q` olarak verir. `p and q` ifadesinde `p` yanlış sayılan bir değerse sonuç `p`'dir, değilse `q`'dur. Sonucun ilk değere bakılarak belirlenip ikinci değere geçilmemesine **kısa devre** (short-circuit) denir.

 CC BY-NC-SA 4.0

Yani `or` ve `and` her zaman iki değerden birini olduğu gibi verir; bu değer `True` ya da `False` olmak zorunda değildir. Bir değerin doğru ya da yanlış sayılması, onun `bool` ile dönüştürülmüş hâline bakılarak belirlenir: `0`, `0.0` ve `""` yanlış, diğerleri doğru sayılır.

```
print(1 or 2)
print(0 or 2)
print(1 and 2)
print(0 and 2)
print("" or "boş metin")
```

Çıktı:

```
1
2
2
0
boş metin
```

`1 or 2` ifadesinde `1` doğru sayıldığı için sonuç `1`'dir; Python `2`'ye hiç bakmaz. `0 and 2` ifadesinde `0` yanlış sayıldığı için sonuç `0`'dır.
Kısa devrenin pratik bir faydası vardır: `b != 0 and a / b > 1` koşulunda `b` sıfırsa `and` ikinci kısma geçmez, bu yüzden sıfıra bölme hatası oluşmaz.

5.3 if Yapısı

Bir koşul doğru olduğunda belli komutları çalıştırmak için `if` yapısı kullanılır.

Tanım 5.7 (if Yapısı).

```
if koşul:
    koşul doğruysa çalışacak komutlar
```

yapısında `koşul` `True` ise girintili komutlar çalışır; `False` ise atlanır ve program `if` bloğundan sonraki ilk girintisiz satırdan devam eder. Koşuldan sonra iki nokta (`:`) konur ve bloğa ait komutlar aynı miktarda girintiyle yazılır.

Yani `if` bir "eğer" cümlesidir: "eğer sayı 5'ten büyükse şunu yaz". Python bloğun nerede bittiğini girintilerden anlar; Girdi ve Çıktı İşlemleri bölümündeki `try-except` yapısında olduğu gibi girinti burada da anlamın bir parçasıdır.

```
sayi = 10
if sayi > 5:
    print("Sayı 5'ten büyüktür.")
    print("Bu satır da if bloğunun içinde.")
print("Bu satır her durumda çalışır.")
```

Çıktı:

```
Sayı 5'ten büyüktür.
Bu satır da if bloğunun içinde.
Bu satır her durumda çalışır.
```

Aynı program `sayi = 3` ile çalıştırılsaydı koşul `False` olur, girintili iki satır atlanır ve yalnız son satır yazılırdı.

5.3.1 else

Koşul sağlanmadığında da bir şey yapılması isteniyorsa `if`'e bir `else` bloğu eklenir.

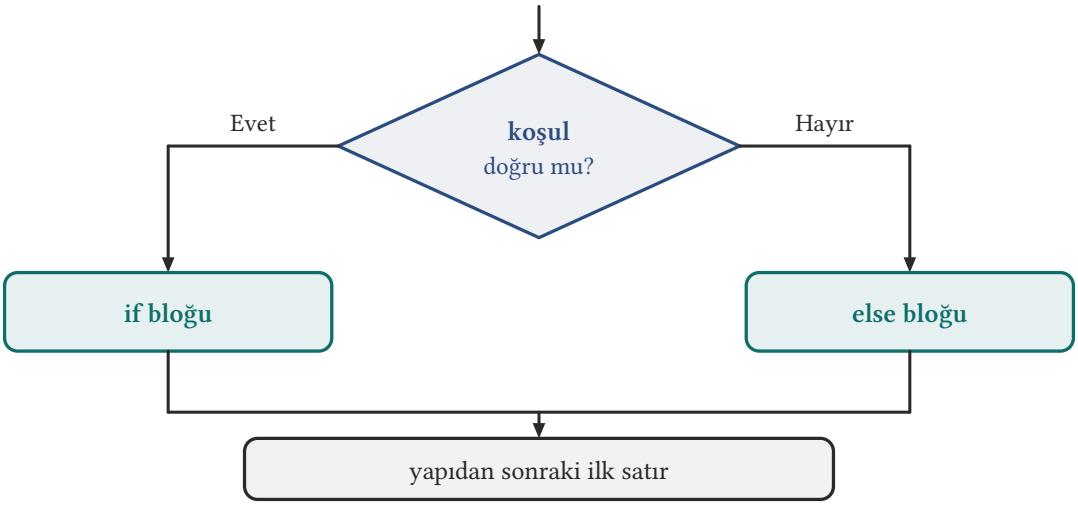
Tanım 5.8 (if-else Yapısı).

 CC BY-NC-SA 4.0

```
if koşul:
    koşul doğruysa çalışacak komutlar
else:
    koşul yanlışsa çalışacak komutlar
```

yapısında iki bloktan **tam olarak biri** çalışır. `else`'in kendine ait bir koşulu yoktur; `if` koşulu sağlanmadığında kendiliğinden çalışır.

Yani `if-else` iki yol ayrımıdır: program ya birinci yoldan ya ikinci yoldan gider, ikisinden birden asla gitmez.



Şekil 5.1: Bir `if-else` yapısının akışı. Koşul doğruysa yalnız `if` bloğu, yanlışsa yalnız `else` bloğu çalışır; iki yol da yapıdan sonraki satırda birleşir.

```
sayi = 3
if sayi > 5:
    print("Sayı 5'ten büyüktür.")
else:
    print("Sayı 5'ten küçük ya da 5'e eşittir.")
```

Çıktı:

```
Sayı 5'ten küçük ya da 5'e eşittir.
```

5.3.2 elif

İkiden fazla durum arasında seçim yapmak için `elif` (else if, "değilse eğer") kullanılır.

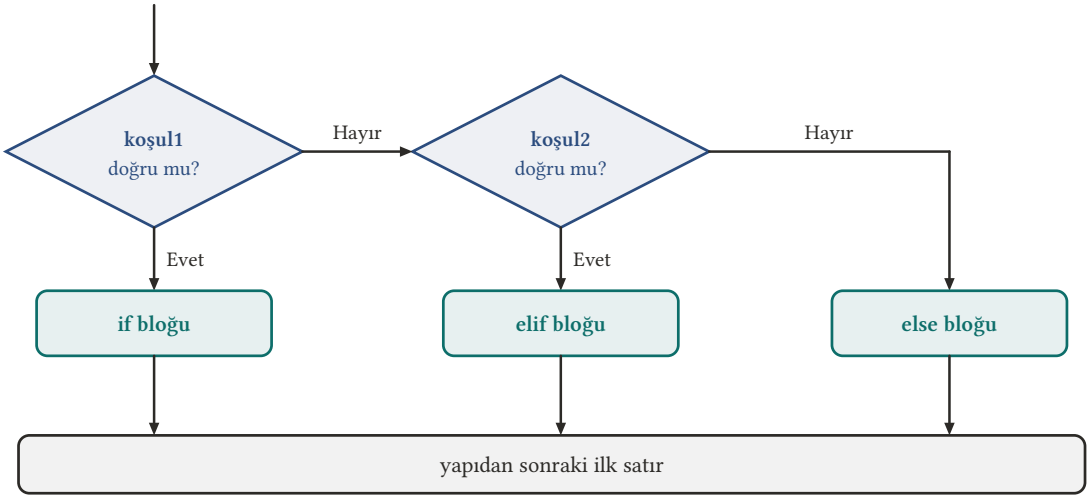
Tanım 5.9 (if-elif-else Yapısı).

```
if koşul1:
    koşul1 doğruysa çalışacak komutlar
elif koşul2:
    koşul1 yanlış, koşul2 doğruysa çalışacak komutlar
else:
    bütün koşullar yanlışsa çalışacak komutlar
```

 CC BY-NC-SA 4.0

yapısında koşullar yukarıdan aşağıya sırayla denenir ve **doğru çıkan ilk koşulun** bloğu çalışır; geri kalan koşullara bakılmaz. Hiçbir koşul doğru değilse `else` bloğu çalışır. İstenildiği kadar `elif` yazılabilir; `else` isteğe bağlıdır.

Yani `if-elif-else` çok kollu bir yol ayrımıdır ve en fazla bir kol seçilir. Koşulların yazılış sırası bu yüzden önemlidir.



Şekil 5.2: Bir `if-elif-else` yapısının akışı. Koşullar yazıldıkları sırayla denenir; doğru çıkan ilk koşulun bloğu çalışır ve geri kalanlara bakılmaz. Hiçbir koşul doğru değilse `else` bloğu çalışır.

```
sayi = 13
if sayi > 10:
    print("Sayı 10'dan büyüktür.")
elif sayi > 5:
    print("Sayı 5'ten büyüktür.")
elif sayi == 5:
    print("Sayı 5'e eşittir.")
else:
    print("Sayı 5'ten küçüktür.")
```

Çıktı:

```
Sayı 10'dan büyüktür.
```

13 hem 10'dan hem 5'ten büyüktür; ama ilk koşul doğru çıktığı için yalnız ilk mesaj yazıldı ve `elif sayi > 5` koşuluna hiç bakılmadı.

Örnek 5.2 (Koşulların Sırası). Bir notun harf karşılığı şöyle belirlensin: 85 ve üstü A, 70 ile 84 arası B, 50 ile 69 arası C, 50'nin altı D. Aşağıdaki program 90 notu için neden yanlış sonuç verir? Nasıl düzeltilir?

```
not_ = 90
if not_ >= 50:
    print("C")
elif not_ >= 70:
    print("B")
elif not_ >= 85:
    print("A")
else:
    print("D")
```

 CC BY-NC-SA 4.0

Çözüm.
$90 \geq 50$ olduğu için ilk koşul doğru çıkar ve program `C` yazar; `elif`'lere hiç bakılmaz:

```
not_ = 90
if not_ >= 50:
    print("C")
elif not_ >= 70:
    print("B")
elif not_ >= 85:
    print("A")
else:
    print("D")
```

Çıktı:

```
C
```

Hata, en geniş koşulun en başa yazılmasıdır: 50'den büyük her not ilk kola düşer. Koşulları en dar olandan başlayarak yazmak gerekir. Bir `elif`'e ulaşıldıysa önceki koşulların hepsi yanlış demektir; bu yüzden `elif not_ >= 70` satırına gelen bir notun 85'ten küçük olduğu zaten bilinir ve ayrıca `not_ < 85` yazmaya gerek yoktur:

```
not_ = 90
if not_ >= 85:
    print("A")
elif not_ >= 70:
    print("B")
elif not_ >= 50:
    print("C")
else:
    print("D")
```

Çıktı:

```
A
```

■

5.4 Koşullarda Mantık Operatörleri

Bir karar birden fazla koşula bağlıysa koşullar `and`, `or` ve `not` ile birleştirilir.

```
sayi = 15
if sayi > 5 and sayi < 10:
    print("Sayı 5 ile 10 arasındadır.")
elif sayi == 10 or sayi == 20:
    print("Sayı 10 ya da 20'dir.")
elif not sayi < 5:
    print("Sayı 5'ten küçük değildir.")
```

Çıktı:

```
Sayı 5'ten küçük değildir.
```

15 için birinci koşul (`15 < 10` yanlış olduğundan) ve ikinci koşul sağlanmaz; üçüncü koşulda `15 < 5` yanlış olduğundan `not` onu doğruya çevirir ve son mesaj yazılır. Birinci koşul zincirleme karşılaştırmayla (Tanım 5.2) `5 < sayi < 10` olarak da yazılabilirdi.

 CC BY-NC-SA 4.0

5.5 İç İçe Koşullar

Bir koşulun bloğunun içine başka bir koşul yazılabilir; böylece ikinci soru yalnız birinci sorunun yanıtı belli bir yönde çıktığında sorulur.

```
sayi = 15
if sayi > 10:
    print("Sayı 10'dan büyüktür.")
    if sayi < 20:
        print("Sayı 20'den küçüktür.")
    else:
        print("Sayı 20'ye eşit ya da 20'den büyüktür.")
else:
    print("Sayı 10 ya da daha küçüktür.")
```

Çıktı:

```
Sayı 10'dan büyüktür.
Sayı 20'den küçüktür.
```

İçteki `if-else` bir girinti daha içeride yazılır ve yalnız dıştaki koşul doğruysa çalışır. Hangi `else`'in hangi `if`'e ait olduğunu girintiler belirler: içteki `else` içteki `if` ile, dıştaki `else` dıştaki `if` ile aynı hizadadır.

5.6 Tek Satırda Koşul

Bir değişkene bir koşula göre iki değerden birini atamak için `if-else` yapısının tek satırlık bir yazımı vardır.

Tanım 5.10 (Koşullu İfade). `a if koşul else b` ifadesinin değeri, `koşul` `True` ise `a`, `False` ise `b`'dir. Bu yazıma **koşullu ifade** ya da üç parçası olduğu için **üçlü operatör** (ternary operator) denir.

Yani `sonuc = "Pozitif" if sayi > 0 else "Pozitif değil"` satırı dört satırlık bir `if-else` yapısının yaptığı işi tek satırda yapar. Koşullu ifade bir komut değil bir değerdir; bu yüzden bir atamanın sağ tarafında ya da bir `print`'in içinde kullanılabilir.

```
sayi = 6
sonuc = "Pozitif" if sayi > 0 else "Negatif ya da sıfır"
print(sonuc)
print("çift" if sayi % 2 == 0 else "tek")
```

Çıktı:

```
Pozitif
çift
```

Koşullu ifade, parçalı tanımlı fonksiyonları hesaplamak için de kullanışlıdır.

Örnek 5.3 (Parçalı Tanımlı Bir Fonksiyon).

$$f(x) = \begin{cases} x^2 + 1, & x \geq 0, \\ -x^2 + 1, & x < 0 \end{cases}$$

fonksiyonunun $x = 3$ ve $x = -2$ noktalarındaki değerlerini koşullu ifadeyle hesaplayınız.

> **Çözüm.**
> Tanımın iki kolunu koşullu ifadenin iki tarafına yazarız. $x = 3 \geq 0$ için birinci kol kullanılır: $f(3) = 9 + 1 = 10$. $x = -2 < 0$ için ikinci kol kullanılır: $f(-2) = -4 + 1 = -3$.

```
x = 3
f = x ** 2 + 1 if x >= 0 else -x ** 2 + 1
print(f)
x = -2
```

 CC BY-NC-SA 4.0

```
f = x ** 2 + 1 if x >= 0 else -x ** 2 + 1
print(f)
```

Çıktı:

```
10
-3
```

`-x ** 2` ifadesinde üs alma eksi işaretinden önce yapıldığı için $-(x^2)$ hesaplanır; $x = -2$ için bu -4'tür. ■

5.7 pass İfadesi

Python'da bir bloğun içi boş bırakılamaz; henüz yazılmamış bir blok için yer tutucu olarak `pass` kullanılır.

Tanım 5.11 (pass İfadesi). `pass` hiçbir şey yapmayan bir komuttur. Yazım kuralları gereği bir komut bulunması gereken ama yapılacak bir iş olmayan yerlerde kullanılır.

Yani `pass`, "burayı sonra dolduracağım" demenin yoludur. Bir `if` bloğu tamamen boş bırakılırsa Python `IndentationError` verir; içine `pass` yazılınca program sorunsuz çalışır.

```
sayi = 25
if sayi > 5:
    pass  # burası daha sonra yazılacak
else:
    print("Sayı 5 ya da daha küçük.")
print("Program bitti.")
```

Çıktı:

```
Program bitti.
```

5.8 Çeşitli Örnekler

Bu bölümde öğrendiklerimizi kullanıcıdan değer alan birkaç küçük programda birleştirelim.

Örnek 5.4 (Mutlak Değer). Kullanıcıdan bir sayı alıp mutlak değerini hazır `abs` fonksiyonunu kullanmadan hesaplayan bir program yazınız.

Çözüm.
Mutlak değerin tanımı $|x| = x$ ($x \geq 0$ ise) ve $|x| = -x$ ($x < 0$ ise) biçimindedir. Önce sonucu sayının kendisine eşitleriz; sayı negatifse işaretini değiştiririz. Kullanıcı -9 girsin:

```
sayi = float(input("Bir sayı giriniz: "))
mutlak = sayi
if sayi < 0:
    mutlak = -sayi
print(f"{sayi} sayısının mutlak değeri {mutlak}")
```

Çıktı:

```
Bir sayı giriniz: -9
-9.0 sayısının mutlak değeri 9.0
```

Burada `else` bloğuna gerek yoktur: sayı negatif değilse `mutlak` zaten doğru değeri taşır. ■

 CC BY-NC-SA 4.0

Örnek 5.5 (Yaşa Göre Kategori). Kullanıcının girdiği yaş 13'ten küçükse "Çocuk", 13–17 arasındaysa "Genç", 18–64 arasındaysa "Yetişkin", 65 ve üstündeyse "Yaşlı" yazan bir program yazınız.

Çözüm.
Dört kategori olduğundan bir `if`, iki `elif` ve bir `else` kullanırız. Koşullar sırayla denendiği için `elif yas < 18` satırına ulaşan bir yaşın 13 ya da daha büyük olduğu zaten bilinir. Kullanıcı 15 girsin:

```
yas = int(input("Yaşınızı girin: "))
if yas < 13:
    print("Çocuk")
elif yas < 18:
    print("Genç")
elif yas < 65:
    print("Yetişkin")
else:
    print("Yaşlı")
```

Çıktı:

```
Yaşınızı girin: 15
Genç
```

■

Örnek 5.6 (Pozitif, Negatif ya da Sıfır). Kullanıcının girdiği bir tam sayının pozitif, negatif ya da sıfır olduğunu yazan bir program yazınız. Üç ayrı `if` kullanan bir çözüm ile `if-elif-else` kullanan bir çözümü karşılaştırınız.

Çözüm.
Üç durumu üç ayrı `if` ile denetleyebiliriz. Kullanıcı 52 girsin:

```
a = int(input("Bir sayı giriniz: "))
if a > 0:
    print("Pozitif")
if a == 0:
    print("Sıfır")
if a < 0:
    print("Negatif")
```

Çıktı:

```
Bir sayı giriniz: 52
Pozitif
```

Bu program doğru sonuç verir, ama sayı pozitif bulunduktan sonra da diğer iki koşulu boşuna sınar. Üç durum birbirini dışladığı için `if-elif-else` daha uygundur: doğru koşul bulunur bulunmaz geri kalanlara bakılmaz, son durum için de koşul yazmaya gerek kalmaz.

```
a = int(input("Bir sayı giriniz: "))
if a > 0:
    print("Pozitif")
elif a == 0:
    print("Sıfır")
else:
    print("Negatif")
```

 CC BY-NC-SA 4.0

Çıktı:

```
Bir sayı giriniz: 52
Pozitif
```

■

5.9 Alıştırmalar

Girdi alan alıştırmalarda farklı değerlerle programın bütün kollarını deneyin.

Alıştırma 5.1 (Çift mi, Tek mi?). Kullanıcının girdiği bir tam sayının çift mi tek mi olduğunu yazan bir program yazınız.

Çözüm.
Bir sayı ancak ve ancak 2'ye bölümünden kalan 0 ise çifttir. Kullanıcı 17 girsin; $17 = 8 \cdot 2 + 1$ olduğundan kalan 1'dir ve sayı tektir.

```
n = int(input("Bir tam sayı girin: "))
if n % 2 == 0:
    print(n, "çift bir sayıdır.")
else:
    print(n, "tek bir sayıdır.")
```

Çıktı:

```
Bir tam sayı girin: 17
17 tek bir sayıdır.
```

■

Alıştırma 5.2 (Üç Sayının En Büyüğü). `a = 7`, `b = 12` ve `c = 9` sayılarının en büyüğünü hazır `max` fonksiyonunu kullanmadan `if-elif-else` ile bulunuz.

Çözüm.
`a` diğer ikisinden büyük ya da onlara eşitse en büyük `a`'dır. Değilse en büyük ya `b` ya `c`'dir; bunu `b >= c` sorusu belirler.

```
a = 7
b = 12
c = 9
if a >= b and a >= c:
    en_buyuk = a
elif b >= c:
    en_buyuk = b
else:
    en_buyuk = c
print("En büyük:", en_buyuk)
```

Çıktı:

```
En büyük: 12
```

`elif b >= c` satırına ulaşıldıysa `a`'nın en büyük olmadığı bilinir; geriye `b` ile `c`'yi karşılaştırmak kalır. ■

 CC BY-NC-SA 4.0

Alıştırma 5.3 (Artık Yıl). Bir yıl 4'e bölünüyorsa artık yıldır; ama 100'e bölünüp 400'e bölünmeyen yıllar artık yıl değildir. Kullanıcının girdiği yılın artık yıl olup olmadığını yazan bir program yazınız ve 1900 için deneyiniz.

Çözüm.
Kural tek bir koşulla yazılabilir: yıl 400'e bölünüyorsa artık yıldır; ya da yıl 4'e bölünüyor **ve** 100'e bölünmüyorsa artık yıldır. 1900 yılı 4'e ve 100'e bölünür ama 400'e bölünmez ($1900 = 4 \cdot 400 + 300$); bu yüzden artık yıl değildir.

```
yil = int(input("Yıl: "))
if yil % 400 == 0 or (yil % 4 == 0 and yil % 100 != 0):
    print(yil, "artık yıldır.")
else:
    print(yil, "artık yıl değildir.")
```

Çıktı:

```
Yıl: 1900
1900 artık yıl değildir.
```

Parantez burada zorunlu değildir, çünkü `and` zaten `or`'dan önce uygulanır; ama koşulun yapısını açıkça gösterdiği için yazılması iyidir. 2000 yılı ise 400'e bölündüğü için artık yıldır. ■

Alıştırma 5.4 (İkinci Dereceden Denklemin Kök Sayısı). $ax^2 + bx + c = 0$ ($a \neq 0$) denkleminin kaç gerçel kökü olduğunu diskriminant $\Delta = b^2 - 4ac$'nin işaretine göre bulan bir program yazınız. $x^2 - 6x + 9 = 0$ için deneyiniz.

Çözüm.
$\Delta > 0$ ise iki farklı gerçel kök, $\Delta = 0$ ise bir (çift) kök vardır, $\Delta < 0$ ise gerçel kök yoktur. $a = 1$, $b = -6$, $c = 9$ için $\Delta = 36 - 36 = 0$ olur; denklemin tek kökü $x = -b/(2a) = 3$'tür.

```
a = 1
b = -6
c = 9
delta = b ** 2 - 4 * a * c
if delta > 0:
    print("İki farklı gerçel kök var.")
elif delta == 0:
    print("Bir (çift) kök var:", -b / (2 * a))
else:
    print("Gerçel kök yok.")
```

Çıktı:

```
Bir (çift) kök var: 3.0
```

■

Alıştırma 5.5 (Üçgen Olabilir mi?). Kenar uzunlukları 3, 4 ve 8 olan bir üçgen var mıdır? Üç kenar uzunluğu verildiğinde bunu söyleyen bir program yazınız.

Çözüm.
Üç pozitif uzunluğun bir üçgenin kenarları olması için her kenar diğer ikisinin toplamından küçük olmalıdır (üçgen eşitsizliği). Üç koşulun üçü de sağlanmalı, bu yüzden `and` ile bağlanır. $3 + 4 = 7 < 8$ olduğundan $a + b > c$ koşulu sağlanmaz ve üçgen yoktur.

 CC BY-NC-SA 4.0

```
a = 3
b = 4
c = 8
if a + b > c and a + c > b and b + c > a:
    print("Bu uzunluklarla bir üçgen çizilebilir.")
else:
    print("Bu uzunluklarla üçgen çizilemez.")
```

Çıktı:

```
Bu uzunluklarla üçgen çizilemez.
```

■

Alıştırma 5.6 (Vücut Kitle İndeksi). Vücut kitle indeksi $\text{VKİ} = \dfrac{\text{kilo}}{\text{boy}^2}$ ile hesaplanır (kilo kilogram, boy metre cinsinden). VKİ 18,5'ten küçükse "zayıf", 18,5 ile 25 arasındaysa "normal", 25 ile 30 arasındaysa "fazla kilolu", 30 ve üstündeyse "obez" yazan bir program yazınız. Kullanıcı kilo olarak 70, boy olarak 1,75 girsin.

Çözüm.

$$\text{VKİ} = 70/1{,}75^2 = 70/3{,}0625 \approx 22{,}86$$

olur; bu değer 18,5 ile 25 arasında olduğundan "normal" kategorisindedir. Kategoriler küçükten büyüğe sıralı olduğu için koşulları da bu sırayla yazarız:

```
kilo = float(input("Kilo (kg): "))
boy = float(input("Boy (m): "))
vki = kilo / boy ** 2
if vki < 18.5:
    kategori = "zayıf"
elif vki < 25:
    kategori = "normal"
elif vki < 30:
    kategori = "fazla kilolu"
else:
    kategori = "obez"
print(f"VKİ = {vki:.2f}, kategori: {kategori}")
```

Çıktı:

```
Kilo (kg): 70
Boy (m): 1.75
VKİ = 22.86, kategori: normal
```

■

Alıştırma 5.7 (İki Koşulun Eşitliği). Aşağıdaki programın çıktısını bulunuz.

```
if (2 > 1) == (5 < 10):
    print("İfadeler aynıdır.")
if (2 > 1) == (5 > 10):
    print("İfadeler aynıdır.")
```

```
else:
    print("İfadeler farklıdır.")
```

Çözüm.
Parantezler önce hesaplanır. Birinci `if`'te `2 > 1` ve `5 < 10` ikisi de `True`'dur; `True == True` doğru olduğundan ilk mesaj yazılır. İkinci `if`'te `2 > 1` `True`, `5 > 10` `False`'tur; `True == False` yanlış olduğundan `else` bloğu çalışır.

```
if (2 > 1) == (5 < 10):
    print("İfadeler aynıdır.")
if (2 > 1) == (5 > 10):
    print("İfadeler aynıdır.")
else:
    print("İfadeler farklıdır.")
```

Çıktı:

```
İfadeler aynıdır.
İfadeler farklıdır.
```

İki `if` birbirinden bağımsızdır; `else` yalnız ikinci `if`'e aittir. ■

Programlarımız artık karar verebiliyor. Ama her karar tek bir değer üzerinde ve yalnız bir kez veriliyor. Bir sonraki bölüm Koleksiyonlar ve Listeler, birçok değeri tek bir değişkende bir arada tutan veri yapılarını tanıtıyor; ardından gelen bölümlerde bu değerlerin her biri üzerinde döngülerle işlem yapacağız.

 CC BY-NC-SA 4.0

6. Koleksiyonlar ve Listeler

Bir sınıftaki otuz öğrencinin notlarını saklamak için otuz ayrı değişken tanımlamak hem zahmetli hem de kullanışsızdır: notların ortalamasını almak ya da en yükseğini bulmak için otuz değişkeni tek tek yazmak gerekirdi. Bunun yerine bütün notları tek bir değişkende, sıralı bir dizi olarak tutmak isteriz.
Birden çok değeri tek bir değişkende bir arada tutan veri tiplerine **koleksiyon** denir. Bu bölümde Python'ın dört temel koleksiyonunu tanıyacağız, sonra en çok kullanılanı olan **listeler** üzerinde ayrıntılı olarak duracağız: elemanlarına erişmeyi, liste parçaları almayı, eleman eklemeyi, silmeyi ve sıralamayı göreceğiz.

6.1 Dört Temel Koleksiyon

Python'da birden çok değeri bir arada tutmak için dört temel koleksiyon vardır; aralarındaki fark, elemanların sıralı olup olmadığı, değiştirilip değiştirilemeyeceği ve tekrar edip edemeyeceğidir.

Tanım 6.1 (Liste). Köşeli parantez içinde virgülle ayrılarak yazılan `[e1, e2, ...]` koleksiyonuna **liste** (list) denir. Listenin elemanları sıralıdır ve sıra numarasıyla erişilir; liste oluşturulduktan sonra elemanları değiştirilebilir, eleman eklenip çıkarılabilir; aynı eleman birden çok kez bulunabilir.

Yani liste, sırası önemli ve içeriği zamanla değişebilen bir değerler dizisidir: bir alışveriş listesi gibi.

Tanım 6.2 (Demet). Normal parantez içinde yazılan `(e1, e2, ...)` koleksiyonuna **demet** (tuple) denir. Demetin elemanları listedeki gibi sıralıdır ve sıra numarasıyla erişilir, aynı eleman birden çok kez bulunabilir; ama demet oluşturulduktan sonra **değiştirilemez**.

Yani demet, değişmeyecek değerler için kullanılan "salt okunur" bir listedir: bir noktanın koordinatları ya da bir tarihin gün, ay ve yılı gibi. Demetleri Demetler bölümünde ayrıntılı göreceğiz.

Tanım 6.3 (Küme). Süslü parantez içinde yazılan `{e1, e2, ...}` koleksiyonuna **küme** (set) denir. Kümede aynı eleman en fazla bir kez bulunur; elemanların bir sırası yoktur, bu yüzden sıra numarasıyla erişilemez. Kümeye eleman eklenip çıkarılabilir.

Yani Python'daki küme matematikteki kümeye karşılık gelir: $\{1, 2, 2, 3\} = \{1, 2, 3\}$ olduğu gibi, Python da tekrar eden elemanları tek bir eleman olarak tutar.

Tanım 6.4 (Sözlük). Süslü parantez içinde `{anahtar1: değer1, anahtar2: değer2, ...}` biçiminde yazılan koleksiyona **sözlük** (dictionary, dict) denir. Sözlükte her değere sıra numarasıyla değil, **anahtarıyla** erişilir: `sozluk[anahtar]`. Bir anahtar sözlükte en fazla bir kez bulunur; değerler değiştirilebilir, yeni anahtar–değer çiftleri eklenebilir.

Yani sözlük, gerçek bir sözlük gibi çalışır: bir sözcüğe (anahtara) bakılır ve karşısındaki açıklama (değer) bulunur.

Koleksiyon	Yazım	Sıralı	Değiştirilebilir	Tekrar eden eleman
liste	`[1, 2, 2]`	evet	evet	olabilir
demet	`(1, 2, 2)`	evet	hayır	olabilir
küme	`{1, 2}`	hayır	evet	olamaz
sözlük	`{"a": 1}`	eklenme sırasıyla	evet	anahtar tekrar edemez

 CC BY-NC-SA 4.0

```
bir_liste = ["elma", "armut", "kiraz", "kiraz"]
bir_demet = ("elma", "armut", "kiraz", "kiraz")
bir_kume = {"elma", "armut", "kiraz", "kiraz"}
bir_sozluk = {"tür": "elma", "kg": 32}
print(bir_liste)
print(bir_demet)
print(len(bir_kume))
print(bir_sozluk["tür"], bir_sozluk["kg"])
```

Çıktı:

```
['elma', 'armut', 'kiraz', 'kiraz']
('elma', 'armut', 'kiraz', 'kiraz')
3
elma 32
```

`len` fonksiyonu bir koleksiyondaki eleman sayısını verir. Kümeye `"kiraz"` iki kez yazıldığı hâlde küme yalnız üç eleman tutar. Kümenin elemanları sırasız olduğu için kümeyi yazdırdığımızda elemanların hangi sırayla görüneceği belli değildir; bu yüzden burada kümeyi değil uzunluğunu yazdırdık.
Kümelere `add` ile eleman eklenir; zaten bulunan bir eleman eklenmeye çalışılırsa küme değişmez:

```
kume = {3, 7, 9, 1, 8}
kume.add(6)
print(len(kume), 6 in kume)
kume.add(3)
print(len(kume))
```

Çıktı:

```
6 True
6
```

`6 in kume` ifadesi "6 kümenin elemanı mı?" sorusudur; `in` operatörünü aşağıda listelerle birlikte göreceğiz.

6.1.1 Koleksiyonlar Arası Dönüşüm

Bir koleksiyon, tipin adını taşıyan fonksiyonla başka bir koleksiyon tipine çevrilebilir.
`list(x)`, `tuple(x)` ve `set(x)` fonksiyonları `x` koleksiyonunun elemanlarından sırasıyla bir liste, bir demet ve bir küme oluşturur. `dict(x)` ise ikili demetlerden oluşan bir listeyi sözlüğe çevirir: her demetin birinci elemanı anahtar, ikinci elemanı değer olur. Bir metin de bir karakter dizisi olduğu için `list("abc")` her karakteri ayrı bir eleman yapar.

```
sayilar = (1, 2, 3)
print(type(sayilar))
sayi_listesi = list(sayilar)
print(sayi_listesi, type(sayi_listesi))
print(tuple([1, 2, 3]))
print(set([1, 2, 2, 3]))
ciftler = [("isim", "Ali"), ("yas", 30)]
print(dict(ciftler))
print(list("Ali"))
```

Çıktı:

```
<class 'tuple'>
[1, 2, 3] <class 'list'>
(1, 2, 3)
{1, 2, 3}
{'isim': 'Ali', 'yas': 30}
['A', 'l', 'i']
```

`set([1, 2, 2, 3])` dönüşümü tekrar eden elemanları atar; bir listedeki farklı elemanları bulmanın kısa bir yoludur.

 CC BY-NC-SA 4.0

6.1.2 Metotlar ve dir Fonksiyonu

Koleksiyonların her birinin, yalnız o tipe ait işlemleri yapan kendi fonksiyonları vardır.

Tanım 6.5 (Metot). Belli bir tipteki bir değere ait olan ve `deger.metot_adi(...)` biçiminde, değerin adından sonra bir nokta konarak çağrılan fonksiyona **metot** (method) denir.

Yani `kume.add(6)` yazımında `add`, kümelere ait bir metottur ve `kume` adlı kümeye uygulanır. `len(kume)` ise bir metot değil, her koleksiyonla kullanılabilen genel bir fonksiyondur.
Bir tipin metotları `dir` ile listelenir. Listenin başındaki alt çizgili adları ayıklamak için aşağıdaki satırı kullanabiliriz; bu satırın nasıl çalıştığını for Döngüsü ve Liste Üreteçleri bölümünde anlayacağız.

```
print([ad for ad in dir(list) if not ad.startswith("_")])
print([ad for ad in dir(tuple) if not ad.startswith("_")])
print([ad for ad in dir(dict) if not ad.startswith("_")])
```

Çıktı:

```
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
'reverse', 'sort']
['count', 'index']
['clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault',
'update', 'values']
```

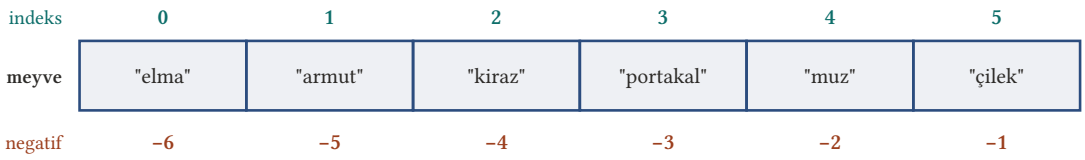
Demetlerin yalnız iki metodu vardır: `count` ve `index`. Bu, demetlerin değiştirilemez olmasının bir sonucudur; eleman ekleyen ya da silen bir metot demetlerde bulunmaz.

6.2 Listelerde İndeksleme

Bir listenin her elemanının listedeki yerini gösteren bir sıra numarası vardır; elemanlara bu numarayla erişilir.

Tanım 6.6 (İndeks). Bir listede bir elemanın sıra numarasına o elemanın **indeksi** denir. İndeksler 0'dan başlar: n elemanlı bir listede ilk elemanın indeksi 0, son elemanın indeksi $n-1$'dir. Listenin sonundan saymak için **negatif indeksler** kullanılır: son elemanın indeksi -1, sondan ikincininki -2'dir. `liste[i]` yazımı `i` indeksli elemanı verir.

Yani `meyve[0]` listenin birinci, `meyve[-1]` son elemanıdır. n elemanlı bir listede `liste[-k]` ile `liste[n - k]` aynı elemandır.



Şekil 6.1: Altı elemanlı `meyve` listesinin indeksleri. Üstteki indeksler baştan 0'dan başlayarak, alttaki negatif indeksler sondan −1'den başlayarak sayılır; örneğin `meyve[2]` ve `meyve[-4]` aynı eleman olan `"kiraz"`'dır.

```
meyve = ["elma", "armut", "kiraz", "portakal", "muz", "çilek"]
print(meyve[0])
print(meyve[2])
print(meyve[-1])
print(meyve[-2])
print(len(meyve))
```

Çıktı:

 CC BY-NC-SA 4.0

```
elma
kiraz
çilek
muz
6
```

Listede olmayan bir indekse erişilmeye çalışılırsa `IndexError` oluşur. Altı elemanlı bir listede geçerli indeksler 0'dan 5'e ve -1'den -6'ya kadardır:

```
meyve = ["elma", "armut", "kiraz", "portakal", "muz", "çilek"]
print(meyve[6])
```

Çıktı:

```
IndexError: list index out of range
```

Listeler değiştirilebilir olduğundan bir indekse yeni bir değer atanabilir:

```
meyve = ["elma", "armut", "kiraz"]
meyve[0] = "portakal"
print(meyve)
liste = [5, 9, 3, -8]
liste[0] = liste[2] * 3
liste[1] = liste[0] * 5
print(liste)
```

Çıktı:

```
['portakal', 'armut', 'kiraz']
[9, 45, 3, -8]
```

İkinci örnekte önce `liste[0]` $3 \cdot 3 = 9$ olur; ardından `liste[1]`, `liste[0]`'ın **yeni** değeri kullanılarak $9 \cdot 5 = 45$ olarak hesaplanır.
Klasik programlama dillerindeki dizilerden farklı olarak bir listenin elemanları aynı tipte olmak zorunda değildir. Bir listenin elemanı başka bir liste bile olabilir; buna **iç içe liste** denir.

```
karisik = ["A", 0.1, [1, 2, 3], "Han", 0.54]
print(karisik[2])
print(karisik[2][0])
print(len(karisik))
```

Çıktı:

```
[1, 2, 3]
1
5
```

`karisik[2]` iç listenin kendisidir; `karisik[2][0]` ise iç listenin ilk elemanıdır. İç liste, dış listede tek bir eleman sayılır; bu yüzden `len(karisik)` 5'tir.

6.3 Dilimleme

Bir listenin tek bir elemanını değil, ardışık bir parçasını almak için dilimleme kullanılır.

Tanım 6.7 (Dilimleme). `liste[a:b]` yazımı, `liste`'nin indeksi `a`'dan başlayıp `b`'ye kadar olan elemanlarından yeni bir liste oluşturur; `a` indeksli eleman dahildir, `b` indeksli eleman **dahil değildir**. `a` yazılmazsa baştan, `b` yazılmazsa sona kadar alınır. `liste[a:b:c]` yazımında üçüncü sayı **adımdır**: `a`'dan başlanır ve her seferinde `c` indeks ilerlenerek `b`'ye varmadan durulur. Bu işleme **dilimleme** (slicing), elde edilen parçaya da **dilim** denir.

Yani `liste[1:4]` indeksi 1, 2 ve 3 olan elemanları verir; 4 dahil değildir. Başlangıç dahil, bitiş hariç kuralı sayesinde `liste[a:b]` dilimindeki eleman sayısı tam olarak $b - a$'dır.

 CC BY-NC-SA 4.0

```
listem = [10, 20, 30, 40, 50]
print(listem[1:4])
print(listem[:3])
print(listem[3:])
print(listem[:])
print(listem[1:2])
print(listem[3:1])
```

Çıktı:

```
[20, 30, 40]
[10, 20, 30]
[40, 50]
[10, 20, 30, 40, 50]
[20]
[]
```

`listem[:]` bütün listenin bir kopyasını verir. `listem[1:2]` tek elemanlı bir listedir; `listem[1]` ise elemanın kendisidir: biri `[20]`, diğeri `20`'dir. `listem[3:1]` boş listedir, çünkü başlangıç bitişten sonradır ve adım pozitiftir.
Adım verildiğinde elemanlar atlanarak alınır. Adım negatifse liste sondan başa doğru dolaşılır; bu durumda başlangıç bitişten büyük olmalıdır. `liste[::-1]` listenin tersini verir.

```
liste = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(liste[::2])
print(liste[1::2])
print(liste[1:8:3])
print(liste[::-1])
print(liste[7:2:-2])
print(liste[-3:])
print(liste[:-2])
```

Çıktı:

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

Bu listede her elemanın değeri kendi indeksine eşit olduğundan sonuçları okumak kolaydır. `liste[1:8:3]` dilimi 1'den başlar, üçer ilerler: 1, 4, 7; bir sonraki 10 indeksi 8'i geçtiği için durulur. `liste[7:2:-2]` 7'den başlar ve ikişer geriye gider: 7, 5, 3; bir sonraki 1, bitiş olan 2'yi geçtiği için alınmaz.

Örnek 6.1 (Dilimleri Bulmak). `harfler = ["a", "b", "c", "d", "e", "f", "g"]` listesinden aşağıdaki listeleri birer dilimle elde ediniz: `["c", "d", "e"]`, `["a", "c", "e", "g"]`, `["g", "f", "e"]`, `["f", "d", "b"]`.

> **Çözüm.**
> Önce indeksleri yazalım: `a` 0, `b` 1, `c` 2, `d` 3, `e` 4, `f` 5, `g` 6.
>
> - `["c", "d", "e"]`: indeks 2'den 4'e kadar; bitiş hariç olduğu için `harfler[2:5]`.
> - `["a", "c", "e", "g"]`: baştan sona ikişer atlayarak: `harfler[::2]`.
> - `["g", "f", "e"]`: sondan başlayıp geriye üç eleman. Başlangıç 6, adım -1 ve bitiş 3 (hariç): `harfler[6:3:-1]`; ya da `harfler[:-4:-1]`.
> - `["f", "d", "b"]`: indeks 5'ten başlayıp ikişer geriye: 5, 3, 1. Bitişi yazmasak da olur: `harfler[5::-2]` 5, 3, 1'i verir ve listenin başında durur.
>
> ```
> harfler = ["a", "b", "c", "d", "e", "f", "g"]
> print(harfler[2:5])
> ```

 CC BY-NC-SA 4.0

```
print(harfler[::2])
print(harfler[6:3:-1], harfler[:-4:-1])
print(harfler[5::-2])
```

Çıktı:

```
['c', 'd', 'e']
['a', 'c', 'e', 'g']
['g', 'f', 'e'] ['g', 'f', 'e']
['f', 'd', 'b']
```

■

6.4 Eleman Ekleme ve Silme

Listeler değiştirilebilir olduğu için listeye eleman eklemenin ve listeden eleman çıkarmanın birkaç yolu vardır.

Komut	Yaptığı iş
`liste.append(x)`	`x`'i listenin sonuna tek bir eleman olarak ekler
`liste.extend(diger)`	`diger` koleksiyonunun **elemanlarını** listenin sonuna ekler
`liste.insert(i, x)`	`x`'i `i` indeksine yerleştirir, sonrakileri bir sağa kaydırır
`liste.remove(x)`	değeri `x` olan **ilk** elemanı siler
`liste.pop(i)`	`i` indeksli elemanı listeden çıkarır ve onu verir; `i` yazılmazsa son elemanı çıkarır
`del liste[i]`	`i` indeksli elemanı siler; `del liste[a:b]` bir dilimi siler
`liste.clear()`	bütün elemanları siler, liste boş kalır

```
listem = [1, 2, 3]
listem.append(4)
listem.append(125)
print(listem)
listem.extend([5, 6])
print(listem)
listem.insert(0, 99)
print(listem)
```

Çıktı:

```
[1, 2, 3, 4, 125]
[1, 2, 3, 4, 125, 5, 6]
[99, 1, 2, 3, 4, 125, 5, 6]
```

`append` ile `extend` arasındaki fark, bir liste eklendiğinde ortaya çıkar: `append` listeyi tek bir eleman olarak ekler ve iç içe bir liste oluşur; `extend` ise listenin elemanlarını tek tek ekler.

```
list1 = [1, 2, 3]
list1.append([4, 5, 6])
print(list1, len(list1))
list2 = [1, 2, 3]
```

 CC BY-NC-SA 4.0

```
list2.extend([4, 5, 6])
print(list2, len(list2))
```

Çıktı:

```
[1, 2, 3, [4, 5, 6]] 4
[1, 2, 3, 4, 5, 6] 6
```

Silme komutlarını bir listede sırayla deneyelim:

```
listem = [1, 2, 3, 4, 5, 3]
listem.remove(3)
print(listem)
cikan = listem.pop(1)
print(cikan, listem)
son = listem.pop()
print(son, listem)
del listem[0]
print(listem)
listem.clear()
print(listem)
```

Çıktı:

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

`remove(3)` yalnız ilk `3`'ü sildi; sondaki `3` listede kaldı. `pop` çıkardığı elemanı verdiği için onu bir değişkende saklayabildik.

⚠ Liste metotları yeni bir liste vermez

`append`, `extend`, `insert`, `remove`, `sort`, `reverse` ve `clear` gibi metotlar listenin **kendisini** değiştirir ve geriye bir değer vermez. Böyle bir metodun sonucu bir değişkene atanırsa o değişken, "değer yok" anlamına gelen `None` değerini alır:

```
listem = [1, 2, 3]
yeni = listem.append(4)
print(yeni)
print(listem)
```

Çıktı:

```
None
[1, 2, 3, 4]
```

6.5 Diğer Liste İşlemleri

Listelerle çalışırken sıralama, arama ve sayma gibi işlemlere de sık sık ihtiyaç duyulur.
Ters çevirme ve sıralama. `liste.reverse()` listenin eleman sırasını tersine çevirir. `liste.sort()` listeyi küçükten büyüğe, `liste.sort(reverse=True)` büyükten küçüğe sıralar. Metinlerden oluşan bir liste alfabetik olarak, daha doğrusu karakterlerin bilgisayardaki sırasına göre sıralanır.

```
listem = [10, 2, 30, 14, 5]
listem.reverse()
print(listem)
listem.sort()
print(listem)
```

 CC BY-NC-SA 4.0

```
listem.sort(reverse=True)
print(listem)
```

Çıktı:

```
[5, 14, 30, 2, 10]
[2, 5, 10, 14, 30]
[30, 14, 10, 5, 2]
```

Sayma ve arama. `liste.count(x)` `x`'in listede kaç kez geçtiğini, `liste.index(x)` `x`'in ilk geçtiği yerin indeksini verir; `x` listede yoksa `index` hata verir. Bir değerin listede bulunup bulunmadığı ise `in` operatörüyle sorulur.

Tanım 6.8 (in ve not in Operatörleri). `x in koleksiyon` ifadesi, `x` koleksiyonun bir elemanıysa `True`, değilse `False` olur. `x not in koleksiyon` ifadesi bunun tersidir.

Yani `in`, "elemanı mı?" sorusunun Python'daki karşılığıdır ve matematikteki $x \in A$ yazımına benzer. Listelerin yanında demetler, kümeler, metinler ve `range` nesneleriyle de kullanılır; sözlüklerde anahtarlar arasında arama yapar.

```
listem = [10, 20, 30, 20, 40, 20]
print(listem.count(20), listem.count(50))
print(listem.index(30))
print(20 in listem, 50 in listem, 50 not in listem)
print("a" in "Ali", "A" in "Ali")
```

Çıktı:

```
3 0
2
True False True
False True
```

Birleştirme ve çoğaltma. İki liste `+` ile birleştirilebilir; bu işlem asıl listeleri değiştirmez, yeni bir liste oluşturur. Bir liste bir tam sayıyla çarpıldığında elemanları o kadar kez tekrarlanır.

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
birlesik = list1 + list2
print(birlesik)
print(list1)
print(3 * [0])
print(2 * [1, 2])
```

Çıktı:

```
[1, 2, 3, 4, 5, 6]
[1, 2, 3]
[0, 0, 0]
[1, 2, 1, 2]
```

`3 * [0]` listenin elemanlarını 3 ile çarpmaz; listeyi üç kez yan yana yazar. Elemanları çarpmanın yolunu bir sonraki bölümde liste üreteçleriyle göreceğiz.

6.5.1 Listeyi Kopyalamak

Sayılarda `b = a` ataması `a`'nın değerini `b`'ye kopyalar ve sonra `a` değişse bile `b` etkilenmez. Listelerde durum farklıdır.

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

 CC BY-NC-SA 4.0

```
print("a:", a)
print("b:", b)
```

Çıktı:

```
a: [1, 2, 3, 4]
b: [1, 2, 3, 4]
```

`b.append(4)` komutu yalnız `b`'yi değiştirdiği hâlde `a` da değişti. Bunun nedeni, `b = a` atamasının listeyi kopyalamaması, aynı listeye ikinci bir ad vermesidir: `a` ve `b` bellekteki tek bir listeyi gösterir. Sayılarda bu fark görünmez, çünkü bir sayı yerinde değiştirilemez; `a = 20` gibi bir atama `a`'yı yeni bir değere bağlar. Bağımsız bir kopya için `copy` metodu ya da `[:]` dilimi kullanılır:

```
a = [1, 2, 3]
b = a.copy()
c = a[:]
b.append(4)
c.append(5)
print("a:", a)
print("b:", b)
print("c:", c)
```

Çıktı:

```
a: [1, 2, 3]
b: [1, 2, 3, 4]
c: [1, 2, 3, 5]
```

6.6 Listelerle Kullanılan Hazır Fonksiyonlar

Python'ın hazır fonksiyonlarından birçoğu bir listeyi bütün olarak alıp tek bir sonuç verir.

Fonksiyon	Verdiği sonuç
`len(liste)`	eleman sayısı
`sum(liste)`	elemanların toplamı
`min(liste)`, `max(liste)`	en küçük ve en büyük eleman
`sorted(liste)`	sıralanmış **yeni** bir liste; asıl liste değişmez
`reversed(liste)`	elemanları ters sırada veren bir nesne; `list` ile listeye çevrilir
`all(liste)`	bütün elemanlar doğru sayılıyorsa `True`
`any(liste)`	en az bir eleman doğru sayılıyorsa `True`

```
notlar = [72, 85, 90, 64, 85]
print(len(notlar), sum(notlar))
print(min(notlar), max(notlar))
print(sum(notlar) / len(notlar))
print(sorted(notlar))
print(notlar)
print(list(reversed(notlar)))
print(all([1, 2, 3]), all([1, 0, 3]), any([0, 0, 5]))
```

Çıktı:

```
5 396
64 90
79.2
[64, 72, 85, 85, 90]
```

 CC BY-NC-SA 4.0

```
[72, 85, 90, 64, 85]
[85, 64, 90, 85, 72]
True False True
```

`sorted(notlar)` sıralanmış bir liste verdiği hâlde `notlar` değişmedi; `notlar.sort()` ise listenin kendisini sıralardı. `all([1, 0, 3])` `False`'tur, çünkü `0` yanlış sayılır.
Birden çok listeyi eleman eleman eşleyen `zip` ve elemanları indeksleriyle birlikte veren `enumerate` fonksiyonlarını döngülerle birlikte kullanacağız.

6.7 range Fonksiyonu

Belli bir aralıktaki tam sayıları sırayla vermek için `range` fonksiyonu kullanılır; döngülerin en sık kullanılan yardımcısıdır.

Tanım 6.9 (range Fonksiyonu). `range(baslangic, bitis, adim)` fonksiyonu `baslangic`'tan başlayıp her seferinde `adim` kadar artan ve `bitis`'e **varmadan** duran tam sayıları verir. `adim` yazılmazsa 1 alınır; yalnız `range(bitis)` yazılırsa başlangıç 0 alınır. Adım negatifse sayılar azalır.

Yani `range` dilimlemeyle aynı kuralı izler: başlangıç dahil, bitiş hariçtir. `range(6)` $0, 1, 2, 3, 4, 5$ sayılarını verir; 6 dahil değildir.
`range` sayıları bir liste olarak saklamaz, gerektiğinde tek tek üretir; bu yüzden yazdırıldığında yalnız `range(0, 6)` görünür. Sayıları görmek için `list` ile listeye çevrilir.

```
a = range(6)
print(a)
print(list(a))
print(list(range(1, 6)))
print(list(range(2, 10, 3)))
print(list(range(10, 0, -2)))
r = range(0, 20, 2)
print(11 in r, 10 in r, r.index(10))
```

Çıktı:

```
range(0, 6)
[0, 1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
[2, 5, 8]
[10, 8, 6, 4, 2]
False True 5
```

`range(0, 20, 2)` çift sayıları verdiği için 11 onun içinde değildir; 10 ise altıncı sayıdır ve indeksi 5'tir.

6.7.1 Liste Üreteçlerine Bir Bakış

Bir kurala göre oluşturulan bir liste, `range` ile birlikte tek satırda yazılabilir. Örneğin 1'den 5'e kadar sayıların kareleri şöyle elde edilir:

```
kareler = [x ** 2 for x in range(1, 6)]
print(kareler)
```

Çıktı:

```
[1, 4, 9, 16, 25]
```

Bu yazım matematikteki $\{x^2 : x = 1, 2, \ldots, 5\}$ gösterimine benzer ve **liste üreteci** (list comprehension) adını taşır. Liste üreteçlerini, dayandıkları `for` döngüsüyle birlikte bir sonraki bölümde ayrıntılı göreceğiz.

6.8 Alıştırmalar

Her alıştırmada önce sonucu kâğıt üzerinde bulun, sonra kodu çalıştırarak kontrol edin.

 CC BY-NC-SA 4.0

Alıştırma 6.1 (Elemanlarla Hesap). `liste = [5, 9, 3, -8]` için aşağıdaki programın çıktısını bulunuz.

```
liste = [5, 9, 3, -8]
liste[0] = liste[2] + liste[3]
print(liste)
liste[3] = liste[1] - liste[0]
print(liste)
```

Çözüm.
Birinci atamada `liste[0]` $3 + (-8) = -5$ olur ve liste `[-5, 9, 3, -8]` hâline gelir. İkinci atama `liste[0]`'ın yeni değerini kullanır: `liste[3]` $9 - (-5) = 14$ olur.

```
liste = [5, 9, 3, -8]
liste[0] = liste[2] + liste[3]
print(liste)
liste[3] = liste[1] - liste[0]
print(liste)
```

Çıktı:

```
[-5, 9, 3, -8]
[-5, 9, 3, 14]
```

■

Alıştırma 6.2 (Negatif İndeksli Dilimler). `liste = [10, 51, 66, 13, 48, 15]` için `liste[-1]`, `liste[-3:]`, `liste[-4:-1]` ve `liste[::-1]` değerlerini bulunuz.

Çözüm.
Negatif indeksleri yazalım: `15` -1, `48` -2, `13` -3, `66` -4, `51` -5, `10` -6.

- `liste[-1]` son elemandır: `15`.
- `liste[-3:]` indeks -3'ten sona kadardır: `[13, 48, 15]`.
- `liste[-4:-1]` indeks -4'ten başlar, -1 hariçtir: `[66, 13, 48]`.
- `liste[::-1]` listenin tersidir: `[15, 48, 13, 66, 51, 10]`.

```
liste = [10, 51, 66, 13, 48, 15]
print(liste[-1])
print(liste[-3:])
print(liste[-4:-1])
print(liste[::-1])
```

Çıktı:

```
15
[13, 48, 15]
[66, 13, 48]
[15, 48, 13, 66, 51, 10]
```

■

Alıştırma 6.3 (Metotları Adım Adım İzlemek). Aşağıdaki programın çıktısını bulunuz.

```
sayilar = [4, 1, 7]
sayilar.append(3)
sayilar.insert(1, 9)
sayilar.remove(7)
sayilar.sort()
x = sayilar.pop()
print(sayilar, x)
```

Çözüm.
Her komuttan sonra listenin durumunu yazalım:

- `append(3)`: `[4, 1, 7, 3]`.
- `insert(1, 9)`: 9 indeks 1'e girer, sonrakiler kayar: `[4, 9, 1, 7, 3]`.
- `remove(7)`: `[4, 9, 1, 3]`.
- `sort()`: `[1, 3, 4, 9]`.
- `pop()`: son eleman 9 çıkarılır ve `x`'e atanır; liste `[1, 3, 4]` kalır.

```
sayilar = [4, 1, 7]
sayilar.append(3)
sayilar.insert(1, 9)
sayilar.remove(7)
sayilar.sort()
x = sayilar.pop()
print(sayilar, x)
```

Çıktı:

```
[1, 3, 4] 9
```

■

Alıştırma 6.4 (Ortalama, En Yüksek ve En Düşük). `notlar = [45, 78, 92, 36, 67, 81, 53]` listesindeki notların ortalamasını virgülden sonra iki basamakla, en yüksek ve en düşük notu ve en yüksek notun listedeki indeksini bulunuz.

Çözüm.
Toplam

$$45 + 78 + 92 + 36 + 67 + 81 + 53 = 452$$

, eleman sayısı 7'dir; ortalama $452/7 \approx 64{,}57$'dir. En yüksek not 92, en düşük not 36'dır. 92 listenin üçüncü elemanıdır, indeksi 2'dir.

```
notlar = [45, 78, 92, 36, 67, 81, 53]
ortalama = sum(notlar) / len(notlar)
print(f"Ortalama: {ortalama:.2f}")
print("En yüksek:", max(notlar), "- En düşük:", min(notlar))
print("En yüksek notun indeksi:", notlar.index(max(notlar)))
```

Çıktı:

```
Ortalama: 64.57
En yüksek: 92 - En düşük: 36
En yüksek notun indeksi: 2
```

■

 CC BY-NC-SA 4.0

Alıştırma 6.5 (Farklı Elemanlar). `liste = [3, 1, 3, 2, 1, 3, 5]` listesinde kaç farklı sayı vardır? Farklı sayıları küçükten büyüğe sıralı bir liste olarak yazdırınız.

Çözüm.
Listeyi kümeye çevirmek tekrar eden elemanları atar: geriye $1, 2, 3, 5$ kalır, yani 4 farklı sayı vardır. Kümenin sırası olmadığı için sıralı bir liste elde etmek üzere `sorted` kullanırız; `sorted` her koleksiyonu alır ve bir liste verir.

```
liste = [3, 1, 3, 2, 1, 3, 5]
farkli = set(liste)
print(len(farkli))
print(sorted(farkli))
```

Çıktı:

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

■

Alıştırma 6.6 (range ile Listeler). `range` kullanarak şu listeleri oluşturunuz: 5'ten 50'ye kadar (dahil) 5'in katları; 20'den 1'e kadar (dahil) azalan tek sayılar.

Çözüm.
Birinci liste 5'ten başlar ve 5'er artar. 50'nin dahil olması için bitiş 50'den büyük olmalıdır; 51 alalım. İkinci liste 19'dan başlar ve 2'şer azalır; 1'in dahil olması için bitiş 1'den küçük olmalıdır, 0 alalım.

```
print(list(range(5, 51, 5)))
print(list(range(19, 0, -2)))
```

Çıktı:

```
[5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
[19, 17, 15, 13, 11, 9, 7, 5, 3, 1]
```

■

Alıştırma 6.7 (Kopya mı, Aynı Liste mi?). Aşağıdaki programın çıktısını bulunuz.

```
a = [1, 2, 3]
b = a
c = a[:]
a[0] = 100
print(a, b, c)
```

Çözüm.
`b = a` yeni bir liste oluşturmaz; `a` ve `b` aynı listeyi gösterir. `c = a[:]` ise elemanları kopyalayarak ayrı bir liste oluşturur. `a[0] = 100` atamasından sonra `a` ve `b` aynı değişmiş listeyi gösterir, `c` ise eski hâlinde kalır.

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

 CC BY-NC-SA 4.0

```
a[0] = 100
print(a, b, c)
```

Çıktı:

```
[100, 2, 3] [100, 2, 3] [1, 2, 3]
```

■

Bu bölümde birçok değeri tek bir değişkende tutmayı ve listeleri değiştirmeyi öğrendik. Şimdiye kadar elemanlara hep tek tek, indeksleriyle eriştik. Bir sonraki bölüm for Döngüsü ve Liste Üreteçleri, bir listenin bütün elemanları üzerinde sırayla işlem yapmayı sağlayan döngüleri ele alıyor.

 CC BY-NC-SA 4.0

7. for Döngüsü ve Liste Üreteçleri

Bir listedeki notların ortalamasını `sum` ve `len` ile bulabiliyoruz; peki her notun yanına harf karşılığını yazmak, ya da 1 ile 100 arasındaki asal sayıları tek tek bulmak istersek? Bu tür işlerde aynı komutları her eleman için, ya da her sayı için yeniden çalıştırmak gerekir. Komutları otuz kez alt alta yazmak yerine Python'a "bunu her eleman için yap" deriz.
Bir kod bloğunu tekrar tekrar çalıştıran yapılara **döngü** denir. Python'da iki döngü türü vardır: bir koleksiyonun elemanları üzerinde sırayla dolaşan `for` döngüsü ve bir koşul doğru olduğu sürece devam eden `while` döngüsü. Bu bölüm `for` döngüsünü ve onunla tek satırda liste oluşturmayı sağlayan liste üreteçlerini ele alıyor; `while` döngüsü bir sonraki bölümün konusu.

7.1 for Döngüsünün Yapısı

`for` döngüsü bir koleksiyonun elemanlarını birer birer alır ve her biri için aynı kod bloğunu çalıştırır.

Tanım 7.1 (for Döngüsü).

```
for degisken in koleksiyon:
    her eleman için çalışacak komutlar
```

yapısında `koleksiyon`un elemanları sırayla `degisken`e atanır ve her atamadan sonra girintili komutlar bir kez çalışır. Elemanlar bitince döngü sona erer ve program döngüden sonraki ilk girintisiz satırdan devam eder. Döngü bloğunun her bir çalışmasına bir **yineleme** (iteration), `degisken`e de **döngü değişkeni** denir.

Yani `for isim in isimler:` satırı "isimler listesindeki her isim için" demektir. Döngü değişkeninin adını biz seçeriz; `isim` yerine `x` de yazılabilir, ama anlamlı bir ad kodu okunaklı yapar.

```
isimler = ["Ali", "Ayşe", "Mehmet"]
for isim in isimler:
    print("Merhaba,", isim)
print("Döngü bitti.")
```

Çıktı:

```
Merhaba, Ali
Merhaba, Ayşe
Merhaba, Mehmet
Döngü bitti.
```

Döngü üç kez çalıştı: önce `isim` `"Ali"` oldu, sonra `"Ayşe"`, sonra `"Mehmet"`. Son satır girintisiz olduğu için döngünün parçası değildir ve bir kez, döngü bittikten sonra çalıştı.

7.1.1 range ile for

Bir işi belli bir sayıda tekrarlamak ya da ardışık sayılar üzerinde çalışmak için `for` döngüsü `range` fonksiyonuyla (Tanım 6.9) birlikte kullanılır.

```
for i in range(5):
    print(i, end=" ")
print()
for i in range(2, 6):
    print(i, end=" ")
print()
for i in range(0, 10, 2):
```

 CC BY-NC-SA 4.0

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

Çıktı:

```
0 1 2 3 4
2 3 4 5
0 2 4 6 8
5 4 3 2 1
```

`end=" "` sayıların alt alta değil yan yana yazılmasını sağladı; her döngüden sonraki boş `print()` ise satırı bitirdi. `range(5, 0, -1)` 5'ten geriye doğru sayar ve 0'a varmadan durur.
`range` ile yazılan bir döngü, bir işi belli bir sayıda tekrarlamanın da en kısa yoludur:

```
for i in range(3):
    print("Dikkat!")
```

Çıktı:

```
Dikkat!
Dikkat!
Dikkat!
```

7.1.2 Farklı Koleksiyonlar Üzerinde Döngü

`for` döngüsü listelerin yanında demetler, kümeler, sözlükler ve metinler üzerinde de çalışır.
Bir demet üzerinde dolaşmak listedeki gibidir. Bir metin üzerinde dolaşıldığında döngü değişkeni metnin karakterlerini sırayla alır.

```
demet = (10, 20, 30)
for eleman in demet:
    print(eleman, end=" ")
print()
for harf in "Python":
    print(harf, end="-")
print()
```

Çıktı:

```
10 20 30
P-y-t-h-o-n-
```

Bir sözlük üzerinde doğrudan dolaşıldığında döngü değişkeni sözlüğün **anahtarlarını** alır. Sözlüğün `keys()`, `values()` ve `items()` metotları sırasıyla anahtarları, değerleri ve anahtar–değer çiftlerini verir; `items()` kullanıldığında döngüye iki değişken yazılır ve her çiftin anahtarı birincisine, değeri ikincisine atanır.

```
sozluk = {"isim": "Ali", "yas": 25, "sehir": "İstanbul"}
for anahtar in sozluk:
    print(anahtar, end=" ")
print()
for deger in sozluk.values():
    print(deger, end=" ")
print()
for anahtar, deger in sozluk.items():
    print(anahtar, ":", deger)
```

Çıktı:

 CC BY-NC-SA 4.0

```
isim yas sehir
Ali 25 İstanbul
isim : Ali
yas : 25
sehir : İstanbul
```

7.2 Döngüyle Toplam ve Çarpım

Döngülerin en sık kullanımlarından biri, bir değeri her yinelemede biraz daha güncelleyerek bir toplam ya da çarpım biriktirmektir.

Tanım 7.2 (Biriktirici Değişken). Döngüden önce bir başlangıç değeri verilen ve her yinelemede birleşik atamayla (Tanım 2.12) güncellenen değişkene **biriktirici** (accumulator) denir. Toplam için başlangıç değeri 0, çarpım için 1 alınır.

Yani biriktirici, döngü boyunca bir kumbara gibi çalışır: başta boştur, her yinelemede içine bir şey eklenir ve döngü bittiğinde sonucu taşır. Başlangıç değerleri toplamada ve çarpmada etkisiz eleman olan 0 ve 1'dir; çarpımın başlangıcı yanlışlıkla 0 alınırsa sonuç her zaman 0 çıkar.

Döngüyle toplam üç adımda

1. Döngüden önce biriktiriciyi başlat: `toplam = 0` (çarpım için `carpim = 1`).
2. Döngünün içinde her değeri biriktiriciye ekle: `toplam += deger` (çarpım için `carpim *= deger`).
3. Döngü bittikten sonra, girintisiz bir satırda sonucu kullan.

Örnek 7.1 (Birden Yüze Kadar Sayıların Toplamı). $1 + 2 + \cdots + 100$ toplamını bir `for` döngüsüyle hesaplayınız ve $\frac{n(n+1)}{2}$ formülüyle karşılaştırınız.

Çözüm.
Tarifin üç adımını uygularız. `range(1, 101)` 1'den 100'e kadar sayıları verir; 101 dahil değildir.

```
toplam = 0
for i in range(1, 101):
    toplam += i
print(toplam)
print(100 * 101 // 2)
```

Çıktı:

```
5050
5050
```

Formül $n = 100$ için $100 \cdot 101/2 = 5050$ verir ve döngünün sonucuyla aynıdır. `print(toplam)` satırı girintisiz yazıldığı için döngü bittikten sonra bir kez çalışır; girintili yazılsaydı her yinelemede ara toplamı yazar ve 100 satırlık bir çıktı oluşurdu. ■

Örnek 7.2 (Faktöriyel). Bir `sayi` için sayi ! değerini bir `for` döngüsüyle hesaplayan, negatif sayılar için uyarı veren ve $0! = 1$ durumunu ayrıca yazan bir program yazınız. `sayi = 5` için deneyiniz.

Çözüm.
$n! = 1 \cdot 2 \cdots n$ bir çarpım olduğundan biriktiriciyi 1 ile başlatırız. Negatif sayıların faktöriyeli tanımlı olmadığından önce bir `if-elif-else` ile durumları ayırırız:

 CC BY-NC-SA 4.0

```
sayi = 5
faktoriyel = 1
if sayi < 0:
    print("Negatif sayının faktöriyeli yoktur.")
elif sayi == 0:
    print("0! = 1")
else:
    for i in range(1, sayi + 1):
        faktoriyel *= i
    print(f"{sayi}! = {faktoriyel}")
```

Çıktı:

```
5! = 120
```

`range(1, sayi + 1)` 1'den `sayi`'ya kadar olan sayıları verir; bitiş hariç olduğu için `sayi + 1` yazdık. Döngü `else` bloğunun içinde olduğundan bir girinti daha içeride yazıldı. ■

Biriktirici yalnız sayı değil, bir liste de olabilir: boş bir listeyle başlanır ve her yinelemede `append` ile bir eleman eklenir.

```
liste = [1, 2, 3, 4, 5]
sonuc = []
for eleman in liste:
    sonuc.append(eleman * 2)
print(sonuc)
```

Çıktı:

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

`print` döngünün içine yazılsaydı, listenin her yinelemede nasıl büyüdüğü görülürdü:

```
liste = [1, 2, 3, 4, 5]
sonuc = []
for eleman in liste:
    sonuc.append(eleman ** 2)
    print(sonuc)
```

Çıktı:

```
[1]
[1, 4]
[1, 4, 9]
[1, 4, 9, 16]
[1, 4, 9, 16, 25]
```

 CC BY-NC-SA 4.0

⚠ Biriktiriciyi sıfırlamayı unutmayın

Bir biriktirici her hesaptan önce yeniden başlatılmalıdır. Aşağıdaki programda ikinci döngüden önce `sonuc = []` yazılmadığı için kareler, ilk döngünün bıraktığı listenin sonuna eklenir. Colab'da bir hücreyi `sonuc = []` satırı olmadan ikinci kez çalıştırmak da aynı hataya yol açar.

```
liste = [1, 2, 3]
sonuc = []
for eleman in liste:
    sonuc.append(eleman * 2)
for eleman in liste:
    sonuc.append(eleman ** 2)
print(sonuc)
```

Çıktı:

```
[2, 4, 6, 1, 4, 9]
```

7.3 break, continue ve pass

Bazen bir döngünün bütün elemanları dolaşmadan bitmesi ya da bazı elemanların atlanması istenir.

Tanım 7.3 (break ve continue). Döngünün içinde çalışan `break` komutu döngüyü hemen sonlandırır ve program döngüden sonraki ilk satırdan devam eder. `continue` komutu ise yalnız o anki yinelemenin geri kalanını atlar ve döngü bir sonraki elemanla devam eder.

Yani `break` "dur, döngüden çık", `continue` ise "bu elemanı geç, sıradakine bak" demektir. İkisi de genellikle bir `if` koşulunun içinde kullanılır.

```
for i in range(5):
    if i == 3:
        break
    print(i, end=" ")
print()
for i in range(5):
    if i == 3:
        continue
    print(i, end=" ")
print()
```

Çıktı:

```
0 1 2
0 1 2 4
```

Birinci döngüde `i` 3 olduğunda `break` döngüyü bitirdi; 3 ve 4 hiç yazılmadı. İkinci döngüde yalnız 3 atlandı.

Örnek 7.3 (İlk Negatif Sayıda Durmak). `sayilar = [1, 512, 6, -8, 3]` listesindeki sayıları sırayla yazan, ama ilk negatif sayıya gelince bir uyarı yazıp duran bir program yazınız. Aynı listede negatif sayıları atlayıp diğerlerini yazan programla karşılaştırınız.

> **Çözüm.**
> İlk program için negatif sayıyı bulunca `break` kullanırız:
>
> ```
> sayilar = [1, 512, 6, -8, 3]
> for x in sayilar:
>     if x < 0:
>         print("Negatif sayı bulundu!")
>         break
> ```

 CC BY-NC-SA 4.0

```
    print(x)
print("Buradan devam...")
```

Çıktı:

```
1
512
6
Negatif sayı bulundu!
Buradan devam...
```

`-8`'de döngü bittiği için ondan sonra gelen `3` yazılmadı; program döngüden sonraki satırdan devam etti. İkinci programda `break` yerine `continue` kullanırız; bu kez yalnız `-8` atlanır:

```
sayilar = [1, 512, 6, -8, 3]
for x in sayilar:
    if x < 0:
        continue
    print(x)
```

Çıktı:

```
1
512
6
3
```

■

Python'da bir `for` bloğu boş bırakılamaz. Döngünün içine henüz bir şey yazılmayacaksa, koşullu ifadelerde olduğu gibi `pass` (Tanım 5.11) yazılır:

```
for x in [0, 512, 6, -8, 3]:
    pass
print("Buradan devam...")
```

Çıktı:

```
Buradan devam...
```

7.3.1 Kullanılmayan Döngü Değişkeni

Bazı döngülerde döngü değişkeninin değerine hiç ihtiyaç duyulmaz; yalnız işin belli sayıda tekrarlanması istenir.

Tanım 7.4 (Alt Çizgi Değişkeni). Değeri kullanılmayacak bir değişkene ad olarak tek bir alt çizgi `_` verilmesi Python'da yaygın bir alışkanlıktır. `_` sıradan bir değişken adıdır; okura "bu değeri kullanmıyorum" mesajını verir.

Yani `for _ in range(3):` "üç kez tekrarla" demektir; döngü değişkeni `0`, `1`, `2` değerlerini yine alır ama programda kullanılmaz. Değişkene `i` adı verilseydi okur `i`'nin nerede kullanıldığını arardı.

```
for _ in range(3):
    print("Merhaba!")
```

Çıktı:

 CC BY-NC-SA 4.0

```
Merhaba!
Merhaba!
Merhaba!
```

Alt çizgi, çoklu atamada istenmeyen değerleri atmak için de kullanılır. Çoklu atamada bir değişkenin önüne * konursa o değişken, diğer değişkenlere düşmeyen bütün değerleri bir liste olarak alır:

```
x, *_, y = (1, 2, 3, 4, 5)
print(x, y)
print(_)
ilk, *orta, son = [10, 20, 30, 40]
print(ilk, orta, son)
```

Çıktı:

```
1 5
[2, 3, 4]
10 [20, 30] 40
```

7.4 enumerate ve zip

Bir döngüde elemanların yanında indekslerine de ihtiyaç duyulduğunda ya da iki liste aynı anda dolaşılmak istendiğinde iki hazır fonksiyon işi kolaylaştırır.

Tanım 7.5 (enumerate Fonksiyonu). `enumerate(koleksiyon)` fonksiyonu koleksiyonun her elemanını, indeksiyle birlikte `(indeks, eleman)` çiftleri olarak verir. `for i, x in enumerate(koleksiyon):` döngüsünde `i` indeksi, `x` elemanı alır.

Yani `enumerate`, elemanları numaralandırarak dolaşmanın yoludur.

```
isimler = ["Ali", "Ayşe", "Mehmet"]
for indeks, isim in enumerate(isimler):
    print(indeks, isim)
print(list(enumerate("abc")))
```

Çıktı:

```
0 Ali
1 Ayşe
2 Mehmet
[(0, 'a'), (1, 'b'), (2, 'c')]
```

Tanım 7.6 (zip Fonksiyonu). `zip(liste1, liste2)` fonksiyonu iki koleksiyonun aynı sıradaki elemanlarını eşleyerek `(liste1[0], liste2[0])`, `(liste1[1], liste2[1])`, … çiftlerini verir. Koleksiyonların uzunlukları farklıysa kısa olan bitince durur.

Yani `zip` bir fermuar gibi iki listeyi diş diş birleştirir. İkiden fazla koleksiyon da verilebilir; o zaman üçlüler, dörtlüler oluşur.

```
isimler = ["Ali", "Ayşe", "Mehmet"]
yaslar = [25, 30, 35]
print(list(zip(isimler, yaslar)))
for isim, yas in zip(isimler, yaslar):
    print(f"{isim} {yas} yaşında")
print(list(zip([1, 2, 3, 4], ["a", "b"])))
```

Çıktı:

```
[('Ali', 25), ('Ayşe', 30), ('Mehmet', 35)]
Ali 25 yaşında
```

 CC BY-NC-SA 4.0

```
Ayşe 30 yaşında
Mehmet 35 yaşında
[(1, 'a'), (2, 'b')]
```

7.5 Liste Üreteçleri

Bir döngüyle boş bir listeyi doldurmak o kadar sık yapılan bir iştir ki Python bunun için tek satırlık bir yazım sunar.

Tanım 7.7 (Liste Üreteci).

```
[ifade for degisken in koleksiyon]
[ifade for degisken in koleksiyon if koşul]
```

yazımına **liste üreteci** (list comprehension) denir. Birinci yazım, koleksiyonun her elemanı için `ifade`'yi hesaplar ve sonuçlardan bir liste oluşturur. İkinci yazımda yalnız `koşul`u sağlayan elemanlar alınır.

Yani `[x ** 2 for x in range(1, 6)]` liste üreteci,

```
sonuc = []
for x in range(1, 6):
    sonuc.append(x ** 2)
```

döngüsünün yaptığı işi tek satırda yapar. Yazım, matematikteki küme gösterimine benzer: $\{x^2 : 1 \leq x \leq 5\}$ ya da $\{x : 0 \leq x < 10, x \text{ çift}\}$.

```
kareler = [x ** 2 for x in range(10)]
print(kareler)
ciftler = [x for x in range(10) if x % 2 == 0]
print(ciftler)
liste = [1, 2, 3, 4, 5]
print([eleman * 2 for eleman in liste])
print([x + x ** 2 for x in liste])
print([1 / x for x in range(2, 7, 3)])
print([(1 / 2) ** n for n in range(1, 6)])
```

Çıktı:

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[0, 2, 4, 6, 8]
[2, 4, 6, 8, 10]
[2, 6, 12, 20, 30]
[0.5, 0.2]
[0.5, 0.25, 0.125, 0.0625, 0.03125]
```

Liste üreteciyle elde edilen liste sıradan bir listedir; indekslenebilir, dilimlenebilir, `sum` ya da `max` gibi fonksiyonlara verilebilir.

```
liste2 = [2 ** x for x in range(1, 10)]
print(liste2)
print(sum(liste2), min(liste2), max(liste2))
print(liste2[0], liste2[-1])
liste = [1, 2, 4, 5]
print(5 * liste)
print([5 * x for x in liste])
```

Çıktı:

```
[2, 4, 8, 16, 32, 64, 128, 256, 512]
1022 2 512
```

 CC BY-NC-SA 4.0

```
2 512
[1, 2, 4, 5, 1, 2, 4, 5, 1, 2, 4, 5, 1, 2, 4, 5, 1, 2, 4, 5]
[5, 10, 20, 25]
```

Son iki satır Koleksiyonlar ve Listeler bölümünde değindiğimiz farkı gösteriyor: `5 * liste` listeyi beş kez yan yana yazar, `[5 * x for x in liste]` ise her elemanı 5 ile çarpar.
İfadenin yerine bir koşullu ifade (Tanım 5.10) de yazılabilir; bu durumda her eleman listeye girer ama koşula göre farklı bir değer alır.

```
sayilar = [1, 2, 3, 4, 5]
print(["çift" if x % 2 == 0 else "tek" for x in sayilar])
print([x if x > 2 else 0 for x in sayilar])
```

Çıktı:

```
['tek', 'çift', 'tek', 'çift', 'tek']
[0, 0, 3, 4, 5]
```

Koşulun yeri anlamı değiştirir: sonda yazılan `if`, elemanı listeye alıp almamaya karar verir; başta yazılan `if ... else ...` ise her elemanın listeye hangi değerle gireceğini belirler.
Bir liste üretecinde birden çok `for` da bulunabilir. Bu durumda sondaki `for`, öndekinin her değeri için baştan sona dolaşılır:

```
print([(x, y) for x in range(1, 4) for y in range(1, 3)])
print([(2 * x + 1, y ** 2) for x in range(1, 3) for y in range(1, 4)])
```

Çıktı:

```
[(1, 1), (1, 2), (2, 1), (2, 2), (3, 1), (3, 2)]
[(3, 1), (3, 4), (3, 9), (5, 1), (5, 4), (5, 9)]
```

Liste üreteci köşeli parantez olmadan, doğrudan `sum`, `max`, `min`, `all` ya da `any` gibi bir fonksiyonun içine de yazılabilir: `sum(x ** 2 for x in range(1, 6))` kareler toplamı 55'i verir. Bu yazımın adı **üreteç ifadesidir** ve Demetler bölümünde ayrıntılı olarak ele alınır.

> **i** Döngü mü, liste üreteci mi?
>
> Bir kurala göre liste oluşturmak için liste üreteci daha kısa, daha okunaklı ve genellikle daha hızlıdır. Döngünün içinde birkaç adımlık bir iş yapılıyor, bir şey yazdırılıyor ya da `break` gerekiyorsa açık bir `for` döngüsü daha uygundur. Çok büyük sayı dizileri üzerinde hızlı hesap için ise NumPy kütüphanesinin dizileri kullanılır; bkz. Python ile Bilimsel Hesaplama.

7.6 Örnekler

`for` döngüsünü koşullu ifadelerle birlikte kullanarak sayılar üzerinde birkaç klasik problemi çözelim.

Örnek 7.4 (Listedeki Sayılar Çift mi, Tek mi?). `sayilar = [10, 15, 20, 25, 30]` listesindeki her sayının çift mi tek mi olduğunu yazınız.

Çözüm.
Her sayı için `% 2` ile kalanı denetleriz. İki durum birbirinin tersi olduğundan `if-else` yeterlidir:

```
sayilar = [10, 15, 20, 25, 30]
for sayi in sayilar:
    if sayi % 2 == 0:
        print(f"{sayi} çift bir sayıdır.")
    else:
        print(f"{sayi} tek bir sayıdır.")
```

Çıktı:

 CC BY-NC-SA 4.0

```
10 çift bir sayıdır.
15 tek bir sayıdır.
20 çift bir sayıdır.
25 tek bir sayıdır.
30 çift bir sayıdır.
```

■

Örnek 7.5 (Not Listesinin Harf Karşılıkları). `notlar = [85, 72, 90, 60, 40, 77]` listesindeki her not için harf karşılığını yazınız: 85 ve üstü A, 70–84 B, 50–69 C, 50'nin altı D.

Çözüm.
Koşullu İfadeler bölümündeki harf notu kurallarını (Örnek 5.2) bir döngünün içine koyarız. `not` bir anahtar kelime olduğu için değişkene `not_` adını verdik:

```
notlar = [85, 72, 90, 60, 40, 77]
for not_ in notlar:
    if not_ >= 85:
        harf = "A"
    elif not_ >= 70:
        harf = "B"
    elif not_ >= 50:
        harf = "C"
    else:
        harf = "D"
    print(f"{not_} - Harf notu: {harf}")
```

Çıktı:

```
85 - Harf notu: A
72 - Harf notu: B
90 - Harf notu: A
60 - Harf notu: C
40 - Harf notu: D
77 - Harf notu: B
```

■

Örnek 7.6 (Yüze Kadar Asal Sayılar). 2 ile 100 arasındaki asal sayıları bulunuz.

Çözüm.
Bir $n \geq 2$ sayısı, 1 ve kendisinden başka pozitif böleni yoksa asaldır. Bir böleni varsa, o bölenlerden biri $\sqrt{n}$'den büyük değildir: $n = a \cdot b$ ve $a \leq b$ ise $a^2 \leq ab = n$ olur. Bu yüzden n'i yalnız 2'den $\lfloor \sqrt{n} \rfloor$'e kadar olan sayılara bölmeyi denemek yeter.
Her sayı için önce asal olduğunu varsayarız (`asal_mi = True`). İçteki döngü bölen adaylarını dener; bir bölen bulunursa varsayım yanlıştır, `asal_mi False` yapılır ve `break` ile içteki döngüden çıkılır. Bir döngünün içine başka bir döngü yazmayı bir sonraki bölümde ayrıntılı olarak göreceğiz.

```
for sayi in range(2, 101):
    asal_mi = True
    for i in range(2, int(sayi ** 0.5) + 1):
        if sayi % i == 0:
            asal_mi = False
            break
    if asal_mi:
```

 CC BY-NC-SA 4.0

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

Çıktı:

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

`sayi ** 0.5` karekökü verir; `int` onu aşağı keser ve `range`'in bitişi hariç olduğu için 1 eklenir. 2 ve 3 için `range(2, 2)` boştur: içteki döngü hiç çalışmaz ve sayılar asal kalır. Yüze kadar 25 asal sayı vardır. ■

Aynı asallık denemesi `all` fonksiyonu ve bir liste üreteciyle tek satırda da yazılabilir: n asaldır ancak ve ancak $2 \le d \le \sqrt{n}$ olan bütün d'ler için $n \bmod d \neq 0$ ise.

```
asallar = [n for n in range(2, 16)
           if all(n % d != 0 for d in range(2, int(n ** 0.5) + 1))]
print(asallar)
```

Çıktı:

```
[2, 3, 5, 7, 11, 13]
```

Uzun bir satır, parantezlerin içinde olduğu sürece alt satıra bölünebilir; Python parantez kapanana kadar satırın sürdüğünü bilir.

Örnek 7.7 (Mükemmel Sayılar). Kendisi hariç pozitif bölenlerinin toplamı kendisine eşit olan sayılara **mükemmel sayı** denir: $6 = 1 + 2 + 3$ gibi. 10 000'den küçük mükemmel sayıları bulunuz.

Çözüm.
Bir x sayısının kendisi hariç bölenleri, $1 \le i < x$ ve $x \bmod i = 0$ koşulunu sağlayan i'lerdir; bunların toplamı `sum(i for i in range(1, x) if x % i == 0)` ile hesaplanır. Bu toplamın x'e eşit olduğu sayıları bir liste üreteciyle seçeriz:

```
mukemmel = [x for x in range(2, 10000)
            if sum(i for i in range(1, x) if x % i == 0) == x]
print(mukemmel)
```

Çıktı:

```
[6, 28, 496, 8128]
```

Örneğin $28 = 1 + 2 + 4 + 7 + 14$'tür. Bu program her sayı için kendisinden küçük bütün sayıları denediğinden yaklaşık $5 \cdot 10^7$ bölme yapar ve birkaç saniye sürebilir. ■

Örnek 7.8 (Üç Basamaklı Armstrong Sayıları). Rakamlarının küpleri toplamı kendisine eşit olan üç basamaklı sayılara **Armstrong sayısı** denir: $153 = 1^3 + 5^3 + 3^3$ gibi. Bütün üç basamaklı Armstrong sayılarını bulunuz.

Çözüm.
Bir sayının rakamlarını almanın kolay bir yolu onu metne çevirip karakterleri üzerinde dolaşmaktır: `str(153)` `"153"` metnini verir, her karakter `int` ile yeniden rakama çevrilir.

```
armstrong = [x for x in range(100, 1000)
             if sum(int(r) ** 3 for r in str(x)) == x]
print(armstrong)
```

Çıktı:

 CC BY-NC-SA 4.0

```
[153, 370, 371, 407]
```

Örneğin $370 = 27 + 343 + 0$ ve $407 = 64 + 0 + 343$'tür. ■

Örnek 7.9 (Fibonacci Dizisi). Her terimi kendinden önceki iki terimin toplamı olan $0, 1, 1, 2, 3, 5, \ldots$ dizisine **Fibonacci dizisi** denir. Dizinin ilk 10 terimini bir liste olarak bulunuz.

Çözüm.
Listeyi ilk iki terimle başlatırız. Her yinelemede listenin son iki elemanı `fib[-1]` ve `fib[-2]` ile alınır, toplamları listenin sonuna eklenir. İlk iki terim hazır olduğundan 8 yineleme yeter:

```
fib = [0, 1]
for _ in range(8):
    fib.append(fib[-1] + fib[-2])
print(fib)
```

Çıktı:

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Döngü değişkeni kullanılmadığı için _ yazdık. Bu işi `[fib.append(...) for _ in range(8)]` gibi bir liste üreteciyle yapmak da mümkündür, ama önerilmez: liste üreteci bir liste **oluşturmak** içindir, bir listeyi değiştirmenin yan etkisi için değil. ■

7.7 Alıştırmalar

Her alıştırmayı önce bir `for` döngüsüyle, uygun olanları sonra bir liste üreteciyle de çözmeyi deneyin.

Alıştırma 7.1 (Kareler Toplamı). $1^2 + 2^2 + \cdots + 10^2$ toplamını bir döngüyle hesaplayınız ve $\frac{n(n+1)(2n+1)}{6}$ formülüyle karşılaştırınız.

Çözüm.
Biriktiriciyi 0 ile başlatır, her i için i^2'yi ekleriz. Formül $n = 10$ için $10 \cdot 11 \cdot 21/6 = 2310/6 = 385$ verir.

```
toplam = 0
for i in range(1, 11):
    toplam += i ** 2
print(toplam)
print(10 * 11 * 21 // 6)
print(sum(i ** 2 for i in range(1, 11)))
```

Çıktı:

```
385
385
385
```

Son satır aynı toplamı bir üreteç ifadesiyle tek satırda hesaplar. ■

Alıştırma 7.2 (Sesli Harfleri Saymak). `"bilgisayar programlama"` metninde kaç sesli harf bulunduğunu bir döngüyle sayınız.

Çözüm.

 CC BY-NC-SA 4.0

Türkçenin sesli harflerini bir metinde toplarız: `"aeıioöuü"`. Metnin her karakteri için, karakter bu metinde varsa (`in`) sayacı bir artırırız. Sayaç da bir biriktiricidir ve 0 ile başlar.

```
metin = "bilgisayar programlama"
sesliler = "aeıioöuü"
sayac = 0
for harf in metin:
    if harf in sesliler:
        sayac += 1
print(sayac)
```

Çıktı:

```
8
```

Sesliler tek tek sayılırsa: “bilgisayar” sözcüğünde i, i, a, a; “programlama” sözcüğünde o, a, a, a. Toplam $4 + 4 = 8$’dir. Boşluk ve sessiz harfler sayılmaz. ■

Alıştırma 7.3 (max Kullanmadan En Büyük Eleman). `liste = [23, 7, 41, 15, 41, 8]` listesinin en büyük elemanını ve ilk geçtiği indeksi `max` fonksiyonunu kullanmadan bulunuz.

Çözüm.
İlk elemanı geçici olarak en büyük kabul ederiz. Sonra elemanları indeksleriyle (`enumerate`) dolaşır, şimdiye kadarki en büyükten **büyük** bir eleman görünce en büyüğü ve indeksini güncelleriz. Eşit bir eleman güncelleme yapmadığı için ilk geçtiği indeks korunur.

```
liste = [23, 7, 41, 15, 41, 8]
en_buyuk = liste[0]
yer = 0
for i, x in enumerate(liste):
    if x > en_buyuk:
        en_buyuk = x
        yer = i
print(en_buyuk, yer)
```

Çıktı:

```
41 2
```

■

Alıştırma 7.4 (Bir Sayının Bölenleri). 36’nın bütün pozitif bölenlerini bir liste üreteciyle bulunuz ve kaç tane olduklarını yazınız.

Çözüm.
d, 36’yı bölüyorsa $36 \bmod d = 0$’dır. 1’den 36’ya kadar bütün d’leri deneriz:

```
n = 36
bolenler = [d for d in range(1, n + 1) if n % d == 0]
print(bolenler)
print(len(bolenler))
```

Çıktı:

 CC BY-NC-SA 4.0

```
[1, 2, 3, 4, 6, 9, 12, 18, 36]
9
```

$36 = 2^2 \cdot 3^2$ olduğundan bölen sayısı $(2+1)(2+1) = 9$'dur ve sonuçla uyuşur. ■

Alıştırma 7.5 (Metni Tersine Çevirmek). `"python"` metnini bir döngüyle tersine çeviriniz.

Çözüm.
Boş bir metinle başlarız; her harfi, o ana kadar elde edilen metnin **başına** ekleriz. Böylece ilk harf en sona kayar.

```
metin = "python"
ters = ""
for harf in metin:
    ters = harf + ters
    print(ters)
print(metin[::-1])
```

Çıktı:

```
p
yp
typ
htyp
ohtyp
nohtyp
nohtyp
```

Döngünün içindeki `print`, metnin adım adım nasıl oluştuğunu gösteriyor. Son satır aynı sonucu metin dilimlemeyle verir: metinler de listeler gibi dilimlenebilir. ■

Alıştırma 7.6 (Pozitif Sayıların Ortalaması). `sayilar = [4, -2, 7, 0, -5, 9]` listesindeki pozitif sayıların ortalamasını bulunuz.

Çözüm.
Önce pozitif sayıları bir liste üreteciyle seçeriz: $4, 7, 9$. Sıfır pozitif değildir. Ortalama $(4+7+9)/3 = 20/3 \approx 6{,}67$'dir.

```
sayilar = [4, -2, 7, 0, -5, 9]
pozitifler = [x for x in sayilar if x > 0]
print(pozitifler)
print(f"{sum(pozitifler) / len(pozitifler):.2f}")
```

Çıktı:

```
[4, 7, 9]
6.67
```

Listede hiç pozitif sayı olmasaydı `len(pozitifler)` sıfır olur ve bölme hata verirdi; gerçek bir programda bu durum önce bir `if` ile denetlenmelidir. ■

Alıştırma 7.7 (Yedinin Çarpım Tablosu). 7'nin 1'den 10'a kadar çarpım tablosunu `7 x 1 = 7` biçiminde yazdırınız.

 CC BY-NC-SA 4.0

Çözüm.
`range(1, 11)` ile çarpanları dolaşır, her satırı bir f-string ile yazarız:

```
for i in range(1, 11):
    print(f"7 x {i} = {7 * i}")
```

Çıktı:

```
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
```

■

Alıştırma 7.8 (Bir Metinde continue ve break). Aşağıdaki iki programın çıktılarını bulunuz.

```
for i in "zeynep":
    if i == "e" or i == "y":
        continue
    print(i, end=" ")
print()
for i in "zeynep":
    if i == "e" or i == "y":
        break
    print(i, end=" ")
print()
```

Çözüm.
"zeynep" metninin harfleri sırayla z, e, y, n, e, p'dir. Birinci döngüde `e` ve `y` harfleri `continue` ile atlanır; geriye z, n, p kalır. İkinci döngüde `break` ilk e'de döngüyü bitirir; bu yüzden yalnız z yazılır.

```
for i in "zeynep":
    if i == "e" or i == "y":
        continue
    print(i, end=" ")
print()
for i in "zeynep":
    if i == "e" or i == "y":
        break
    print(i, end=" ")
print()
```

Çıktı:

```
z n p
z
```

■

 CC BY-NC-SA 4.0

Bu bölümde `for` döngüsüyle bir koleksiyonun bütün elemanları üzerinde işlem yapmayı ve liste üreteçleriyle tek satırda liste oluşturmayı öğrendik. Asal sayı örneğinde bir döngünün içine ikinci bir döngü yazdık; ayrıca `for` döngüsü, kaç kez çalışacağı önceden belli olan işler içindir. Bir sonraki bölüm İç İçe Döngüler, while Döngüsü ve match-case, iç içe döngüleri ve bir koşul sağlandığı sürece süren `while` döngüsünü ele alıyor.

 CC BY-NC-SA 4.0

8. İç İçe Döngüler, while Döngüsü ve match-case

Bir önceki bölümde bir listenin elemanları üzerinde `for` döngüsüyle dolaştık. Bazı problemlerde ise iki şey birden dolaşılır: bir tablonun satırları ve her satırın sütunları, ya da bir sayı aralığı ve her sayının bölen adayları. Bunun için bir döngünün içine ikinci bir döngü yazılır.
Bazı tekrarların sayısı ise önceden belli değildir: kullanıcı doğru sayıyı bulana kadar tahmin almak, ya da geçerli bir değer girene kadar soruyu yeniden sormak gibi. Bu tür işler için bir koşul doğru olduğu sürece dönen `while` döngüsü kullanılır. Bölümün sonunda, bir değişkenin değerine göre çok sayıda durum arasından seçim yapmayı kolaylaştıran `match-case` yapısını göreceğiz.

8.1 İç İçe Döngüler

Bir döngünün bloğunun içine başka bir döngü yazılabilir; böylece dıştaki döngünün her yinelemesinde içteki döngü baştan sona bir kez çalışır.

Tanım 8.1 (İç İçe Döngü).

```
for dis_degisken in dis_koleksiyon:
    for ic_degisken in ic_koleksiyon:
        içteki döngüde yapılacak işlemler
```

yapısına **iç içe döngü** (nested loop) denir. Dıştaki döngünün her yinelemesinde içteki döngü bütün elemanlarını dolaşır. Dıştaki döngü m, içteki döngü n kez dönüyorsa içteki blok toplam $m \cdot n$ kez çalışır.

Yani iç içe döngü bir saatin akrep ve yelkovanı gibi çalışır: yelkovan (içteki döngü) tam bir tur attığında akrep (dıştaki döngü) bir adım ilerler.

```
for i in range(3):
    for j in range(2):
        print(f"Dış döngü: {i}, İç döngü: {j}")
```

Çıktı:

```
Dış döngü: 0, İç döngü: 0
Dış döngü: 0, İç döngü: 1
Dış döngü: 1, İç döngü: 0
Dış döngü: 1, İç döngü: 1
Dış döngü: 2, İç döngü: 0
Dış döngü: 2, İç döngü: 1
```

Dıştaki döngü 3, içteki 2 kez döndüğü için içteki `print` $3 \cdot 2 = 6$ kez çalıştı. `i` her değerini alırken `j` baştan başlayıp 0 ve 1 değerlerini aldı.
İçteki döngünün aralığı, dıştaki döngünün değişkenine bağlı da olabilir. Aşağıdaki programda i. satırda $i + 1$ yıldız yazılır:

```
for i in range(5):
    for j in range(i + 1):
        print("*", end=" ")
    print()
```

Çıktı:

 CC BY-NC-SA 4.0

```
*
* *
* * *
* * * *
* * * * *
```

İçteki döngü yıldızları `end=" "` ile aynı satıra yazar; dıştaki döngünün bloğundaki boş `print()` ise her satırın sonunda yeni satıra geçer. Girintiye dikkat edin: `print()` içteki döngünün değil, dıştaki döngünün içindedir.

Örnek 8.1 (Çarpım Tablosu). 1'den 5'e kadar sayıların çarpım tablosunu hizalı bir tablo olarak yazdırınız.

Çözüm.
Dıştaki döngü satırları, içteki döngü sütunları dolaşır; i. satırın j. sütununa $i \cdot j$ yazılır. Sayıların hizalı durması için her çarpımı f-string'in `:4` biçim belirteciyle dört karakterlik bir alana sağa yaslı olarak yazarız:

```
for i in range(1, 6):
    for j in range(1, 6):
        print(f"{i * j:4}", end="")
    print()
```

Çıktı:

```
   1   2   3   4   5
   2   4   6   8  10
   3   6   9  12  15
   4   8  12  16  20
   5  10  15  20  25
```

Tablo köşegene göre simetriktir, çünkü $i \cdot j = j \cdot i$'dir. ■

8.1.1 İç İçe Döngülerde break ve continue

İç içe döngülerde `break` ve `continue` yalnız **içinde bulundukları** döngüyü etkiler.
İçteki döngüde çalışan bir `break` yalnız içteki döngüyü bitirir; dıştaki döngü bir sonraki yinelemesiyle devam eder.

```
for i in range(3):
    for j in range(3):
        if j == 1:
            break
        print(f"i: {i}, j: {j}")
```

Çıktı:

```
i: 0, j: 0
i: 1, j: 0
i: 2, j: 0
```

Her `i` için içteki döngü `j = 0`'ı yazdı, `j = 1`'de bitti; ama dıştaki döngü durmadı. `continue` kullanılsaydı yalnız `j = 1` atlanırdı:

```
for i in range(3):
    for j in range(3):
        if j == 1:
            continue
        print(f"i: {i}, j: {j}")
```

Çıktı:

 CC BY-NC-SA 4.0

```
i: 0, j: 0
i: 0, j: 2
i: 1, j: 0
i: 1, j: 2
i: 2, j: 0
i: 2, j: 2
```

8.1.2 Matrisler Üzerinde Dolaşmak

Elemanları da birer liste olan bir liste, satırları ve sütunları olan bir tabloyu, yani bir **matrisi** temsil eder; matrisin elemanları iç içe bir döngüyle dolaşılır.
`matris = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]` listesinde `matris[0]` birinci satırdır, `matris[1][2]` ikinci satırın üçüncü elemanı, yani 6'dır. Dıştaki döngü satırları, içteki döngü bir satırın elemanlarını dolaşır:

```
matris = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
for satir in matris:
    for eleman in satir:
        print(eleman, end=" ")
    print()
print(matris[1][2])
```

Çıktı:

```
1 2 3
4 5 6
7 8 9
6
```

İçteki döngüye bir koşul eklenerek belli elemanlar seçilebilir; `enumerate` ise satır ve sütun numaralarını da verir.

```
matris = [[1, 24, 3], [4, 5, 6], [7, 18, 9]]
for i, satir in enumerate(matris):
    for j, eleman in enumerate(satir):
        if eleman % 2 == 0:
            print(f"Satır {i}, sütun {j}: {eleman} çift bir sayıdır.")
```

Çıktı:

```
Satır 0, sütun 1: 24 çift bir sayıdır.
Satır 1, sütun 0: 4 çift bir sayıdır.
Satır 1, sütun 2: 6 çift bir sayıdır.
Satır 2, sütun 1: 18 çift bir sayıdır.
```

Örnek 8.2 (İki Listenin Bütün Çiftleri). Aşağıdaki programın çıktısını bulunuz.

```
birinci = []
ikinci = []
for x in [2, 3, 4]:
    for y in [3, 4, 7]:
        birinci.append(x + y)
        ikinci.append(x * y)
print("birinci:", birinci)
print("ikinci:", ikinci)
```

> **Çözüm.**
> Dıştaki döngünün her `x` değeri için `y` sırayla 3, 4, 7 olur; toplam $3 \cdot 3 = 9$ çift vardır. `x = 2` için toplamlar 5, 6, 9, çarpımlar 6, 8, 14; `x = 3` için toplamlar 6, 7, 10, çarpımlar 9, 12, 21; `x = 4` için toplamlar 7, 8, 11, çarpımlar 12, 16, 28'dir.

 CC BY-NC-SA 4.0

```
birinci = []
ikinci = []
for x in [2, 3, 4]:
    for y in [3, 4, 7]:
        birinci.append(x + y)
        ikinci.append(x * y)
print("birinci:", birinci)
print("ikinci:", ikinci)
```

Çıktı:

```
birinci: [5, 6, 9, 6, 7, 10, 7, 8, 11]
ikinci: [6, 8, 14, 9, 12, 21, 12, 16, 28]
```

■

8.2 while Döngüsü

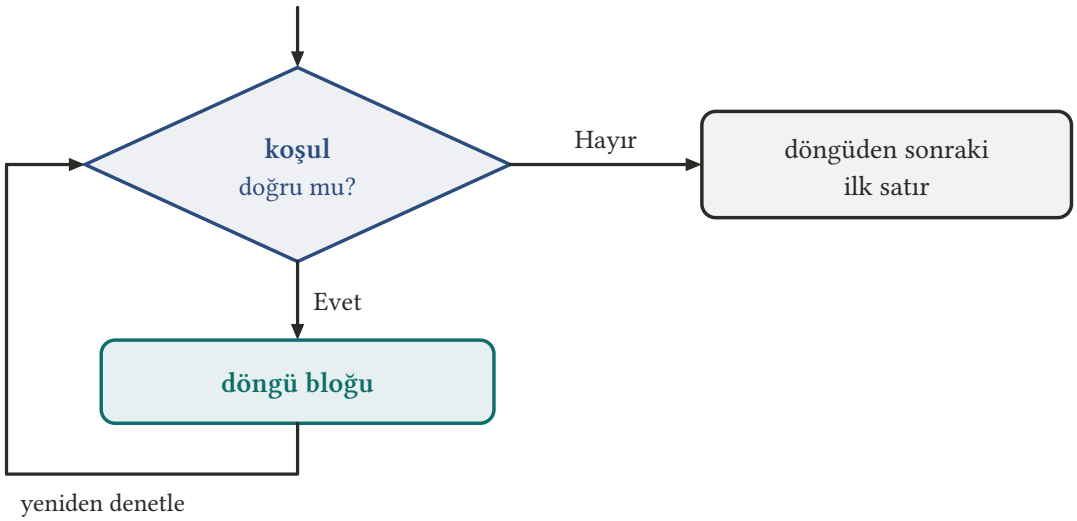
`for` döngüsü bir koleksiyonun elemanları bitince durur; `while` döngüsü ise bir koşul doğru olduğu sürece döner.

Tanım 8.2 (while Döngüsü).

```
while koşul:
    koşul doğru olduğu sürece çalışacak komutlar
```

yapısında önce `koşul` denetlenir. Koşul `True` ise girintili komutlar çalışır ve koşul yeniden denetlenir; bu, koşul `False` olana kadar sürer. Koşul ilk denetlemede `False` ise blok hiç çalışmaz.

Yani `while` "koşul sağlandıkça tekrarla" demektir. Döngünün bir gün bitmesi için bloğun içinde koşulu eninde sonunda `False` yapacak bir değişiklik bulunmalıdır; genellikle bu, koşulda kullanılan bir sayacın artırılmasıdır.



Şekil 8.1: Bir `while` döngüsünün akışı. Koşul her yinelemeden önce yeniden denetlenir; doğru olduğu sürece blok çalışır ve akış koşula geri döner. Koşul yanlış çıktığında döngüden çıkılır.

 CC BY-NC-SA 4.0

```
sayi = 1
while sayi <= 5:
    print(sayi)
    sayi += 1
print("Döngüden sonra sayi =", sayi)
```

Çıktı:

```
1
2
3
4
5
Döngüden sonra sayi = 6
```

`sayi` 5 iken blok son kez çalıştı ve `sayi`'yı 6 yaptı. $6 \leq 5$ yanlış olduğu için döngü bitti; döngüden sonra `sayi`'nın değeri 6'dır.

Örnek 8.3 (Bir while Döngüsünü İzlemek). Aşağıdaki programın çıktısını bulunuz.

```
sayac = 0
toplam = 0
while sayac <= 10:
    sayac = sayac + 2
    print("sayac:", sayac)
    toplam = toplam + sayac
print("toplam:", toplam)
```

Çözüm.
Koşul, sayaç **artırılmadan önce** denetlenir. `sayac` $0, 2, 4, 6, 8, 10$ iken koşul doğrudur ve blok çalışır; her seferinde önce `sayac` 2 artar, sonra yazılır ve toplama eklenir. `sayac` 10 iken blok son kez çalışır ve `sayac`'ı 12 yapar; $12 \leq 10$ yanlış olduğundan döngü biter. Toplam $2 + 4 + 6 + 8 + 10 + 12 = 42$'dir.

```
sayac = 0
toplam = 0
while sayac <= 10:
    sayac = sayac + 2
    print("sayac:", sayac)
    toplam = toplam + sayac
print("toplam:", toplam)
```

Çıktı:

```
sayac: 2
sayac: 4
sayac: 6
sayac: 8
sayac: 10
sayac: 12
toplam: 42
```

12'nin de yazılıp toplama eklenmesi şaşırtıcı görünebilir: koşul ancak bir sonraki denetlemede, yani blok bittikten sonra yeniden sorulur. ■

Bir `for` döngüsüyle yapılan her iş, sayacı kendimiz yöneterek bir `while` döngüsüyle de yapılabilir. 1'den 100'e kadar sayıların toplamı şöyle hesaplanır:

 CC BY-NC-SA 4.0

```
toplam = 0
i = 1
while i <= 100:
    toplam = toplam + i
    i = i + 1
print("Sayıların toplamı:", toplam)
```

Çıktı:

```
Sayıların toplamı: 5050
```

Burada `for i in range(1, 101):` daha kısa ve hatasızdır; `while`, tekrar sayısının önceden bilinmediği işlerde gerçekten gerekir.

> **i** for mu, while mı?
>
> Bir koleksiyonun elemanları ya da belli bir sayı aralığı dolaşılacaksa `for` kullanılır: tekrar sayısı baştan bellidir. Döngünün ne zaman biteceği çalışma sırasında ortaya çıkıyorsa, örneğin kullanıcının girdiğine ya da bir hesabın sonucuna bağlıysa, `while` kullanılır.

8.2.1 while Döngüsünde break ve continue

`break` ve `continue` komutları `while` döngüsünde de `for` döngüsündeki (Tanım 7.3) gibi çalışır.

```
sayi = 1
while sayi <= 10:
    if sayi == 5:
        break
    print(sayi, end=" ")
    sayi += 1
print()
sayi = 1
while sayi < 10:
    sayi += 1
    if sayi % 2 != 0:
        continue
    print(sayi, end=" ")
print()
```

Çıktı:

```
1 2 3 4
2 4 6 8 10
```

İkinci döngüde sayaç, `continue`'dan **önce** artırılıyor. Sayaç `continue`'dan sonra artırılsaydı, `continue` artırma satırını da atlardı ve sayaç bir daha hiç değişmezdi. Aşağıdaki programda bu yüzden 5 için sayaç ayrıca artırılıyor:

```
a = 0
while a < 10:
    if a == 5:
        a += 1
        continue
    print(a, end=" ")
    a += 1
print()
```

Çıktı:

```
0 1 2 3 4 6 7 8 9
```

 CC BY-NC-SA 4.0

`if` bloğundaki `a += 1` satırı silinirse `a` 5'te takılır: koşul hep doğru kalır, `continue` her seferinde artırma satırını atlar ve döngü hiç bitmez.

8.2.2 Sonsuz Döngü

Koşulu hiçbir zaman `False` olmayan bir döngü durmadan çalışır.

Tanım 8.3 (Sonsuz Döngü). Koşulu hiçbir zaman yanlış olmadığı için kendiliğinden bitmeyen döngüye **sonsuz döngü** denir. `while True:` yazımı bilerek kurulmuş bir sonsuz döngüdür ve ancak bloğun içindeki bir `break` ile sona erer.

Yani sonsuz döngü bazen bir hatadır, bazen de bilinçli bir tercih: bitiş koşulu döngünün başında değil ortasında denetlenecekse `while True:` yazılır ve uygun yerde `break` kullanılır. Yanlışlıkla bir sonsuz döngü başlatılırsa Colab'da hücrenin solundaki durdurma düğmesine, VS Code'da terminalin durdurma düğmesine basılır ya da terminalde **Ctrl + C** tuşlanır.

```
while True:
    girdi = input("Çıkmak için 'exit' yazın: ")
    if girdi.lower() == "exit":
        print("Döngü sonlandırılıyor.")
        break
    print("Devam ediliyor...")
```

Çıktı:

```
Çıkmak için 'exit' yazın: devam
Devam ediliyor...
Çıkmak için 'exit' yazın: Exit
Döngü sonlandırılıyor.
```

`girdi.lower()` metni küçük harflere çevirir; böylece kullanıcı `EXIT` ya da `Exit` yazsa da döngü biter. Metinlerin buna benzer başka metotları da vardır: `upper()` bütün harfleri büyütür, `capitalize()` yalnız ilk harfi büyük yapar.

Örnek 8.4 (Sayı Tahmin Oyunu). Program 1 ile 100 arasında bir sayı tutsun. Kullanıcı sayıyı bulana kadar tahmin alsın; her yanlış tahminde "daha büyük" ya da "daha küçük" ipucu versin ve sonunda kaç tahminde bulunduğunu yazsın.

Çözüm.
Kullanıcının kaç tahmin yapacağı önceden bilinmediği için `while True:` ile bir sonsuz döngü kurarız ve doğru tahminde `break` ile çıkarız. Her tahminde bir sayaç artırılır. Tutulan sayı şimdilik sabit olarak 45 seçildi; Rastgele Sayılar bölümünde onu bilgisayara rastgele seçtireceğiz.

```
tutulan = 45
sayac = 0
print("1-100 arası bir sayı tuttum, tahmin et!")
while True:
    sayac += 1
    tahmin = int(input("Tahminin: "))
    if tahmin > tutulan:
        print("Daha küçük bir sayı girmelisin.")
    elif tahmin < tutulan:
        print("Daha büyük bir sayı girmelisin.")
    else:
        print("Tebrikler, tuttuğum sayıyı bildin!")
        break
print(f"{sayac} tahminde buldun.")
```

Çıktı:

 CC BY-NC-SA 4.0

```
1-100 arası bir sayı tuttum, tahmin et!
Tahminin: 50
Daha küçük bir sayı girmelisin.
Tahminin: 25
Daha büyük bir sayı girmelisin.
Tahminin: 37
Daha büyük bir sayı girmelisin.
Tahminin: 43
Daha büyük bir sayı girmelisin.
Tahminin: 46
Daha küçük bir sayı girmelisin.
Tahminin: 44
Daha büyük bir sayı girmelisin.
Tahminin: 45
Tebrikler, tuttuğum sayıyı bildin!
7 tahminde buldun.
```

Kullanıcı her seferinde kalan aralığın ortasındaki sayıyı tahmin ederse ($50, 25, 37, \ldots$), aralık her adımda yarıya iner ve 1–100 arasındaki her sayı en çok 7 tahminde bulunur, çünkü $2^7 = 128 \geq 100$'dür. ■

Aynı oyun bir koşullu `while` döngüsüyle de yazılabilir: döngü, tahmin tutulan sayıya eşit olmadığı sürece döner. Döngüye ilk kez girebilmek için tahmine başlangıçta, "henüz değer yok" anlamına gelen `None` değeri verilir.

```
tutulan = 5
tahmin = None
deneme = 0
while tahmin != tutulan:
    tahmin = int(input("Tahmininiz: "))
    deneme += 1
    if tahmin < tutulan:
        print("Daha yüksek bir sayı tahmin edin.")
    elif tahmin > tutulan:
        print("Daha düşük bir sayı tahmin edin.")
print(f"Tebrikler! {deneme} denemede doğru tahmin ettiniz.")
```

Çıktı:

```
Tahmininiz: 10
Daha düşük bir sayı tahmin edin.
Tahmininiz: 3
Daha yüksek bir sayı tahmin edin.
Tahmininiz: 5
Tebrikler! 3 denemede doğru tahmin ettiniz.
```

8.2.3 Geçerli Bir Girdi Alana Kadar Sormak

Girdi ve Çıktı İşlemleri bölümünde `try-except` ile hatalı bir girdiyi yakalayıp bir uyarı yazmıştık; bir `while True` döngüsüyle birleştirildiğinde program, kullanıcı geçerli değerler girene kadar soruyu yeniden sorar.

```
while True:
    try:
        a = float(input("1. sayıyı gir: "))
        b = float(input("2. sayıyı gir: "))
        print(f"Girdiğiniz sayılar: {a} ve {b}")
        print("Toplam =", a + b)
        break
```

 CC BY-NC-SA 4.0

```
    except ValueError:
        print("Hata: Lütfen geçerli bir sayı girin! Tekrar deneyin.")
```

Çıktı:

```
1. sayıyı gir: a
Hata: Lütfen geçerli bir sayı girin! Tekrar deneyin.
1. sayıyı gir: 85
2. sayıyı gir: b
Hata: Lütfen geçerli bir sayı girin! Tekrar deneyin.
1. sayıyı gir: 5
2. sayıyı gir: 6
Girdiğiniz sayılar: 5.0 ve 6.0
Toplam = 11.0
```

`break`, iki dönüşüm de başarılı olduğunda çalışır. Bir dönüşüm hata verirse `break`'e hiç ulaşılmaz, `except` bloğu çalışır ve döngü baştan başlar.

Örnek 8.5 (Kullanıcının Girdiği Notların Ortalaması). Kullanıcıdan, `q` yazana kadar not alan ve sonunda girilen notların ortalamasını yazan bir program yazınız.

Çözüm.
Kaç not girileceği bilinmediği için `while True:` kullanırız. Her girdi önce `q` ile karşılaştırılır; `q` değilse sayıya çevrilip toplama eklenir ve not sayısı bir artırılır. Hiç not girilmezse sıfıra bölmemek için sonda `adet`'i denetleriz.

```
toplam = 0
adet = 0
while True:
    girdi = input("Notu girin (çıkmak için q): ")
    if girdi == "q":
        break
    toplam += float(girdi)
    adet += 1
if adet > 0:
    print(f"Notların ortalaması: {toplam / adet:.2f}")
else:
    print("Hiç not girilmedi.")
```

Çıktı:

```
Notu girin (çıkmak için q): 45
Notu girin (çıkmak için q): 63.5
Notu girin (çıkmak için q): 85
Notu girin (çıkmak için q): 70.56
Notu girin (çıkmak için q): q
Notların ortalaması: 66.02
```

Girilen notların toplamı $45 + 63{,}5 + 85 + 70{,}56 = 264{,}06$, ortalaması $264{,}06/4 = 66{,}015 \approx 66{,}02$'dir. ■

Örnek 8.6 (while ile Fibonacci Dizisi). Fibonacci dizisinin ilk 15 terimini bir `while` döngüsüyle yazdırınız.

Çözüm.
Dizinin ardışık iki terimini `a` ve `b` değişkenlerinde tutarız. Her adımda `a`'yı yazarız ve takastaki (Tanım 2.10) gibi bir çoklu atamayla `a, b = b, a + b` yaparız: yeni `a` eski `b`, yeni `b` ise eski iki terimin toplamı olur.

```
n = 15
a, b = 0, 1
sayac = 0
while sayac < n:
    print(a, end=" ")
    a, b = b, a + b
    sayac += 1
print()
```

Çıktı:

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

`a, b = b, a + b` yerine `a = b` ve `b = a + b` diye iki ayrı satır yazılsaydı, ikinci satırdaki `a` zaten değişmiş olurdu ve dizi yanlış çıkardı. ■

8.3 match-case Yapısı

Bir değişkenin değerine göre çok sayıda durumdan birinin seçileceği programlarda uzun bir `if-elif-else` zinciri yazmak yerine `match-case` yapısı kullanılabilir.
Haftanın gününü sıra numarasından bulan bir program `if-elif-else` ile şöyle yazılır:

```
gun = 4
if gun == 1:
    print("Pazartesi")
elif gun == 2:
    print("Salı")
elif gun == 3:
    print("Çarşamba")
elif gun == 4:
    print("Perşembe")
elif gun == 5:
    print("Cuma")
elif gun == 6:
    print("Cumartesi")
elif gun == 7:
    print("Pazar")
else:
    print("Geçersiz gün numarası")
```

Çıktı:

```
Perşembe
```

Her satırda `gun ==` yazmak hem uzun hem de hataya açıktır.

Tanım 8.4 (match-case Yapısı).

```
match deger:
    case secenek1:
        deger secenek1 ise çalışacak komutlar
    case secenek2:
        deger secenek2 ise çalışacak komutlar
    case _:
        hiçbir seçenek tutmazsa çalışacak komutlar
```

yapısında `deger` sırayla `case` satırlarındaki seçeneklerle karşılaştırılır ve eşleşen **ilk** seçeneğin bloğu çalışır. `case _:` her değerle eşleşir ve `if-elif-else`'teki `else`'in görevini görür. Birden çok seçenek `|` ile birleştirilebilir: `case 6 | 7:`. Bu yapı Python 3.10 ile dile eklenmiştir.

 CC BY-NC-SA 4.0

Yani `match-case`, "değer şuysa şunu, buysa bunu yap" listesidir. Burada _, alt çizgi değişkeninde (Tanım 7.4) olduğu gibi "ne olursa olsun" anlamına gelir.

```
gun = 6
match gun:
    case 1:
        print("Pazartesi")
    case 2:
        print("Salı")
    case 3:
        print("Çarşamba")
    case 4:
        print("Perşembe")
    case 5:
        print("Cuma")
    case 6 | 7:
        print("Hafta sonu")
    case _:
        print("Geçersiz gün numarası")
```

Çıktı:

```
Hafta sonu
```

`match-case` değerleri yalnız eşitlik açısından karşılaştırır. "Yaş 13'ten küçükse" gibi aralık koşulları için `if-elif-else` daha uygundur.

Örnek 8.7 (Hava Durumuna Göre Öneri). Kullanıcıdan bugünkü hava durumunu (Güneşli, Yağmurlu, Karlı, Rüzgârlı) alıp bir giysi önerisi yazan bir program yazınız. Kullanıcı sözcüğü küçük harfle yazsa da program çalışmalıdır.

Çözüm.
Girdiyi `capitalize()` ile ilk harfi büyük, gerisi küçük olacak biçime getiririz; böylece `yağmurlu` ya da `YAĞMURLU` girdileri de `Yağmurlu` olur ve `case` seçenekleriyle eşleşir.

```
hava = input("Bugünkü hava durumu nedir? ").capitalize()
match hava:
    case "Güneşli":
        print("Güneş kremi sürmeyi unutmayın!")
    case "Yağmurlu":
        print("Şemsiye almayı unutmayın.")
    case "Karlı":
        print("Kalın bir mont giyin.")
    case "Rüzgârlı":
        print("Bir rüzgârlık almanız iyi olur.")
    case _:
        print("Geçersiz hava durumu girdiniz.")
```

Çıktı:

```
Bugünkü hava durumu nedir? yağmurlu
Şemsiye almayı unutmayın.
```

■

8.4 Çeşitli Örnekler

Aşağıdaki kısa programların çıktısını önce kâğıt üzerinde, değişkenlerin değerlerini adım adım izleyerek bulmaya çalışın.

Örnek 8.8 (Koşula Bağlı Güncelleme). Aşağıdaki programın çıktısını bulunuz.

 CC BY-NC-SA 4.0

```
a, b, toplam = 0, 5, 20
while a < 5:
    if a + b > 6:
        toplam = a + b
        toplam += 1
    print(toplam)
    a += 1
```

Çözüm.
Döngü `a` 0'dan 4'e kadar beş kez çalışır. `a + b > 6` koşulu `a = 0` (5) ve `a = 1` (6) için yanlıştır; `toplam` değişmez ve 20 yazılır. `a = 2, 3, 4` için koşul doğrudur: `toplam` önce `a + b`, sonra bir fazlası olur ve sırasıyla 8, 9, 10 yazılır.

```
a, b, toplam = 0, 5, 20
while a < 5:
    if a + b > 6:
        toplam = a + b
        toplam += 1
    print(toplam)
    a += 1
```

Çıktı:

```
20
20
8
9
10
```

■

Örnek 8.9 (Tek Sayıların Toplamı). 1'den 9'a kadar tek sayıların toplamını bir `while` döngüsüyle hesaplayınız ve ara toplamları yazdırınız. Aynı toplamı `continue` kullanan bir döngüyle de bulunuz.

Çözüm.
`a`'yı 1'den başlatıp her adımda 2 artırırsak yalnız tek sayıları dolaşırız. Ara toplamlar $1, 1+3=4, 4+5=9, 16, 25$ olur; ardışık tek sayıların toplamlarının tam kare çıkması tesadüf değildir: $1+3+\cdots+(2k-1)=k^2$.

```
toplam = 0
a = 1
while a < 10:
    toplam = toplam + a
    a = a + 2
    print(toplam, end=" ")
print()
```

Çıktı:

```
1 4 9 16 25
```

İkinci yolda bütün sayıları dolaşır, çiftleri `continue` ile atlarız. Sayaç, `continue`'dan önce artırıldığı için döngü takılmaz:

 CC BY-NC-SA 4.0

```
toplam = 0
a = 0
while a < 10:
    a = a + 1
    if a % 2 == 0:
        continue
    toplam = toplam + a
print(toplam)
```

Çıktı:

```
25
```

Aynı fikirle 1'den 99'a kadar tek sayıların toplamı $50^2 = 2500$ bulunur. ■

Örnek 8.10 (Üçün Katlarının Ortalaması). Kullanıcının girdiği bir N sayısına kadar (dahil) 3'e bölünebilen pozitif sayıların ortalamasını bulan bir program yazınız. $N = 16$ için ekrana $(3+6+9+12+15)/5 = 9{,}0$ yazılmalıdır.

Çözüm.
N'den büyük olmayan en büyük 3 katı $X = N - (N \bmod 3)$'tür: $N = 16$ için $16 - 1 = 15$. Bu sayıdan başlayıp 3'er azaltarak 0'a kadar gideriz; her adımda sayıyı toplama ekler, sayacı artırırız.

```
N = int(input("Bir sayı girin: "))
X = N - (N % 3)
T = 0
sayac = 0
while X > 0:
    T = T + X
    X = X - 3
    sayac = sayac + 1
print("Ortalama =", T / sayac)
```

Çıktı:

```
Bir sayı girin: 16
Ortalama = 9.0
```

Kullanıcı 1 ya da 2 girerse $X = 0$ olur, döngü hiç çalışmaz ve `sayac` 0 kalır; bu durumda son satır sıfıra bölme hatası verir. Programı bu durumlara karşı korumak için son satırdan önce `sayac`'ın sıfır olup olmadığına bakan bir `if` eklenmelidir. ■

Örnek 8.11 (while ile Faktöriyel). Aşağıdaki program ne hesaplar? Çıktısını bulunuz.

```
n = 5
f = 1
i = n
while i >= 1:
    f = f * i
    i = i - 1
print(f)
```

Çözüm.

 CC BY-NC-SA 4.0

`i`, `n`'den başlayıp 1'e kadar azalır ve her adımda `f` ile çarpılır: $f = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$. Program $n! = 5!$ değerini, çarpanları büyükten küçüğe alarak hesaplar.

```
n = 5
f = 1
i = n
while i >= 1:
    f = f * i
    i = i - 1
print(f)
```

Çıktı:

```
120
```

■

8.5 Alıştırmalar

`while` döngüsü kurarken her seferinde döngünün bitmesini sağlayan değişikliği bulun.

Alıştırma 8.1 (Rakamlar Toplamı). Bir pozitif tam sayının rakamlarının toplamını metne çevirmeden, `%` ve `//` kullanan bir `while` döngüsüyle bulunuz. $n = 2026$ için deneyiniz.

Çözüm.
Son rakam `n % 10` ile alınır, `n //= 10` ise son rakamı atar. Bu iki adımı `n` sıfır olana kadar tekrarlarız. 2026 için sırayla 6, 2, 0, 2 rakamları alınır ve toplam 10 olur.

```
n = 2026
toplam = 0
while n > 0:
    toplam += n % 10
    n //= 10
print(toplam)
```

Çıktı:

```
10
```

Döngü her adımda `n`'yi küçülttüğü için mutlaka biter. ■

Alıştırma 8.2 (Öklid Algoritmasıyla EBOB). a ve b pozitif tam sayıları için $\text{ebob}(a, b) = \text{ebob}(b, a \bmod b)$ ve $\text{ebob}(a, 0) = a$ eşitliklerini kullanarak $\text{ebob}(252, 105)$ değerini bir `while` döngüsüyle bulunuz ve `math.gcd` ile karşılaştırınız.

Çözüm.
`b` sıfır olmadığı sürece `a, b = b, a % b` çoklu atamasını yaparız; `b` sıfır olduğunda EBOB `a`'dadır. Adımlar:

- $(252, 105)$: $252 = 2 \cdot 105 + 42$, yeni çift $(105, 42)$.
- $(105, 42)$: $105 = 2 \cdot 42 + 21$, yeni çift $(42, 21)$.
- $(42, 21)$: $42 = 2 \cdot 21 + 0$, yeni çift $(21, 0)$.

```
import math
a, b = 252, 105
```

 CC BY-NC-SA 4.0

```
while b != 0:
    a, b = b, a % b
    print(a, b)
print("EBOB:", a, "- math.gcd:", math.gcd(252, 105))
```

Çıktı:

```
105 42
42 21
21 0
EBOB: 21 - math.gcd: 21
```

■

Alıştırma 8.3 (Bini Aşan İlk İkinin Kuvveti). $2^k \geq 1000$ olan en küçük k doğal sayısını bir `while` döngüsüyle bulunuz.

Çözüm.
Kaç adım gerektiği önceden bilinmediği için `while` uygundur. `kuvvet`'i $1 = 2^0$ ile başlatır ve 1000'den küçük olduğu sürece ikiyle çarpıp `k`'yı artırırız. $2^9 = 512 < 1000 \leq 1024 = 2^{10}$ olduğundan $k = 10$ bulunmalıdır.

```
k = 0
kuvvet = 1
while kuvvet < 1000:
    kuvvet *= 2
    k += 1
print(k, kuvvet)
```

Çıktı:

```
10 1024
```

■

Alıştırma 8.4 (Sayıyı Tersine Çevirmek). $n = 12345$ sayısının rakamlarını ters sırada yazarak elde edilen 54321 sayısını bir `while` döngüsüyle bulunuz.

Çözüm.
Her adımda `n`'nin son rakamını alır ve `ters` sayısının sağına ekleriz: `ters = 10 * ters + son_rakam`. `ters` sırayla 5, 54, 543, 5432, 54321 olur.

```
n = 12345
ters = 0
while n > 0:
    ters = 10 * ters + n % 10
    n //= 10
print(ters)
```

Çıktı:

```
54321
```

■

 CC BY-NC-SA 4.0

Alıştırma 8.5 (Sayı Üçgeni). İç içe iki döngüyle aşağıdaki deseni yazdırınız:

```
1
1 2
1 2 3
1 2 3 4
```

Çözüm.
i. satırda ($i = 1, 2, 3, 4$) 1'den i'ye kadar sayılar yazılır. Dıştaki döngü satır numarasını, içteki döngü o satırdaki sayıları dolaşır; içteki döngünün bitişi dıştaki döngünün değişkenine bağlıdır.

```
for i in range(1, 5):
    for j in range(1, i + 1):
        print(j, end=" ")
    print()
```

Çıktı:

```
1
1 2
1 2 3
1 2 3 4
```

■

Alıştırma 8.6 (while ile Asallık Denemesi). Aşağıdaki program ne yapar? Çıktısını bulunuz.

```
for sayi in range(2, 21):
    bolen = 2
    asal = True
    while bolen < sayi:
        if sayi % bolen == 0:
            asal = False
            break
        bolen += 1
    if asal:
        print(sayi, end=" ")
print()
```

Çözüm.
Dıştaki `for` 2'den 20'ye kadar sayıları dolaşır. Her sayı için içteki `while`, 2'den sayının bir eksiğine kadar bölen adaylarını dener; bir bölen bulunursa `asal` `False` yapılır ve `break` yalnız içteki `while`'dan çıkar. Hiç bölen bulunmazsa sayı asaldır ve yazılır. Program 20'ye kadar asal sayıları yazar: $2, 3, 5, 7, 11, 13, 17, 19$.

```
for sayi in range(2, 21):
    bolen = 2
    asal = True
    while bolen < sayi:
        if sayi % bolen == 0:
            asal = False
            break
        bolen += 1
    if asal:
```

 CC BY-NC-SA 4.0

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

Çıktı:

```
2 3 5 7 11 13 17 19
```

`sayi = 2` için `2 < 2` yanlış olduğundan `while` hiç çalışmaz ve 2 asal kalır. ■

Alıştırma 8.7 (match-case ile Harf Notundan Puana). Kullanıcının girdiği harf notunu (A, B, C, D, F) 4'lük sistemdeki puanına çeviren bir program yazınız: A 4, B 3, C 2, D 1, F 0. Küçük harf de kabul edilsin.

Çözüm.
Girdiyi `upper()` ile büyük harfe çevirir, `match-case` ile beş seçeneği ve geçersiz girdiyi ayırırız. Kullanıcı `b` girsin:

```
harf = input("Harf notu: ").upper()
match harf:
    case "A":
        puan = 4
    case "B":
        puan = 3
    case "C":
        puan = 2
    case "D":
        puan = 1
    case "F":
        puan = 0
    case _:
        puan = None
if puan is None:
    print("Geçersiz harf notu.")
else:
    print(f"{harf} notunun puanı: {puan}")
```

Çıktı:

```
Harf notu: b
B notunun puanı: 3
```

`puan is None` ifadesi, `puan`'ın "değer yok" anlamındaki `None` olup olmadığını sorar; `None` ile karşılaştırmada `==` yerine `is` kullanılması bir alışkanlıktır. ■

Döngü araçlarımız artık tamamlandı: `for` ile koleksiyonları, iç içe döngülerle tabloları, `while` ile sayısı belli olmayan tekrarları yönetebiliyoruz. Bir sonraki bölüm Demetler, Koleksiyonlar ve Listeler bölümünde kısaca tanıdığımız değiştirilemez koleksiyonu ayrıntılı olarak ele alıyor.

 CC BY-NC-SA 4.0

9. Demetler

Koleksiyonlar ve Listeler bölümünde demeti, elemanları sıralı ama değiştirilemeyen bir koleksiyon olarak tanımıştık (Tanım 6.2). İlk bakışta listenin yapabildiği her şeyi yapamayan, kısıtlı bir liste gibi görünür. Oysa bu kısıt bir güvencedir: bir noktanın koordinatlarını, bir tarihi ya da bir öğrencinin numarasıyla adını bir demette tuttuğumuzda, programın başka bir yerinde bu değerlerin yanlışlıkla değiştirilmeyeceğinden emin oluruz.
Bu bölümde demet oluşturmanın yollarını, elemanlarına erişmeyi ve demetlerle yapılabilen işlemleri göreceğiz. Bölümün sonunda, demete çok benzeyen ama ondan farklı bir yapı olan üreteç ifadelerini tanıyacağız.

9.1 Demet Oluşturmak

Bir demet, elemanları normal parantez içinde ve virgülle ayrılarak yazılarak oluşturulur.
Listelerde olduğu gibi demetin elemanları farklı tiplerde olabilir ve aynı eleman birden çok kez bulunabilir. `tuple` fonksiyonu ise bir koleksiyonun ya da bir metnin elemanlarından bir demet oluşturur; bir metin verildiğinde her karakter ayrı bir eleman olur.

```
demet1 = ("elma", "armut", "kiraz", "kiraz")
demet2 = (1, 2, 3, 4, 5)
demet3 = (1, "a", 3.5, 2.5)
print(demet1)
print(demet2)
print(demet3)
print(type(demet3))
print(tuple("istatistik"))
print(tuple(range(5)))
print(tuple([1, 2, 3]))
```

Çıktı:

```
('elma', 'armut', 'kiraz', 'kiraz')
(1, 2, 3, 4, 5)
(1, 'a', 3.5, 2.5)
<class 'tuple'>
('i', 's', 't', 'a', 't', 'i', 's', 't', 'i', 'k')
(0, 1, 2, 3, 4)
(1, 2, 3)
```

Demet oluşturan asıl işaret parantez değil virgüldür: parantezsiz yazılmış virgüllü bir değer dizisi de bir demettir.

Tanım 9.1 (Paketleme ve Açma). `x = 3, 7, 10` atamasında sağdaki değerler bir demette toplanır ve `x` bu demet olur; buna **demet paketleme** (packing) denir. `a, b, c = x` atamasında ise demetin elemanları sırayla değişkenlere dağıtılır; buna **demet açma** (unpacking) denir. Açmada değişken sayısı demetin eleman sayısına eşit olmalıdır.

Yani Değişkenler, Veri Tipleri ve Atama bölümünde gördüğümüz çoklu atama (Tanım 2.9) aslında bir demet işlemidir: `x, y, z = 1, 2, 3` satırında sağ taraf bir demete paketlenir, sonra sol taraftaki değişkenlere açılır. Takas (Tanım 2.10) da aynı yolla çalışır.

```
x = 3, 7, 10
print(x, type(x))
a, b, c = x
print(a, b, c)
```

 CC BY-NC-SA 4.0

```
gun, ay, yil = (4, 11, 2026)
print(f"{gun}.{ay}.{yil}")
```

Çıktı:

```
(3, 7, 10) <class 'tuple'>
3 7 10
4.11.2026
```

9.1.1 Tek Elemanlı ve Boş Demet

Parantezler matematikte işlemleri gruplamak için de kullanıldığından tek elemanlı bir demet yazarken dikkat etmek gerekir.
`('Ali')` yazımı bir demet değildir; parantez yalnız gruplama yapar ve değer `'Ali'` metninin kendisidir. Tek elemanlı bir demet için elemandan sonra bir virgül konur: `('Ali',)`. Hiç elemanı olmayan **boş demet** ise `()` diye yazılır.

```
tek_demet = ("Ali",)
bir_metin = ("Ali")
print(tek_demet, type(tek_demet))
print(bir_metin, type(bir_metin))
print((5), (5,))
bos = ()
print(bos, len(bos))
```

Çıktı:

```
('Ali',) <class 'tuple'>
Ali <class 'str'>
5 (5,)
() 0
```

9.2 Elemanlara Erişim ve Dilimleme

Demetin elemanlarına, listelerde olduğu gibi indeksle (Tanım 6.6) erişilir ve demetler aynı kurallarla dilimlenir (Tanım 6.7).
İlk elemanın indeksi 0, son elemanın indeksi -1'dir. `demet[a:b:c]` yazımı `a`'dan başlayıp `b`'ye varmadan `c`'şer atlayarak elemanları alır; bir demetten alınan dilim de yine bir demettir.

```
meyveler = ("elma", "armut", "kiraz", "kiraz")
print(meyveler[0])
print(meyveler[-1])
print(meyveler[1:3])
print(meyveler[1:4:2])
print(meyveler[::2])
print(meyveler[::-1])
```

Çıktı:

```
elma
kiraz
('armut', 'kiraz')
('armut', 'kiraz')
('elma', 'kiraz')
('kiraz', 'kiraz', 'armut', 'elma')
```

Örnek 9.1 (Bir Demetten Dilimler). `sifre = ("sıfır", "bir", "iki", "üç", "dört", 7, 8)` demetinden `sifre[1:4]`, `sifre[2:]`, `sifre[:2]` ve `sifre[::2]` dilimlerini bulunuz.

> **Çözüm.**

 CC BY-NC-SA 4.0

İndeksler 0'dan 6'ya kadardır: `"sıfır"` 0, `"bir"` 1, `"iki"` 2, `"üç"` 3, `"dört"` 4, `7` 5, `8` 6.

- `sifre[1:4]`: indeks 1, 2, 3: `("bir", "iki", "üç")`.
- `sifre[2:]`: indeks 2'den sona kadar: `("iki", "üç", "dört", 7, 8)`.
- `sifre[:2]`: baştan indeks 2'ye kadar, 2 hariç: `("sıfır", "bir")`.
- `sifre[::2]`: indeks 0, 2, 4, 6: `("sıfır", "iki", "dört", 8)`.

```
sifre = ("sıfır", "bir", "iki", "üç", "dört", 7, 8)
print(sifre[1:4])
print(sifre[2:])
print(sifre[:2])
print(sifre[::2])
```

Çıktı:

```
('bir', 'iki', 'üç')
('iki', 'üç', 'dört', 7, 8)
('sıfır', 'bir')
('sıfır', 'iki', 'dört', 8)
```

■

9.3 Demetler Değiştirilemez

Demetin listeden ayrıldığı tek ama önemli nokta, oluşturulduktan sonra değiştirilememesidir.

Tanım 9.2 (Değiştirilemez Nesne). Oluşturulduktan sonra içeriği değiştirilemeyen değerlere **değiştirilemez** (immutable) değerler denir. Demetler, metinler ve sayılar değiştirilemez; listeler, kümeler ve sözlükler **değiştirilebilir** (mutable) değerlerdir.

Yani bir demetin bir elemanına yeni bir değer atanamaz, demete eleman eklenemez ve demetten eleman çıkarılamaz. Bunlardan biri denenirse Python hata verir:

```
meyveler = ("elma", "armut", "kiraz")
meyveler[0] = "portakal"
```

Çıktı:

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

```
meyveler = ("elma", "armut", "kiraz")
meyveler.append("muz")
```

Çıktı:

```
AttributeError: 'tuple' object has no attribute 'append'
```

İkinci hata, demetlerin `append` diye bir metodu olmadığını söylüyor. Bir demette yine de değişiklik yapmak gerekirse demet önce `list` ile listeye çevrilir, liste değiştirilir ve sonuç `tuple` ile yeniden demete çevrilerek aynı ada atanır:

```
meyveler = ("elma", "armut", "kiraz", "kiraz")
liste = list(meyveler)
liste[2] = "karpuz"
liste.append("muz")
meyveler = tuple(liste)
print(meyveler)
```

Çıktı:

 CC BY-NC-SA 4.0

```
('elma', 'armut', 'karpuz', 'kiraz', 'muz')
```

Burada eski demet değiştirilmedi; yeni bir demet oluşturuldu ve `meyveler` adı artık bu yeni demeti gösteriyor. Aynı biçimde `demet = demet + (4,)` gibi bir atama da eski demeti değiştirmez, yeni bir demet oluşturur.

9.4 Demetlerle İşlemler

Demeti değiştirmeyen bütün liste işlemleri demetlerde de kullanılabilir.
`for` döngüsü (Tanım 7.1) demetin elemanlarını sırayla dolaşır; `in` operatörü (Tanım 6.8) bir değerin demette olup olmadığını sorar; `len` eleman sayısını verir. İki demet `+` ile birleştirilir, bir demet bir tam sayıyla çarpılırsa elemanları o kadar kez tekrarlanır. Sayılardan oluşan bir demetle `sum`, `min` ve `max` da kullanılabilir.

```
isimler = ("Ali", "Zeynep", "Nadir")
for x in isimler:
    print(x, end=" ")
print()
if "Nadir" in isimler:
    print("Nadir demette mevcuttur.")
print(len(isimler))
yaslar = (70, 40, 15)
print(isimler + yaslar)
print(yaslar * 2)
print(sum(yaslar), min(yaslar), max(yaslar))
```

Çıktı:

```
Ali Zeynep Nadir
Nadir demette mevcuttur.
3
('Ali', 'Zeynep', 'Nadir', 70, 40, 15)
(70, 40, 15, 70, 40, 15)
125 15 70
```

`sorted` bir demeti de sıralar, ama sonucu her zaman bir **liste** olarak verir; demetin kendisi değişmez.

```
sayilar = (5, 2, 9, 1)
print(sorted(sayilar))
print(tuple(sorted(sayilar)))
print(sayilar)
```

Çıktı:

```
[1, 2, 5, 9]
(1, 2, 5, 9)
(5, 2, 9, 1)
```

9.4.1 count ve index Metotları

Demetlerin yalnız iki metodu vardır ve ikisi de demeti değiştirmez.
`demet.count(x)` `x`'in demette kaç kez geçtiğini, `demet.index(x)` ise `x`'in ilk geçtiği yerin indeksini verir; `x` demette yoksa `index` `ValueError` verir. Aynı iki metot metinlerde de vardır.

```
canta = ("kalem", "silgi", "defter", "kitap", "kalem", "defter", "kalem")
print(canta.count("kalem"), canta.count("defter"))
meyveler = ("elma", "armut", "kiraz", "şeftali", "kiraz")
print(meyveler.index("kiraz"))
isim = "ArithmeticError"
print(isim.index("r"), isim.count("r"))
```

Çıktı:

 CC BY-NC-SA 4.0

```
3 2
2
1 4
```

`"ArithmeticError"` metninde `r` harfi ilk kez indeks 1'de geçer ve toplam 4 kez bulunur. Büyük ve küçük harfler farklı karakterler olduğu için metinde bir büyük `R` bulunsaydı sayılmazdı.

9.4.2 Demet mi, Liste mi?

Hangi durumda demet, hangi durumda liste kullanılacağına koleksiyonun içeriğinin değişip değişmeyeceği karar verir.

- **Liste:** eleman eklenip çıkarılacak ya da elemanlar değiştirilecekse. Bir sınıfın not listesi, bir alışveriş sepeti.
- **Demet:** içerik baştan belliyse ve değişmemesi gerekiyorsa. Bir noktanın koordinatları `(3, 4)`, bir tarih `(4, 11, 2026)`, bir rengin kırmızı–yeşil–mavi bileşenleri `(255, 128, 0)`.

Demetler değiştirilemez oldukları için bir sözlükte anahtar olarak kullanılabilir; listeler kullanılamaz. Ayrıca demetler, aynı elemanları taşıyan listelerden biraz daha az bellek kaplar.

9.5 Üreteç İfadeleri

Liste üretecinin (Tanım 7.7) köşeli parantezleri normal parantezle değiştirilince bir demet üreteci elde edileceği düşünülebilir; ama Python'da öyle bir şey yoktur.

Tanım 9.3 (Üreteç İfadesi). `(ifade for degisken in koleksiyon)` yazımına **üreteç ifadesi** (generator expression) denir. Üreteç ifadesi değerlerini bir koleksiyonda saklamaz; onları istendikçe tek tek üretir. Değerler bir kez üretildikten sonra tükenir: üreteç ikinci kez dolaşılmak istenirse hiçbir değer vermez.

Yani `(x ** 2 for x in range(5))` bir demet değil, kareleri sırayla üreten bir nesnedir. `range` gibi, değerlerini hepsini birden bellekte tutmadan verir. Değerleri bir demette toplamak için üreteç `tuple`'a, bir listede toplamak için `list`'e verilir.

```
kareler = (x ** 2 for x in range(5))
print(type(kareler))
print(tuple(kareler))
print(tuple(kareler))
```

Çıktı:

```
<class 'generator'>
(0, 1, 4, 9, 16)
()
```

İkinci `tuple(kareler)` boş bir demet verdi, çünkü üreteç ilk dönüşümde bütün değerlerini üretip tükenmişti. Aynı değerler yeniden gerekiyorsa üreteç ifadesi yeniden yazılır ya da sonuç baştan bir demette saklanır.

Örnek 9.2 (Tükenen Bir Üreteç). Aşağıdaki programın ikinci satırında neden yalnız küpler yazılır?

```
demet1 = (x ** 2 for x in range(5))
demet2 = (x ** 3 for x in range(5))
print(tuple(demet1))
print(tuple(demet1) + tuple(demet2))
```

> **Çözüm.**
> `demet1` ve `demet2` birer demet değil, birer üreteçtir. Birinci `print` `demet1`'in bütün değerlerini üretir ve `(0, 1, 4, 9, 16)` yazar; bu noktada `demet1` tükenmiştir. İkinci `print`'te `tuple(demet1)` boş demet `()`'i verir; `() + (0, 1, 8, 27, 64)` toplamı yalnız küplerden oluşur.

 CC BY-NC-SA 4.0

```
demet1 = (x ** 2 for x in range(5))
demet2 = (x ** 3 for x in range(5))
print(tuple(demet1))
print(tuple(demet1) + tuple(demet2))
```

Çıktı:

```
(0, 1, 4, 9, 16)
(0, 1, 8, 27, 64)
```

Kareler ile küplerin birleşimi isteniyorsa değerler baştan demetlere çevrilip saklanmalıdır: `d1 = tuple(x ** 2 for x in range(5))`. Bir demet ne kadar kullanılırsa kullanılsın tükenmez. ■

Üreteç ifadesinin en kullanışlı olduğu yer, değerlerin yalnız bir kez gerektiği `sum`, `max`, `min`, `any` ve `all` gibi fonksiyonlardır. Böyle bir fonksiyona tek argüman olarak verildiğinde üretecin kendi parantezleri yazılmayabilir.

```
print(sum(x ** 2 for x in range(1, 6)))
print(max(x % 7 for x in range(20, 30)))
sayilar = [-3, -2, -1, 0, 1, 2, 3, 4]
print(sum(x * x for x in sayilar if x > 0))
```

Çıktı:

```
55
6
30
```

Üreteç bir `for` döngüsüyle de dolaşılabilir; her yinelemede bir sonraki değer üretilir:

```
for sayi in (x ** 2 for x in range(5)):
    print(sayi, end=" ")
print()
```

Çıktı:

```
0 1 4 9 16
```

9.6 Alıştırmalar

Demetlerin değiştirilemediğini unutmadan, her alıştırmada sonucun tipine de dikkat edin.

Alıştırma 9.1 (Demet İşlemlerinin Çıktısı). `t = (1, 12, -3, "a", 7.2)` için `t[1:3]`, `t[-3:-1]`, `len(t)`, `2 in t` ve `t.index(-3)` değerlerini bulunuz.

Çözüm.
İndeksler: `1` 0, `12` 1, `-3` 2, `"a"` 3, `7.2` 4; negatif indekslerle `7.2` -1, `"a"` -2, `-3` -3'tür.

- `t[1:3]`: indeks 1 ve 2: `(12, -3)`.
- `t[-3:-1]`: indeks -3 ve -2: `(-3, 'a')`.
- `len(t)`: 5.
- `2 in t`: demette `2` yoktur, `False`.
- `t.index(-3)`: 2.

```
t = (1, 12, -3, "a", 7.2)
print(t[1:3])
print(t[-3:-1])
print(len(t))
```

 CC BY-NC-SA 4.0

```
print(2 in t)
print(t.index(-3))
```

Çıktı:

```
(12, -3)
(-3, 'a')
5
False
2
```

■

Alıştırma 9.2 (İki Nokta Arasındaki Uzaklık). `A = (1, 2)` ve `B = (4, 6)` noktaları demetlerle verilsin. İki nokta arasındaki uzaklığı demet açmayla bulunuz.

Çözüm.
Demetleri açarak koordinatları ayrı değişkenlere alırız. Uzaklık

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} = \sqrt{3^2 + 4^2} = \sqrt{25} = 5$$

'tir.

```
import math
A = (1, 2)
B = (4, 6)
x1, y1 = A
x2, y2 = B
print(math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2))
```

Çıktı:

```
5.0
```

■

Alıştırma 9.3 (Bir Demette Eleman Değiştirmek). `renk = (255, 128, 0)` demetinin ortadaki elemanını `64` yapınız.

Çözüm.
Demetler değiştirilemez; `renk[1] = 64` `TypeError` verir. Demeti listeye çevirir, listeyi değiştirir ve yeniden demete çeviririz. İkinci bir yol, dilimlerden yeni bir demet kurmaktır: ilk eleman, yeni değer ve son eleman birleştirilir.

```
renk = (255, 128, 0)
liste = list(renk)
liste[1] = 64
print(tuple(liste))
print(renk[:1] + (64,) + renk[2:])
```

Çıktı:

```
(255, 64, 0)
(255, 64, 0)
```

 CC BY-NC-SA 4.0

İkinci yolda `(64,)` yazımındaki virgül zorunludur: `(64)` bir sayıdır ve bir demete `+` ile eklenemez. ■

Alıştırma 9.4 (Bir İsmin Harfleri). `isim = "Zeynep"` metnini bir demete çevirip `e` harfinin kaç kez geçtiğini ve ilk kez hangi indekste bulunduğunu bulunuz.

Çözüm.
`tuple(isim)` her harfi ayrı bir eleman yapar: `('Z', 'e', 'y', 'n', 'e', 'p')`. `e` harfi indeks 1 ve 4'te, yani iki kez geçer; ilk indeksi 1'dir.

```
isim = "Zeynep"
harfler = tuple(isim)
print(harfler)
print(harfler.count("e"), harfler.index("e"))
```

Çıktı:

```
('Z', 'e', 'y', 'n', 'e', 'p')
2 1
```

■

Alıştırma 9.5 (Baş ve Son Eleman). `a, *_, b = 2, 7, 9, 8, 6` atamasından sonra `a + b` kaçtır? `_` neyi tutar?

Çözüm.
Sağ taraf `(2, 7, 9, 8, 6)` demetine paketlenir. Açmada `a` ilk elemanı, `b` son elemanı alır; yıldızlı `_` ise aradaki bütün elemanları bir liste olarak toplar. Böylece $a + b = 2 + 6 = 8$ ve `_` `[7, 9, 8]` olur.

```
a, *_, b = 2, 7, 9, 8, 6
print(a + b)
print(_)
```

Çıktı:

```
8
[7, 9, 8]
```

■

Alıştırma 9.6 (Üreteçten Liste). Aşağıdaki programın çıktısını bulunuz.

```
uretec = (2 * x ** 2 for x in range(5))
liste = []
for sayi in uretec:
    liste.append(sayi)
print(liste)
print(list(uretec))
```

Çözüm.
Üreteç $x = 0, 1, 2, 3, 4$ için $2x^2$ değerlerini, yani $0, 2, 8, 18, 32$'yi üretir; döngü bunları listeye ekler. Döngü bittiğinde üreteç tükenmiştir; `list(uretec)` bu yüzden boş liste verir.

 CC BY-NC-SA 4.0

```
uretec = (2 * x ** 2 for x in range(5))
liste = []
for sayi in uretec:
    liste.append(sayi)
print(liste)
print(list(uretec))
```

Çıktı:

```
[0, 2, 8, 18, 32]
[]
```

■

Alıştırma 9.7 (Liste, Demet ve Küme). `liste = [1, 5, 3, 9, 2, 5]` listesini bir demete ve bir kümeye çeviriniz. Hangisinin eleman sayısı farklıdır, neden?

Çözüm.
Demete dönüşüm elemanların sırasını ve tekrarlarını korur; eleman sayısı 6 kalır. Küme tekrar eden elemanları tek bir eleman olarak tuttuğu için iki `5`'ten biri düşer ve eleman sayısı 5 olur. Kümenin sırası olmadığından onu sıralı göstermek için `sorted` kullanırız.

```
liste = [1, 5, 3, 9, 2, 5]
demet = tuple(liste)
kume = set(liste)
print(demet, len(demet))
print(sorted(kume), len(kume))
```

Çıktı:

```
(1, 5, 3, 9, 2, 5) 6
[1, 2, 3, 5, 9] 5
```

■

Bu bölümle birlikte Python'ın temel koleksiyonlarını ve döngülerini tamamladık. Bir sonraki bölüm Rastgele Sayılar: random Modülü, zar atmak, kura çekmek ya da bir listeyi karıştırmak gibi işler için bilgisayara rastgele sayılar ürettirmeyi ele alıyor.

 CC BY-NC-SA 4.0

10. Rastgele Sayılar: random Modülü

Zar atan bir oyun, bir sınıftan rastgele öğrenci seçen bir program ya da bir olayın olasılığını binlerce deneme yaparak tahmin eden bir simülasyon yazmak için bilgisayarın rastgele sayılar üretebilmesi gerekir. Oysa bilgisayar her komutu kesin kurallarla yürüten bir makinedir; gerçek anlamda "rastgele" bir şey yapamaz.
Python bu sorunu, rastgele görünen ama aslında belirli bir hesapla üretilen sayılarla çözer. Bu bölümde bu sayıları üreten `random` modülünü tanıyacağız: sayı üretmeyi, bir listeden rastgele seçim yapmayı ve bir listeyi karıştırmayı göreceğiz. Ayrıca tohum değeri sayesinde "rastgele" sonuçları nasıl tekrar elde edebileceğimizi öğreneceğiz.

10.1 Sözde Rastgele Sayılar

Bilgisayarın ürettiği rastgele sayılar bir başlangıç değerinden belirli bir formülle hesaplanır.

Tanım 10.1 (Sözde Rastgele Sayı). Bir başlangıç değerinden belirli bir algoritmayla hesaplanan, ama istatistiksel olarak rastgele sayılardan ayırt edilemeyecek kadar düzensiz görünen sayılara **sözde rastgele sayılar** (pseudo-random numbers) denir. Bu sayıları üreten algoritmaya **rastgele sayı üreteci** denir.

Yani `random` modülünün verdiği sayılar gerçek bir zar atışı gibi öngörülemez değildir; başlangıç değeri bilinirse bütün dizi yeniden hesaplanabilir. Oyunlar, simülasyonlar ve istatistik deneyleri için bu yeterlidir. Şifre ya da güvenlik anahtarı üretmek gibi öngörülemezliğin önemli olduğu işler için ise Python'ın ayrı bir modülü (`secrets`) kullanılır.
Modül, öteki modüller gibi `import random` ile programa dahil edilir; içindeki fonksiyonlar `random.fonksiyon_adi(...)` biçiminde çağrılır. Alt çizgiyle başlamayan adlar şunlardır:

```
import random
print([ad for ad in dir(random) if not ad.startswith("_")])
```

Çıktı:

```
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST',
'SystemRandom', 'TWOPI', 'betavariate', 'binomialvariate', 'choice', 'choices',
'expovariate', 'gammavariate', 'gauss', 'getrandbits', 'getstate', 'lognormvariate',
'normalvariate', 'paretovariate', 'randbytes', 'randint', 'random', 'randrange',
'sample', 'seed', 'setstate', 'shuffle', 'triangular', 'uniform', 'vonmisesvariate',
'weibullvariate']
```

Bu listedeki büyük harfli adlar modülün iç sabitleri ve sınıflarıdır; biz küçük harfli fonksiyonlardan en sık kullanılanlarını göreceğiz.

10.2 Tohum Değeri

Bir rastgele sayı üretecinin ürettiği dizi tamamen başlangıç değerine bağlıdır; bu değeri kendimiz verebiliriz.

Tanım 10.2 (Tohum Değeri). Rastgele sayı üretecinin başlangıç değerine **tohum** (seed) denir ve `random.seed(a)` ile belirlenir. Aynı tohumla başlatılan üreteç her seferinde aynı sayı dizisini üretir. Tohum verilmezse Python, işletim sisteminden ya da saatten alınan, her çalıştırmada farklı bir başlangıç değeri kullanır.

Yani tohum, "rastgele" sonuçların tekrar elde edilebilmesini sağlar: tohumu 42 olan bir program, kim nerede çalıştırırsa çalıştırsın aynı sayıları üretir. Bu, bir programı test ederken, bir simülasyonun sonucunu başkasıyla paylaşırken ya da bir oyunun bir bölümünü hep aynı koşullarla başlatırken çok kullanışlıdır.

 CC BY-NC-SA 4.0

```
import random
random.seed(1)
print(random.random())
print(random.random())
random.seed(1)
print(random.random())
random.seed(2)
print(random.random())
```

Çıktı:

```
0.13436424411240122
0.8474337369372327
0.13436424411240122
0.9560342718892494
```

Tohum 1 yeniden verildiğinde üreteç başa döndü ve aynı sayıyı yeniden üretti. Tohum verildikten sonraki ikinci sayı ise birinciden farklıdır: tohum dizinin başlangıcını belirler, her çağrı dizide bir adım ilerler.

> **i** Bu bölümdeki çıktılar
>
> Bu bölümdeki programların çoğu bir tohum değeriyle başlar; çıktılar bu tohumlarla ve Python 3.12 ile elde edilmiştir. Kendi bilgisayarınızda aynı tohumla aynı sonuçları alırsınız. `random.seed(...)` satırı silinirse sonuçlar her çalıştırmada değişir.

10.3 Sayı Üretmek

`random` modülü hem ondalıklı hem tam sayı üreten fonksiyonlar içerir.

Fonksiyon	Ürettiği değer
`random.random()`	$[0, 1)$ aralığında ondalıklı bir sayı; parametre almaz
`random.uniform(a, b)`	a ile b arasında ondalıklı bir sayı
`random.randint(a, b)`	a ile b arasında, **iki uç da dahil**, bir tam sayı
`random.randrange(bas, bit, adim)`	`range(bas, bit, adim)` sayılarından biri; `bit` dahil değil

`randint` ile `randrange` arasındaki farka dikkat etmek gerekir: `randint(1, 6)` 6'yı da üretebilir, `randrange(1, 6)` üretemez; çünkü `randrange`, `range` gibi bitişi dahil etmez.

```
import random
random.seed(42)
print(random.random())
print(random.uniform(5, 10))
print(random.randint(1, 6))
print(random.randrange(0, 100, 5))
```

Çıktı:

```
0.6394267984578837
5.125053776113335
3
35
```

`randrange(0, 100, 5)` yalnız $0, 5, 10, \ldots, 95$ sayılarından birini verir; bu yüzden sonuç her zaman 5'in katıdır.

Örnek 10.1 (Zar Atmak). Bir zarı 20 kez atıp sonuçları bir listede toplayınız ve her yüzün kaç kez geldiğini sayınız.

 CC BY-NC-SA 4.0

Çözüm.
Her atış için `random.randint(1, 6)` kullanırız; iki uç dahil olduğu için 1'den 6'ya kadar her yüz gelebilir. Atışları bir liste üreteciyle toplar, sonra her yüz için `count` ile sayarız. Döngü değişkeni kullanılmadığından adı `_`'dir.

```
import random
random.seed(7)
atislar = [random.randint(1, 6) for _ in range(20)]
print(atislar)
for yuz in range(1, 7):
    print(yuz, ":", atislar.count(yuz))
```

Çıktı:

```
[3, 2, 4, 6, 1, 1, 5, 1, 3, 5, 1, 5, 2, 1, 1, 4, 4, 1, 2, 1]
1 : 8
2 : 3
3 : 2
4 : 3
5 : 3
6 : 1
```

20 atışta her yüzün ortalama $20/6 \approx 3{,}3$ kez gelmesi beklenir; az sayıda atışta sayıların bundan sapması doğaldır. Atış sayısı arttıkça her yüzün oranı $1/6$'ya yaklaşır. ■

10.4 Rastgele Seçim ve Karıştırma

Sayı üretmenin yanında, var olan bir koleksiyondan rastgele eleman seçmek ya da elemanlarının sırasını karıştırmak da sık yapılan işlerdir.

Fonksiyon	Yaptığı iş
`random.choice(seq)`	`seq` dizisinden (liste, demet, metin, `range`) rastgele bir eleman seçer
`random.choices(seq, weights=..., k=n)`	`seq`'ten `n` eleman seçer; aynı eleman tekrar seçilebilir; `weights` her elemanın seçilme ağırlığıdır
`random.sample(seq, k)`	`seq`'ten `k` **farklı** eleman seçer ve yeni bir liste olarak verir
`random.shuffle(liste)`	listenin elemanlarını **yerinde** karıştırır; yeni liste vermez

`choices` ile `sample` arasındaki fark, torbadan top çekme deneylerindeki farka karşılık gelir.

Tanım 10.3 (İadeli ve İadesiz Seçim). Bir topluluktan art arda seçim yapılırken seçilen eleman her seferinde topluluğa geri konuyorsa **iadeli**, geri konmuyorsa **iadesiz** seçim yapılmış olur. `random.choices` iadeli, `random.sample` iadesiz seçim yapar.

Yani `choices` ile aynı eleman birden çok kez seçilebilir; `sample` ile seçilen elemanların hepsi farklıdır ve `k` topluluğun eleman sayısından büyük olamaz.

```
import random
random.seed(3)
meyveler = ["elma", "armut", "muz", "kiraz"]
print(random.choice(meyveler))
print(random.choice("Ahmet"))
print(random.choice(range(1, 110, 5)))
```

 CC BY-NC-SA 4.0

```
print(random.choices(meyveler, k=5))
print(random.sample(meyveler, 3))
```

Çıktı:

```
armut
t
86
['elma', 'kiraz', 'armut', 'muz', 'muz']
['kiraz', 'armut', 'elma']
```

`choices` ile seçilen beş meyvede aynı meyve birden çok kez görünebilir; `sample` ile seçilen üç meyve ise birbirinden farklıdır.
`weights` parametresi elemanların seçilme olasılıklarını değiştirir. `weights=[2, 1, 1]` verildiğinde birinci elemanın seçilme olasılığı $2/4 = 1/2$, diğerlerininki $1/4$ olur.

```
import random
random.seed(5)
meyveler = ["elma", "armut", "muz"]
secim = random.choices(meyveler, weights=[2, 1, 1], k=1000)
for meyve in meyveler:
    print(meyve, secim.count(meyve))
```

Çıktı:

```
elma 521
armut 240
muz 239
```

1000 seçimde elmanın yaklaşık 500, diğerlerinin yaklaşık 250 kez seçilmesi beklenir; sayılar bu değerlere yakındır.

10.4.1 shuffle ve sample

Bir listeyi karıştırmanın iki yolu vardır ve aralarındaki fark, asıl listenin korunup korunmadığıdır.
`random.shuffle(liste)` listenin kendisini değiştirir; liste metotları gibi (bkz. Koleksiyonlar ve Listeler) geriye bir değer vermez. Asıl listenin bozulmaması isteniyorsa `random.sample(liste, len(liste))` kullanılır: listenin bütün elemanlarını iadesiz seçmek, karışık sırada yeni bir liste verir.

```
import random
random.seed(10)
deste = ["Maça As", "Kupa Kız", "Sinek Vale", "Karo 10"]
random.shuffle(deste)
print(deste)
orijinal = [1, 2, 3, 4, 5]
karisik = random.sample(orijinal, len(orijinal))
print(karisik)
print(orijinal)
```

Çıktı:

```
['Karo 10', 'Sinek Vale', 'Kupa Kız', 'Maça As']
[5, 1, 4, 2, 3]
[1, 2, 3, 4, 5]
```

10.5 Dağılımlardan Sayı Üretmek

`random.uniform` bir aralıktaki bütün sayıları eşit olasılıkla üretir; bazı uygulamalarda ise değerlerin bir ortalama etrafında toplanması istenir.
`random.gauss(mu, sigma)` ve `random.normalvariate(mu, sigma)` fonksiyonları ortalaması `mu`, standart sapması `sigma` olan **normal dağılımdan** sayı üretir: üretilen sayıların çoğu `mu`'ya yakındır, `mu`'dan uzak-

 CC BY-NC-SA 4.0

laştıkça sayılar seyrekleşir. İki fonksiyon aynı dağılımdan sayı üretir ama farklı yöntemler kullandığından aynı tohumla farklı diziler verir. Normal dağılımın kendisi Olasılık Teorisi notlarının konusudur.

```
import random
random.seed(42)
liste1 = [random.normalvariate(5, 10) for _ in range(6)]
print([round(x, 2) for x in liste1])
random.seed(42)
liste2 = [random.gauss(5, 10) for _ in range(6)]
print([round(x, 2) for x in liste2])
```

Çıktı:

```
[7.45, 0.03, 17.55, 3.62, -4.76, 10.65]
[3.56, 3.27, 3.89, 12.02, 3.72, -9.97]
```

Örnek 10.2 (Not Simülasyonu). Ortalaması 70, standart sapması 12 olan normal dağılımdan 1000 not üretiniz. Notları tam sayıya yuvarlayıp 0–100 aralığına sınırlayınız ve ortalamayı, en yüksek ve en düşük notu yazdırınız.

Çözüm.
Her notu `random.gauss(70, 12)` ile üretir, `round` ile yuvarlarız. Normal dağılım ender de olsa 100'den büyük ya da 0'dan küçük değerler üretebilir; bunları sınırlamak için `max(0, min(100, n))` yazarız: `min(100, n)` 100'ü aşan değeri 100'e indirir, `max(0, ...)` da negatif değeri 0'a çıkarır.

```
import random
random.seed(2026)
notlar = [random.gauss(70, 12) for _ in range(1000)]
notlar = [max(0, min(100, round(n))) for n in notlar]
print(f"Ortalama: {sum(notlar) / len(notlar):.2f}")
print("En yüksek:", max(notlar), "- En düşük:", min(notlar))
print("İlk 10 not:", notlar[:10])
```

Çıktı:

```
Ortalama: 70.08
En yüksek: 100 - En düşük: 25
İlk 10 not: [80, 80, 46, 68, 77, 75, 58, 61, 73, 55]
```

Ortalama 70'e çok yakın çıkar. Normal dağılımda değerlerin yaklaşık %95'i ortalamanın iki standart sapma yakınındadır, yani 46 ile 94 arasında kalır; 1000 not arasında bu aralığın dışında kalan birkaç uç not da bulunur; en yüksek not 100 sınırına dayanmıştır. ■

10.6 Örnekler

`random` modülünü önceki bölümlerde öğrendiğimiz döngüler ve koleksiyonlarla birleştirelim.

Örnek 10.3 (Sayısal Loto Çekilişi). 1'den 49'a kadar sayılar arasından 6 farklı sayı çekip küçükten büyüğe sıralı olarak yazınız.

Çözüm.
Çekilen bir top torbaya geri konmadığı için iadesiz seçim, yani `sample` gerekir; `choices` aynı sayıyı iki kez çekebilirdi. Sayıları `range(1, 50)` verir; `sorted` sonucu sıralar.

```
import random
random.seed(49)
cekilis = random.sample(range(1, 50), 6)
print("Çekilen sayılar:", sorted(cekilis))
```

 CC BY-NC-SA 4.0

Çıktı:

```
Çekilen sayılar: [5, 8, 21, 23, 27, 36]
```

49 sayıdan 6'sını seçmenin $\binom{49}{6} = 13\,983\,816$ yolu vardır; tek bir kuponun altı sayıyı da bilme olasılığı bu yüzden yaklaşık on dört milyonda birdir. ■

Örnek 10.4 (Rastgele Sayılı Tahmin Oyunu). İç İçe Döngüler, while Döngüsü ve match-case bölümündeki sayı tahmin oyununu (Örnek 8.4), tutulan sayıyı bilgisayar rastgele seçecek biçimde yazınız.

Çözüm.
Tek fark, tutulan sayının `random.randint(1, 100)` ile seçilmesidir. Aşağıdaki çıktıyı tekrarlanabilir kılmak için bir tohum verdik; gerçek oyunda bu satır silinir, yoksa bilgisayar her oyunda aynı sayıyı tutar.

```
import random
random.seed(8)          # gerçek oyunda bu satırı silin
tutulan = random.randint(1, 100)
sayac = 0
while True:
    sayac += 1
    tahmin = int(input("Tahminin (1-100): "))
    if tahmin > tutulan:
        print("Daha küçük.")
    elif tahmin < tutulan:
        print("Daha büyük.")
    else:
        print(f"Bildin! Sayı {tutulan} idi, {sayac} tahminde buldun.")
        break
```

Çıktı:

```
Tahminin (1-100): 50
Daha küçük.
Tahminin (1-100): 25
Daha büyük.
Tahminin (1-100): 37
Daha küçük.
Tahminin (1-100): 31
Daha küçük.
Tahminin (1-100): 28
Daha büyük.
Tahminin (1-100): 30
Bildin! Sayı 30 idi, 6 tahminde buldun.
```

■

Örnek 10.5 (İki Zarın Toplamının 7 Olması). İki zar atıldığında toplamın 7 gelme olasılığını, iki zarı 10 000 kez atarak tahmin ediniz ve gerçek olasılıkla karşılaştırınız.

Çözüm.
İki zarın $6 \cdot 6 = 36$ eşit olasılıklı sonucundan toplamı 7 olanlar $(1,6), (2,5), (3,4), (4,3), (5,2), (6,1)$'dir; gerçek olasılık $6/36 = 1/6 \approx 0{,}1667$'dir. Deneyde her atışta iki zar atar, toplam 7 ise sayacı artırırız; sonunda sayacı deneme sayısına böleriz.

 CC BY-NC-SA 4.0

```
import random
random.seed(1)
deneme = 10000
yedi = 0
for _ in range(deneme):
    zar1 = random.randint(1, 6)
    zar2 = random.randint(1, 6)
    if zar1 + zar2 == 7:
        yedi += 1
print("Deneysel olasılık:", yedi / deneme)
print("Gerçek olasılık  :", round(1 / 6, 4))
```

Çıktı:

```
Deneysel olasılık: 0.1675
Gerçek olasılık  : 0.1667
```

Bir olasılığı çok sayıda rastgele deneme yaparak tahmin etmeye **Monte Carlo yöntemi** denir; deneme sayısı arttıkça tahmin gerçek değere yaklaşır. Bu yöntemin daha ileri uygulamaları Python ile Bilimsel Hesaplama notlarındadır. ■

10.7 Alıştırmalar

Alıştırmalarda verilen tohumları kullanırsanız sonuçlarınız buradakilerle aynı çıkar.

Alıştırma 10.1 (Yazı Tura). Bir parayı 100 kez atıp kaç kez yazı geldiğini sayınız. Tohum olarak 11 kullanınız.

Çözüm.
Her atışta `random.choice(["yazı", "tura"])` ile iki sonuçtan birini seçeriz ve sonuçları bir listede toplarız. Yazı sayısı `count` ile bulunur.

```
import random
random.seed(11)
atislar = [random.choice(["yazı", "tura"]) for _ in range(100)]
print("Yazı:", atislar.count("yazı"))
print("Tura:", atislar.count("tura"))
```

Çıktı:

```
Yazı: 56
Tura: 44
```

100 atışta yaklaşık 50 yazı beklenir; birkaç atışlık sapmalar doğaldır. ■

Alıştırma 10.2 (Altı Gelene Kadar). Bir zarı 6 gelene kadar atan ve kaç atış yapıldığını yazan bir program yazınız. Tohum olarak 4 kullanınız.

Çözüm.
Kaç atış gerektiği önceden bilinmediğinden bir `while` döngüsü kullanırız. Döngüye girebilmek için `zar`'a başlangıçta 6'dan farklı bir değer veririz.

```
import random
random.seed(4)
```

 CC BY-NC-SA 4.0

```
zar = 0
atis = 0
while zar != 6:
    zar = random.randint(1, 6)
    atis += 1
    print(zar, end=" ")
print()
print(atis, "atışta 6 geldi.")
```

Çıktı:

```
2 3 1 6
4 atışta 6 geldi.
```

Her atışta 6 gelme olasılığı $1/6$ olduğundan, ilk 6 için gereken atış sayısı ortalamada 6'dır. ■

Alıştırma 10.3 (İki Takıma Ayırmak). `ogrenciler = ["Ali", "Ayşe", "Can", "Deniz", "Ece", "Fatih"]` listesindeki öğrencileri rastgele iki üçlü takıma ayırınız. Tohum olarak 12 kullanınız.

Çözüm.
Listeyi `shuffle` ile karıştırır, sonra ilk üç elemanı bir takıma, son üç elemanı öbür takıma dilimleriz. Asıl listenin bozulmaması için önce bir kopyasını alırız.

```
import random
random.seed(12)
ogrenciler = ["Ali", "Ayşe", "Can", "Deniz", "Ece", "Fatih"]
karisik = ogrenciler.copy()
random.shuffle(karisik)
print("1. takım:", karisik[:3])
print("2. takım:", karisik[3:])
```

Çıktı:

```
1. takım: ['Fatih', 'Ayşe', 'Ali']
2. takım: ['Ece', 'Can', 'Deniz']
```

■

Alıştırma 10.4 (Rastgele Şifre). Küçük harflerden ve rakamlardan oluşan 8 karakterlik rastgele bir metin üretiniz. Tohum olarak 99 kullanınız.

Çözüm.
Kullanılabilecek karakterleri bir metinde toplarız. `random.choices(karakterler, k=8)` iadeli seçimle 8 karakter seçer; aynı karakter iki kez gelebilir. Seçilen karakterleri bir döngüyle boş bir metne ekleyerek birleştiririz.

```
import random
random.seed(99)
karakterler = "abcdefghijklmnopqrstuvwxyz0123456789"
secilen = random.choices(karakterler, k=8)
sifre = ""
for k in secilen:
    sifre += k
print(sifre)
```

 CC BY-NC-SA 4.0

Çıktı:

```
ohgi1jny
```

Gerçek bir şifre için `random` değil, öngörülemez sayılar üreten `secrets` modülü kullanılmalıdır; `random`'ın ürettiği dizi, tohumu bilen herkes tarafından yeniden hesaplanabilir. ■

Alıştırma 10.5 (Aynı Tohum, Aynı Liste). Aşağıdaki programın son satırı ne yazar? Neden?

```
import random
random.seed(42)
liste1 = [random.randint(1, 100) for _ in range(10)]
random.seed(42)
liste2 = [random.randint(1, 100) for _ in range(10)]
liste3 = [random.randint(1, 100) for _ in range(10)]
print(liste1)
print(liste1 == liste2, liste1 == liste3)
```

Çözüm.
`liste1` ve `liste2` aynı tohumdan hemen sonra üretildiği için aynı on sayıyı içerir. `liste3` ise `liste2`'den sonra, dizinin kaldığı yerden devam ederek üretilir; üreteç başa dönmediği için farklı sayılar içerir. Son satır `True False` yazar.

```
import random
random.seed(42)
liste1 = [random.randint(1, 100) for _ in range(10)]
random.seed(42)
liste2 = [random.randint(1, 100) for _ in range(10)]
liste3 = [random.randint(1, 100) for _ in range(10)]
print(liste1)
print(liste1 == liste2, liste1 == liste3)
```

Çıktı:

```
[82, 15, 4, 95, 36, 32, 29, 18, 95, 14]
True False
```

İki listenin `==` ile karşılaştırılması, elemanlarının sırasıyla eşit olup olmadığını sorar. ■

Alıştırma 10.6 (Beş Farklı Sayı). 1 ile 20 arasından beş farklı sayıyı iki yolla seçiniz: `sample` ile ve bir `while` döngüsünde `randint` ile. Tohum olarak 6 kullanınız.

Çözüm.
`sample` iadesiz seçim yaptığı için doğrudan beş farklı sayı verir. İkinci yolda boş bir listeyle başlar, `randint` ile sayı üretir ve sayı listede **yoksa** ekleriz; liste beş elemana ulaşınca döngü biter.

```
import random
random.seed(6)
print(sorted(random.sample(range(1, 21), 5)))
secilenler = []
while len(secilenler) < 5:
    sayi = random.randint(1, 20)
    if sayi not in secilenler:
```

 CC BY-NC-SA 4.0

```
        secilenler.append(sayi)
print(sorted(secilenler))
```

Çıktı:

```
[2, 3, 9, 16, 19]
[1, 5, 12, 16, 19]
```

İki satır farklı sayılar içerir, çünkü ikinci seçim üretecin kaldığı yerden devam eder. ■

Bu bölümle birlikte dersin ilk yarısındaki bütün araçları gördük: değişkenler ve tipler, işlemler, girdi ve çıktı, karar yapıları, listeler ve demetler, döngüler ve rastgele sayılar. Bir sonraki bölüm Problemler: Tipler, İşlemler ve Koşullar, bu konuları "bu kodun çıktısı nedir?" sorularıyla tekrar ediyor.

 CC BY-NC-SA 4.0

11. Problemler: Tipler, İşlemler ve Koşullar

Bir programlama dilini öğrenmenin en etkili yollarından biri, kısa kodları çalıştırmadan okuyup ne yapacaklarını tahmin etmektir. Bu bölümdeki her problem bir kod parçası verir ve onun çıktısını sorar. Çözümler kodu satır satır izler, her adımda değişkenlerin değerlerini ve tiplerini yazar.
Problemler Değişkenler, Veri Tipleri ve Atama, Aritmetik İşlemler ve math Modülü, Girdi ve Çıktı İşlemleri ve Karşılaştırma, Mantık Operatörleri ve Koşullu İfadeler bölümlerinin konularını kapsar; son kısımda kısa `for` döngüleri de yer alır. Her problemi önce kâğıt üzerinde çözün, sonra çözümü açın ve kodu çalıştırarak sonucunuzu doğrulayın.

11.1 Tipler ve Dönüşümler

Bu kısımdaki problemlerde aynı görünen değerlerin metin mi sayı mı olduğuna dikkat edin.

Alıştırma 11.1 (Rakam İçeren Metinler). Aşağıdaki programın çıktısını bulunuz.

```
a = "3"
b = "5"
print(type(a))
print(a + b)
print(3 * b)
print(3 * int(b))
print(a + b + "2")
```

Çözüm.
`a` ve `b` tırnak içinde yazıldığı için birer metindir; tipleri `str`'dir.

- `a + b`: iki metin birleştirilir: `"35"`.
- `3 * b`: bir metin bir tam sayıyla çarpıldığında üç kez yan yana yazılır: `"555"`.
- `3 * int(b)`: `int(b)` 5 sayısıdır; $3 \cdot 5 = 15$.
- `a + b + "2"`: üç metin soldan sağa birleştirilir: `"352"`.

Çıktı:

```
<class 'str'>
35
555
15
352
```

■

Alıştırma 11.2 (Metinleri Sayıya Çevirmek). Aşağıdaki programın çıktısını bulunuz.

```
a = "3"
b = "5"
print(int(a) + int(b))
print(int(a) + int(b) + 2)
print(float(a) + float(b))
x = str(3)
y = str(5)
print(x + y)
```

 CC BY-NC-SA 4.0

Çözüm.
`int(a)` ve `int(b)` metinleri 3 ve 5 tam sayılarına çevirir; toplamları 8, bir fazlası 2 eklenince 10 olur. `float` ile çevrilen sayıların toplamı ondalıklı olarak `8.0` yazılır. Son iki satırda ise ters yönde bir dönüşüm vardır: `str(3)` ve `str(5)` sayıları metne çevirir ve `x + y` birleştirme yapar: `"35"`.
Çıktı:

```
8
10
8.0
35
```

■

Alıştırma 11.3 (Metinlerin Mantıksal Değeri). Aşağıdaki programın çıktısını bulunuz.

```
a = "3"
print(bool(a))
print(bool("0"))
print(bool(0))
print(bool(""))
```

Çözüm.
`bool` bir metni yalnız boşsa `False`'a çevirir. `"3"` ve `"0"` boş olmayan metinlerdir; içlerindeki rakamın ne olduğu önemli değildir, ikisi de `True` verir. `0` sayısı ise `False`'tur; boş metin `""` de `False`'tur.
Çıktı:

```
True
True
False
False
```

■

Alıştırma 11.4 (Kesme ve Toplama). Aşağıdaki programın çıktısını bulunuz.

```
x = float(3)
y = int(5.7)
z = x + y
print(z)
print(int(2.9))
```

Çözüm.
`x` 3,0 olur. `int(5.7)` ondalık kısmı atar, yuvarlamaz: `y` 5'tir. Bir `float` ile bir `int` toplandığında sonuç `float` olur: $3{,}0 + 5 = 8{,}0$. `int(2.9)` de aynı nedenle 3 değil 2'dir.
Çıktı:

```
8.0
2
```

■

Alıştırma 11.5 (Parantez İçinde Tip Adı). Aşağıdaki programın çıktısını bulunuz.

 CC BY-NC-SA 4.0

```
a = (int)(3.5 + ((float)(5.0 / 2) * 3))
print(a)
b = int(3.1 + (float(5.0 / 2) * 3))
print(b)
c = int(3.1 + (float(5.2 / 2) * 3))
print(c)
```

Çözüm.
Birinci satırdaki `(int)` ve `(float)` yazımları C dilindeki tip dönüştürme yazımına benzer; Python'da ise yalnız gereksiz parantez içine alınmış `int` ve `float` fonksiyonlarıdır ve `int(...)`, `float(...)` ile aynı işi yapar.

- `a`: $5{,}0/2 = 2{,}5$; $2{,}5 \cdot 3 = 7{,}5$; $3{,}5 + 7{,}5 = 11{,}0$; `int` 11 verir.
- `b`: $3{,}1 + 7{,}5 = 10{,}6$; `int` ondalık kısmı atar ve 10 verir.
- `c`: $5{,}2/2 = 2{,}6$; $2{,}6 \cdot 3 = 7{,}8$; $3{,}1 + 7{,}8 = 10{,}9$; `int` yine 10 verir.

Çıktı:

```
11
10
10
```

`int` sonucu yuvarlamadığı için 10,9 bile 10 olur. ■

Alıştırma 11.6 (Anahtar Kelime mi?). Aşağıdaki programın çıktısını bulunuz.

```
import keyword
print(keyword.iskeyword("not"))
print(keyword.iskeyword("lambda"))
print(keyword.iskeyword("print"))
print(keyword.iskeyword("True"))
```

Çözüm.
`not` ve `lambda` Python'ın anahtar kelimeleridir. `print` ise hazır bir fonksiyonun adıdır, anahtar kelime değildir; bu yüzden değişken adı olarak kullanılabilir (ama kullanılmamalıdır). `True` bir anahtar kelimedir; küçük harfle yazılmış `true` olsaydı anahtar kelime olmazdı.
Çıktı:

```
True
True
False
True
```

■

11.2 Yazdırma ve Biçimlendirme

Bu kısımda `print`'in değerleri nasıl ayırdığına ve biçimlendirmenin hangi değeri nereye yerleştirdiğine dikkat edin.

Alıştırma 11.7 (Tırnak İçinde Kalan Değişkenler). Aşağıdaki programın çıktısını bulunuz.

```
a = 5
b = 3
```

 CC BY-NC-SA 4.0

```
print("a, ile,b, nin toplamı:,a+b,dir.")
print(a, "ile", b, "nin toplamı:", a + b, "dir.")
```

Çözüm.
Birinci `print`'e tek bir metin verilmiştir; virgüller ve değişken adları metnin içindedir ve olduğu gibi yazılır. İkinci `print`'e ise altı ayrı değer verilmiştir: `a`, `"ile"`, `b`, `"nin toplamı:"`, `a + b` ve `"dir."`. Değişkenlerin değerleri hesaplanır ve değerler aralarında birer boşlukla yazılır.
Çıktı:

```
a, ile,b, nin toplamı:,a+b,dir.
5 ile 3 nin toplamı: 8 dir.
```

■

Alıştırma 11.8 (Aynı Cümle, Farklı Yazımlar). Aşağıdaki programın çıktısını bulunuz.

```
a = 5
b = 3
print(f"{a} ile {b} nin toplamı {a + b} dir.")
print("{} ile {} nin toplamı {} dir.".format(a, b, a + b))
print("{} ile {} nin toplamı {} dir.".format(b, a + b, a))
print("%d ile %d nin toplamı %d dir." % (a, b, a + b))
print("%d ile %d nin toplamı %f dir." % (a, b, a + b))
```

Çözüm.
İlk iki satır ve dördüncü satır aynı cümleyi üç farklı yolla yazar: `5 ile 3 nin toplamı 8 dir.`. Üçüncü satırda `format`'a verilen değerlerin sırası değiştirilmiştir; `{}`'ler sırayla `3`, `8` ve `5` ile dolar ve anlamsız bir cümle çıkar. Son satırda toplam `%f` ile yazılır; `%f` virgülden sonra altı basamak kullanır: `8.000000`.
Çıktı:

```
5 ile 3 nin toplamı 8 dir.
5 ile 3 nin toplamı 8 dir.
3 ile 8 nin toplamı 5 dir.
5 ile 3 nin toplamı 8 dir.
5 ile 3 nin toplamı 8.000000 dir.
```

■

Alıştırma 11.9 (sep ile end). Aşağıdaki programın çıktısını bulunuz.

```
print("Python kullanmayı biliyor musun", end="?")
print()
print("Python kullanmayı biliyor musun", sep="&&")
print("Python", "kullanmayı", "biliyor", "musun", sep="&&")
```

Çözüm.
Birinci satır cümleyi yazar ve satır sonuna yeni satır yerine `?` koyar; hemen ardından gelen boş `print()` satırı bitirir. Üçüncü satırda `print`'e tek bir değer verildiği için ayırıcıya hiç gerek kalmaz; `sep="&&"` etkisizdir. Son satırda dört değer vardır ve aralarına `&&` konur.
Çıktı:

 CC BY-NC-SA 4.0

```
Python kullanmayı biliyor musun?
Python kullanmayı biliyor musun
Python&&kullanmayı&&biliyor&&musun
```

■

Alıştırma 11.10 (Virgül ve Kaçış Karakterleri). Aşağıdaki programın çıktısını bulunuz. İkinci ve üçüncü satırların başında neden bir boşluk görünür?

```
a1 = "Matematik"
a2 = "İstatistik"
print(a1, a2)
print(a1 + a2)
print(a1, "\n", a2)
print(a1, "\\", a2)
```

Çözüm.
`print(a1, a2)` iki metni aralarına bir boşluk koyarak yazar; `a1 + a2` ise boşluksuz birleştirir. Üçüncü `print`'e üç değer verilmiştir: `a1`, yeni satır karakteri `"\n"` ve `a2`. `print` her değerin arasına bir boşluk koyar; bu yüzden `Matematik`'ten sonra bir boşluk ve yeni satır, yeni satırdan sonra da bir boşluk ve `İstatistik` gelir. `İstatistik`'in önündeki boşluk bu ikinci boşluktur. Son satırda `"\\"` tek bir ters eğik çizgi yazar.
Çıktı:

```
Matematik İstatistik
MatematikİstatistikMatematik
 İstatistik
Matematik \ İstatistik
```

Satır başındaki boşluk istenmiyorsa `print(a1 + "\n" + a2)` ya da `print(a1, a2, sep="\n")` yazılır. ■

11.3 Atama ve Aritmetik

Bu kısımda her atamadan sonra değişkenlerin değerlerini ve tiplerini yazarak ilerleyin.

Alıştırma 11.11 (Negatif Sayı ve Yuvarlama). Aşağıdaki programın çıktısını bulunuz.

```
a1 = 3 - (-5)
a2 = 3.2 + (-5)
print("a1:", a1)
print("a2:", a2)
print("a2:", round(a2, 2))
```

Çözüm.
$3-(-5)=8$'dir. $3{,}2+(-5)=-1{,}8$ olmalıdır; ama $3{,}2$ ikilik tabanda tam olarak saklanamadığından sonuç $-1{,}7999999999999998$ olarak yazılır. `round(a2, 2)` bu küçük farkı yuvarlayarak $-1{,}8$ verir.
Çıktı:

```
a1: 8
a2: -1.7999999999999998
a2: -1.8
```

■

 CC BY-NC-SA 4.0

Alıştırma 11.12 (Zincir Atama ve Birleşik Atama). Aşağıdaki programın çıktısını bulunuz.

```
z = y = 21
z *= 5
y /= z
print("z =", z)
print("y =", y)
```

Çözüm.
İlk satır `z`'ye ve `y`'ye 21 değerini verir. `z *= 5` $z = 105$ yapar; `y` değişmez. `y /= z` $y = 21/105 = 0{,}2$ yapar; bölme `/` ile yapıldığı için sonuç `float`'tur.
Çıktı:

```
z = 105
y = 0.2
```

■

Alıştırma 11.13 (Atamaları İzlemek). Aşağıdaki programın çıktısını bulunuz.

```
i, j = 18, 2
i /= j
j *= i
j = i + 1
i = j - 1
print("i:", i)
print("j:", j)
```

Çözüm.
Satır satır:

- `i, j = 18, 2`: $i = 18$, $j = 2$.
- `i /= j`: $i = 9{,}0$ (`float`).
- `j *= i`: $j = 2 \cdot 9{,}0 = 18{,}0$.
- `j = i + 1`: $j = 10{,}0$; önceki $18{,}0$ değeri kaybolur.
- `i = j - 1`: $i = 9{,}0$.

Çıktı:

```
i: 9.0
j: 10.0
```

■

Alıştırma 11.14 (Uzun Bir İfade). Aşağıdaki programın çıktısını bulunuz.

```
c1 = 5 - 6 / 3 * 2 - 9 / 3 / 3 - 5 + 7 * 6 / 3
print("c1:", c1)
```

Çözüm.
Çarpma ve bölmeler soldan sağa önce yapılır:

- `6 / 3 * 2`: $2{,}0 \cdot 2 = 4{,}0$.
- `9 / 3 / 3`: $3{,}0/3 = 1{,}0$.
- `7 * 6 / 3`: $42/3 = 14{,}0$.

 CC BY-NC-SA 4.0

İfade $5 - 4{,}0 - 1{,}0 - 5 + 14{,}0$ olur ve soldan sağa $9{,}0$ bulunur.
Çıktı:

```
c1: 9.0
```

■

Alıştırma 11.15 (Parantezler ve Üs). Aşağıdaki programın çıktısını bulunuz.

```
c2 = (6 + 2 * 4) / 2 - 3 ** 2 + (6 / 3) / (2 - 3)
print("c2:", c2)
```

Çözüm.
Parantezler önce hesaplanır: $6 + 2 \cdot 4 = 14$, $6/3 = 2{,}0$ ve $2 - 3 = -1$. Üs alınır: $3^2 = 9$. Bölmeler yapılır: $14/2 = 7{,}0$ ve $2{,}0/(-1) = -2{,}0$. Son olarak $7{,}0 - 9 + (-2{,}0) = -4{,}0$ bulunur.
Çıktı:

```
c2: -4.0
```

■

Alıştırma 11.16 (Kesirli Üs). Aşağıdaki programın çıktısını bulunuz.

```
a = 36
z = -(5 * (a ** (3 / 2)) / 4)
print("z:", z)
```

Çözüm.
$3/2 = 1{,}5$ ve $36^{1{,}5} = 36 \cdot \sqrt{36} = 36 \cdot 6 = 216$'dır; kesirli üs sonucu `float` verir: $216{,}0$. Sonra $5 \cdot 216{,}0 = 1080{,}0$, $1080{,}0/4 = 270{,}0$ ve dıştaki eksi işaretiyle $z = -270{,}0$ olur.
Çıktı:

```
z: -270.0
```

■

Alıştırma 11.17 (Negatif Sayının Kalanı). Aşağıdaki programın çıktısını bulunuz.

```
Y = -4 * 16 % 4 + 2 ** (2 * 2 + 2)
print("Y:", Y)
print(16 % 4, 4 % 16)
```

Çözüm.
Parantez içi 6, üs $2^6 = 64$'tür. Eksi işareti `-4`'e aittir; `*` ve `%` soldan sağa yapılır: $-4 \cdot 16 = -64$ ve $-64 = (-16) \cdot 4 + 0$ olduğundan $-64 \bmod 4 = 0$. Böylece $Y = 0 + 64 = 64$ olur. İkinci satırda $16 \bmod 4 = 0$; $4 \bmod 16 = 4$'tür, çünkü $4 = 0 \cdot 16 + 4$.
Çıktı:

```
Y: 64
0 4
```

 CC BY-NC-SA 4.0

■

11.4 Mantık ve Koşullar

Bu kısımda koşulların doğru mu yanlış mı olduğunu belirleyip hangi kolun çalıştığını izleyin.

Alıştırma 11.18 (Bir Mantık İfadesi). Aşağıdaki programın çıktısını bulunuz.

```
print((2 ** 2 == 4 and False) or (not (16 < 15 and True)))
```

Çözüm.
Soldaki parantezde $2^2 = 4$ olduğundan `2 ** 2 == 4` `True`'dur; `True and False` `False` olur. Sağdaki parantezde `16 < 15` `False`'tur; `False and True` `False` olur ve `not` onu `True`'ya çevirir. Sonuç `False or True`, yani `True`'dur.
Çıktı:

```
True
```

■

Alıştırma 11.19 (is Operatörü). Aşağıdaki programın çıktısını bulunuz.

```
x = False
y = True
print(y)
print(x is y)
print(y is y)
print(x == (not y))
```

Çözüm.
`is` iki adın bellekteki **aynı** nesneyi gösterip göstermediğini sorar. Python'da yalnız bir tane `True` ve bir tane `False` nesnesi vardır. `x is y` `False` `is True` sorusudur ve yanlıştır; `y is y` doğrudur. Son satırda `not y` `False`'tur ve `x == False` doğrudur.
Çıktı:

```
True
False
True
True
```

Sayıları ve metinleri karşılaştırırken `is` değil `==` kullanılmalıdır: iki eşit değer bellekte farklı nesneler olabilir. `is`'in güvenle kullanıldığı başlıca yer `None` ile karşılaştırmadır. ■

Alıştırma 11.20 (Bir if İçinde Takas). Aşağıdaki programın çıktısını bulunuz.

```
a = 5
b = 9
c = 0
if a < b:
    c = a
    a = b
    b = c
print(a, b, sep=",")
```

 CC BY-NC-SA 4.0

Çözüm.
$5 < 9$ doğru olduğundan blok çalışır. `c = a` $c = 5$ yapar ve `a`'nın eski değerini saklar; `a = b` $a = 9$ yapar; `b = c` $b = 5$ yapar. Bu, üçüncü bir değişkenle yapılan bir takastır; `a` ile `b` yer değiştirmiştir.
Çıktı:

```
9,5
```

■

Alıştırma 11.21 (Koşul Bloğunda Ardışık Hesap). Aşağıdaki programın çıktısını bulunuz.

```
a = 5
b = 9
c = 3
if a < b:
    c = a + c * b
    a = b - a + c
    b = c
print("a:", a)
print("b:", b)
print("c:", c)
```

Çözüm.
Koşul doğrudur. Her satır, önceki satırların değiştirdiği değerleri kullanır:

- `c = a + c * b`: $c = 5 + 3 \cdot 9 = 32$.
- `a = b - a + c`: $a = 9 - 5 + 32 = 36$; burada `c`'nin yeni değeri 32 kullanılır.
- `b = c`: $b = 32$.

Çıktı:

```
a: 36
b: 32
c: 32
```

■

Alıştırma 11.22 (Bağımsız ve İç İçe Koşullar). Aşağıdaki programın çıktısını bulunuz.

```
a, b, c = 2, 7, 6
if a + b > c:
    print(a + b + c)
if a + c > b:
    print(a - b + c)
    if a - b + c > 5:
        print("sonuç doğru")
    else:
        print("hatalı sonuç")
```

Çözüm.
İki dış `if` birbirinden bağımsızdır; ikisi de ayrı ayrı denetlenir. $2 + 7 = 9 > 6$ olduğundan $2 + 7 + 6 = 15$ yazılır. $2 + 6 = 8 > 7$ olduğundan ikinci blok da çalışır ve $2 - 7 + 6 = 1$ yazılır. İçteki koşul $1 > 5$ yanlış olduğundan `else` kolu çalışır.
Çıktı:

 CC BY-NC-SA 4.0

```
15
1
hatalı sonuç
```

■

Alıştırma 11.23 (Bir Hesabın Büyüklüğüne Göre Karar). Aşağıdaki programın çıktısını bulunuz.

```
x, y, z = 12, 5, 8
kontrol = (x * y) + (y * z) - (x // z)
print(f"Kontrol değeri: {kontrol}")
if kontrol > 50:
    print("Büyük değer elde edildi!")
elif kontrol > 25:
    print("Orta değer elde edildi.")
else:
    print("Küçük değer elde edildi.")
print("Program sona erdi.")
```

Çözüm.
$x \cdot y = 60$, $y \cdot z = 40$ ve $12 // 8 = 1$'dir; `kontrol` $60 + 40 - 1 = 99$ olur. $99 > 50$ olduğundan ilk kol çalışır ve geri kalan koşullara bakılmaz. Son satır koşulların dışındadır ve her durumda çalışır.
Çıktı:

```
Kontrol değeri: 99
Büyük değer elde edildi!
Program sona erdi.
```

■

Alıştırma 11.24 (İşaret Değiştirip Hesaplamak). Aşağıdaki programın çıktısını bulunuz.

```
X = -3
if X != 3:
    X = -X
X = 2 * X + 1
print(X)
```

Çözüm.
$-3 \neq 3$ doğru olduğundan blok çalışır ve $X = 3$ olur. Son atama `if` bloğunun dışındadır ve her durumda çalışır: $X = 2 \cdot 3 + 1 = 7$.
Çıktı:

```
7
```

■

11.5 Döngülerle Kısa Sorular

Bu kısımdaki döngülerde her yinelemede döngü değişkeninin değerini yazarak ilerleyin.

Alıştırma 11.25 (Her Harfe Bir e). Aşağıdaki programın çıktısını bulunuz.

 CC BY-NC-SA 4.0

```
for i in "zeynep":
    a = i + "e"
    print(a)
```

Çözüm.
Döngü "zeynep" metninin harflerini sırayla alır: z, e, y, n, e, p. Her harfin sonuna `"e"` eklenip yazılır: `ze`, `ee`, `ye`, `ne`, `ee`, `pe`.
Çıktı:

```
ze
ee
ye
ne
ee
pe
```

■

Alıştırma 11.26 (Yalnız Bir Harfi Atlamak). Aşağıdaki programın çıktısını bulunuz.

```
for i in "zeynep":
    if i == "e":
        continue
    print(i)
```

Çözüm.
`e` harfine gelindiğinde `continue` o yinelemenin geri kalanını, yani `print`'i atlar. İki `e` atlanır; z, y, n ve p yazılır.
Çıktı:

```
z
y
n
p
```

■

Alıştırma 11.27 (Döngü Değişkenine Atama). Aşağıdaki programın çıktısını bulunuz.

```
for i in "zeynep":
    if i == "e" or i == "y":
        i = "yok"
    else:
        i = "ne"
    print(i)
```

Çözüm.
Döngünün içinde `i`'ye yeni bir değer atanabilir; bu yalnız o yinelemeyi etkiler, bir sonraki yinelemede `i` yine metnin sıradaki harfini alır. Harf `e` ya da `y` ise `yok`, değilse `ne` yazılır: z → `ne`, e → `yok`, y → `yok`, n → `ne`, e → `yok`, p → `ne`.
Çıktı:

 CC BY-NC-SA 4.0

```
ne
yok
yok
ne
yok
ne
```

■

Alıştırma 11.28 (Tek Numaralı Sıra). Aşağıdaki programın çıktısını bulunuz.

```
isimler = ["Ali", "Ayşe", "Mehmet"]
for indeks, isim in enumerate(isimler):
    indeks = 2 * indeks + 1
    print(indeks, isim)
```

Çözüm.
`enumerate` indeksleri 0, 1, 2 olarak verir. Her yinelemede `indeks` $2 \cdot \text{indeks} + 1$ değeriyle değiştirilir: 1, 3, 5. Bu değişiklik `enumerate`'in bir sonraki indeksi etkilemez.
Çıktı:

```
1 Ali
3 Ayşe
5 Mehmet
```

■

Alıştırma 11.29 (2, 3 ve 7'ye Bölünenler). Aşağıdaki program 1 ile 100 arasında 2'ye, 3'e ve 7'ye bölünebilen sayıları bulur. Çıktısını bulunuz.

```
cozum = []
for i in range(1, 101):
    if i % 2 == 0 and i % 3 == 0 and i % 7 == 0:
        cozum.append(i)
print(cozum)
```

Çözüm.
Bir sayı 2'ye, 3'e ve 7'ye bölünüyorsa, bu sayılar aralarında asal olduğu için $2 \cdot 3 \cdot 7 = 42$'ye bölünür. 100'e kadar 42'nin katları 42 ve 84'tür.
Çıktı:

```
[42, 84]
```

■

Alıştırma 11.30 (and ile or Bir Arada). Aşağıdaki programın çıktısını bulunuz.

```
cozum = []
for i in range(1, 81):
    if i % 2 == 0 and i % 3 == 0 or i % 7 == 0:
        cozum.append(i)
print(cozum)
```

 CC BY-NC-SA 4.0

Çözüm.
`and`, `or`'dan önce uygulanır; koşul `(i % 2 == 0 and i % 3 == 0) or i % 7 == 0` olarak okunur. Bir sayı ya 6'ya bölünüyorsa ya da 7'ye bölünüyorsa listeye girer. 80'e kadar 6'nın katları $6, 12, \ldots, 78$ (13 sayı), 7'nin katları $7, 14, \ldots, 77$ (11 sayı) ve ikisinin ortak katı 42'dir; liste $13 + 11 - 1 = 23$ sayı içerir ve sayılar küçükten büyüğe sıralıdır.
Çıktı:

```
[6, 7, 12, 14, 18, 21, 24, 28, 30, 35, 36, 42, 48, 49, 54, 56, 60, 63, 66, 70, 72,
77, 78]
```

Koşul `i % 2 == 0 and (i % 3 == 0 or i % 7 == 0)` diye parantezlenseydi yalnız çift sayılar alınırdı; örneğin 7 ve 21 listeye girmezdi. ■

Bir sonraki bölüm Problemler: Listeler, Döngüler ve Demetler, aynı yöntemle listeler, liste üreteçleri, `while` döngüleri ve demetler üzerine problemler içeriyor.

 CC BY-NC-SA 4.0

12. Problemler: Listeler, Döngüler ve Demetler

Bu bölüm, Problemler: Tipler, İşlemler ve Koşullar bölümündeki yöntemi listelere, döngülere ve demetlere uygular: her problem bir kod parçası verir ve çıktısını sorar. Problemlerin bir kısmı kısa bir hesabı adım adım izlemeyi, bir kısmı da bir liste üretecinin ya da bir döngünün hangi sayıları seçtiğini anlamayı gerektirir.
Problemler Koleksiyonlar ve Listeler, for Döngüsü ve Liste Üreteçleri, İç İçe Döngüler, while Döngüsü ve match-case ve Demetler bölümlerinin konularını kapsar. Uzun döngülerde değişkenlerin değerlerini bir tabloya yazmak işi çok kolaylaştırır.

12.1 Liste Dilimleme

Bu kısımdaki bütün problemler `liste = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]` listesini kullanır; her elemanın değeri kendi indeksine eşit olduğundan dilimdeki sayılar doğrudan seçilen indeksleri gösterir.

Alıştırma 12.1 (Başlangıç ve Bitiş). Aşağıdaki programın çıktısını bulunuz.

```
liste = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(liste[:2])
print(liste[1:2])
print(liste[5:2])
print(liste[2:])
print(liste[:-2])
```

Çözüm.
Dilimde başlangıç dahil, bitiş hariçtir (Tanım 6.7).

- `liste[:2]`: indeks 0 ve 1.
- `liste[1:2]`: yalnız indeks 1; sonuç tek elemanlı bir listedir.
- `liste[5:2]`: adım pozitifken başlangıç bitişten sonra olduğu için boş liste.
- `liste[2:]`: indeks 2'den sona kadar.
- `liste[:-2]`: baştan, sondan ikinci elemana kadar; -2 indeksli 8 hariçtir.

Çıktı:

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

■

Alıştırma 12.2 (Pozitif ve Negatif Adım). Aşağıdaki programın çıktısını bulunuz.

```
liste = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(liste[::2])
print(liste[1::2])
print(liste[:5:2])
```

```
print(liste[::-2])
print(liste[5:1:-2])
```

Çözüm.

- `liste[::2]`: baştan sona ikişer: $0, 2, 4, 6, 8$.
- `liste[1::2]`: 1'den başlayıp ikişer: $1, 3, 5, 7, 9$.
- `liste[:5:2]`: baştan 5'e kadar (5 hariç) ikişer: $0, 2, 4$.
- `liste[::-2]`: adım negatif ve başlangıç yazılmadığı için sondan başlanır: $9, 7, 5, 3, 1$.
- `liste[5:1:-2]`: 5'ten geriye ikişer: $5, 3$; bir sonraki 1, bitiş olduğu için alınmaz.

Çıktı:

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

■

Alıştırma 12.3 (Farklı Adımlarla Dilimler). Aşağıdaki programın çıktısını bulunuz.

```
liste = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(liste[5:0:-1])
print(liste[::3])
print(liste[2:9:2])
print(liste[3::4])
print(liste[::] == liste[:])
```

Çözüm.

- `liste[5:0:-1]`: 5'ten geriye birer: $5, 4, 3, 2, 1$; bitiş olan 0 alınmaz.
- `liste[::3]`: üçer: $0, 3, 6, 9$.
- `liste[2:9:2]`: 2'den 9'a kadar ikişer: $2, 4, 6, 8$.
- `liste[3::4]`: 3'ten dörder: $3, 7$; bir sonraki 11 listenin dışındadır.
- `liste[::]` ve `liste[:]` ikisi de listenin tam bir kopyasını verir; elemanları aynı olduğu için `== True` verir.

Çıktı:

```
[5, 4, 3, 2, 1]
[0, 3, 6, 9]
[2, 4, 6, 8]
[3, 7]
True
```

■

12.2 Liste Üreteçleri

Bu kısımda her liste üretecinin hangi değerleri dolaştığını, hangilerini seçtiğini ve seçtiklerine hangi işlemi uyguladığını ayrı ayrı belirleyin.

Alıştırma 12.4 (Dizilerin İlk Terimleri). Aşağıdaki programın çıktısını bulunuz.

 CC BY-NC-SA 4.0

```
from math import factorial
print([factorial(x) for x in range(1, 7)])
print([2 ** n for n in range(10)])
print([1 / x ** 2 for x in range(1, 5)])
```

Çözüm.

- Birinci liste $1!, 2!, \ldots, 6!$: $1, 2, 6, 24, 120, 720$.
- İkinci liste $2^0, 2^1, \ldots, 2^9$: ilk terimi 1, oranı 2 olan geometrik dizinin ilk on terimi.
- Üçüncü liste $1/1^2, 1/2^2, 1/3^2, 1/4^2$: 1,0, 0,25, 0,111..., 0,0625. `/` kullanıldığı için terimler `float`'tur.

`from math import factorial` satırı yalnız `factorial` adını dahil ettiğinden fonksiyon `math.` yazmadan çağrılabilir.
Çıktı:

```
[1, 2, 6, 24, 120, 720]
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
[1.0, 0.25, 0.1111111111111111, 0.0625]
```

■

Alıştırma 12.5 (Genel Terimi Verilen Diziler). Aşağıdaki programın çıktısını bulunuz.

```
print([5 * n + 3 for n in range(10)])
print([round(1 / (5 * n + 3), 3) for n in range(10)])
```

Çözüm.
Birinci liste genel terimi $a_n = 5n + 3$ olan aritmetik dizinin $n = 0, 1, \ldots, 9$ için terimleridir: 3'ten başlar ve 5'er artar. İkinci liste bu terimlerin terslerini üç basamağa yuvarlar: $1/3 \approx 0{,}333$, $1/8 = 0{,}125$, $1/13 \approx 0{,}077$ gibi. `round` sondaki sıfırları yazmadığı için $1/33 \approx 0{,}0303$ terimi `0.03` olarak görünür.
Çıktı:

```
[3, 8, 13, 18, 23, 28, 33, 38, 43, 48]
[0.333, 0.125, 0.077, 0.056, 0.043, 0.036, 0.03, 0.026, 0.023, 0.021]
```

■

Alıştırma 12.6 (Koşulla Seçilen Sayılar). Aşağıdaki programın çıktısını bulunuz.

```
print([x for x in range(50, 101) if x % 7 == 0])
print([x for x in range(20) if x % 3 == 0])
print([x for x in range(10) if x % 2 == 1])
print([x for x in range(10) if x % 2 == 1 and x > 5])
```

Çözüm.

- 50 ile 100 arasındaki 7'nin katları: $56, 63, 70, 77, 84, 91, 98$.
- 0 ile 19 arasındaki 3'ün katları: $0, 3, 6, 9, 12, 15, 18$; 0 da 3'ün katıdır.
- 0 ile 9 arasındaki tek sayılar: $1, 3, 5, 7, 9$.
- Bunlardan 5'ten büyük olanlar: $7, 9$.

Çıktı:

 CC BY-NC-SA 4.0

```
[56, 63, 70, 77, 84, 91, 98]
[0, 3, 6, 9, 12, 15, 18]
[1, 3, 5, 7, 9]
[7, 9]
```

■

Alıştırma 12.7 (İç İçe Koşullu İfade). Aşağıdaki programın çıktısını bulunuz.

```
sayilar = [-5, 10, -2, 0, 8]
durum = ["p" if x > 0 else "n" if x < 0 else "s" for x in sayilar]
print(durum)
```

Çözüm.
Koşullu ifade iç içe yazılmıştır ve `"p" if x > 0 else ("n" if x < 0 else "s")` olarak okunur: sayı pozitifse `p`, negatifse `n`, ikisi de değilse, yani sıfırsa `s`. Sayılar sırasıyla negatif, pozitif, negatif, sıfır, pozitiftir.
Çıktı:

```
['n', 'p', 'n', 's', 'p']
```

■

Alıştırma 12.8 (Pisagor Üçlüleri). Aşağıdaki program $a^2 + b^2 = c^2$ eşitliğini sağlayan ve bütün sayıları 20'den küçük olan (a, b, c) pozitif tam sayı üçlülerini bulur. Çıktısını bulunuz.

```
pisagor = [(a, b, c) for a in range(1, 20)
           for b in range(a, 20)
           for c in range(b, 20)
           if a ** 2 + b ** 2 == c ** 2]
print(pisagor)
```

Çözüm.
Üç `for` iç içe dolaşılır: önce `a`, sonra her `a` için `b`, sonra her `b` için `c`. `b`'nin `a`'dan, `c`'nin `b`'den başlaması $a \le b \le c$ sırasını sağlar; böylece $(3, 4, 5)$ ile $(4, 3, 5)$ gibi aynı üçlü iki kez sayılmaz. Bulunan üçlüler $(3, 4, 5)$, $(5, 12, 13)$, $(6, 8, 10)$, $(8, 15, 17)$ ve $(9, 12, 15)$'tir; örneğin $5^2 + 12^2 = 25 + 144 = 169 = 13^2$. Üçlüler `a`'ya, sonra `b`'ye göre sıralı çıkar.
Çıktı:

```
[(3, 4, 5), (5, 12, 13), (6, 8, 10), (8, 15, 17), (9, 12, 15)]
```

$(6, 8, 10)$ ve $(9, 12, 15)$, $(3, 4, 5)$'in 2 ve 3 katıdır. ■

Alıştırma 12.9 (Çevresi 60 Olan Dik Üçgenler). Aşağıdaki program kenarları tam sayı ve çevresi 60 olan dik üçgenleri bulur. Çıktısını bulunuz.

```
s10 = [(a, b, 60 - a - b) for a in range(1, 60) for b in range(a, 60)
       if 60 - a - b > b and a ** 2 + b ** 2 == (60 - a - b) ** 2]
print(s10)
```

Çözüm.

 CC BY-NC-SA 4.0

Çevre 60 olduğundan hipotenüs $c = 60 - a - b$ ile hesaplanır; böylece üç değil iki döngü yeter. `60 - a - b > b` koşulu hipotenüsün en uzun kenar olmasını sağlar. Bulunan üçlüler $(10, 24, 26)$ ve $(15, 20, 25)$'tir: $100 + 576 = 676 = 26^2$ ve $225 + 400 = 625 = 25^2$; ikisinin de çevresi 60'tır.
Çıktı:

```
[(10, 24, 26), (15, 20, 25)]
```

■

Alıştırma 12.10 (Ortak Katlar ve Matris Düzleştirme). Aşağıdaki programın çıktısını bulunuz.

```
s1 = [n for n in range(1, 31) if n % 3 == 0 and n % 5 == 0]
print(s1)
matris = [[1, 2], [3, 4], [5, 6]]
s3 = [x for satir in matris for x in satir]
print(s3)
```

Çözüm.
Hem 3'e hem 5'e bölünen sayılar 15'in katlarıdır; 30'a kadar 15 ve 30. İkinci liste üretecinde dıştaki `for` matrisin satırlarını, içteki `for` her satırın elemanlarını dolaşır; bütün elemanlar tek bir listede, satır satır sıralanır. Buna matrisi **düzleştirmek** denir.
Çıktı:

```
[15, 30]
[1, 2, 3, 4, 5, 6]
```

■

Alıştırma 12.11 (Zar Çiftleri ve Rakamlar Toplamı). Aşağıdaki programın çıktısını bulunuz.

```
s5 = [(z1, z2) for z1 in range(1, 7) for z2 in range(1, 7)
      if z1 + z2 == 7]
print(s5)
s6 = [n for n in range(1, 100) if sum(int(r) for r in str(n)) == 10]
print(s6)
```

Çözüm.
Birinci liste iki zarın toplamı 7 olan 6 sonucunu verir: $(1, 6), (2, 5), \ldots, (6, 1)$. İkinci liste üretecinde her sayı metne çevrilir, her karakter yeniden rakama çevrilir ve rakamlar toplanır. 100'den küçük ve rakamları toplamı 10 olan sayılar $19, 28, 37, 46, 55, 64, 73, 82, 91$'dir; tek basamaklı hiçbir sayının rakam toplamı 10 olamaz.
Çıktı:

```
[(1, 6), (2, 5), (3, 4), (4, 3), (5, 2), (6, 1)]
[19, 28, 37, 46, 55, 64, 73, 82, 91]
```

■

Alıştırma 12.12 (Palindrom Sayılar). Tersten okunuşu da aynı olan sayılara **palindrom** denir: 121 gibi. Aşağıdaki programın çıktısını bulunuz.

 CC BY-NC-SA 4.0

```
s7 = [n for n in range(100, 201) if str(n) == str(n)[::-1]]
print(s7)
```

Çözüm.
`str(n)[::-1]` metni tersine çevirir; sayı, metni kendi tersine eşitse palindromdur. 100 ile 200 arasında üç basamaklı bir palindromun ilk ve son rakamı 1 olmalıdır; ortadaki rakam 0'dan 9'a kadar herhangi biri olabilir. Bu yüzden on palindrom vardır: $101, 111, \ldots, 191$. 200 palindrom değildir, çünkü tersi 002'dir.
Çıktı:

```
[101, 111, 121, 131, 141, 151, 161, 171, 181, 191]
```

■

Alıştırma 12.13 (İki Karenin Toplamı 50). Aşağıdaki programın çıktısını bulunuz.

```
s9 = [(x, y) for x in range(1, 8) for y in range(1, 8)
      if x ** 2 + y ** 2 == 50]
print(s9)
```

Çözüm.
x ve y 1'den 7'ye kadar dolaşılır; $8^2 = 64 > 50$ olduğundan daha büyük değerlere gerek yoktur. $50 = 1 + 49 = 25 + 25 = 49 + 1$ olduğundan çiftler $(1, 7)$, $(5, 5)$ ve $(7, 1)$'dir. Sıra önemli olduğu için $(1, 7)$ ve $(7, 1)$ ayrı çiftlerdir.
Çıktı:

```
[(1, 7), (5, 5), (7, 1)]
```

■

Alıştırma 12.14 (Bir Listenin Elemanlarını Dönüştürmek). Aşağıdaki programın çıktısını bulunuz.

```
fahrenheit = [32, 68, 100, 212]
print([round((f - 32) * 5 / 9, 3) for f in fahrenheit])
print([x ** 0.5 for x in [4, 9, 16, 25, 36]])
print([abs(x) for x in [-5, 10, -3.14, 0, -8]])
```

Çözüm.
Birinci liste $C = (F - 32) \cdot 5/9$ dönüşümünü uygular: $32\,^\circ\mathrm{F} = 0\,^\circ\mathrm{C}$, $68\,^\circ\mathrm{F} = 20\,^\circ\mathrm{C}$, $100\,^\circ\mathrm{F} \approx 37{,}778\,^\circ\mathrm{C}$, $212\,^\circ\mathrm{F} = 100\,^\circ\mathrm{C}$. İkinci liste kareköklerdir; `** 0.5` sonucu `float` verir. Üçüncü liste mutlak değerlerdir; `abs` tam sayıları tam sayı, ondalıklı sayıları ondalıklı bırakır.
Çıktı:

```
[0.0, 20.0, 37.778, 100.0]
[2.0, 3.0, 4.0, 5.0, 6.0]
[5, 10, 3.14, 0, 8]
```

■

 CC BY-NC-SA 4.0

12.3 Döngüler ve Koşullar

Bu kısımda bir listenin her elemanı için hangi koşulun doğru çıktığını ve elemanın hangi listeye hangi değerle eklendiğini tek tek belirleyin.

Alıştırma 12.15 (İki Listeye Toplam ve Kuvvet). Aşağıdaki programın çıktısını bulunuz.

```
birinci = []
ikinci = []
for x in range(2, 7, 3):
    for y in range(3, 13, 5):
        birinci.append(x + y)
        ikinci.append(x ** y)
print("birinci:", birinci)
print("ikinci:", ikinci)
```

Çözüm.
`range(2, 7, 3)` 2 ve 5'i, `range(3, 13, 5)` 3 ve 8'i verir. Dört çift sırayla $(2,3)$, $(2,8)$, $(5,3)$, $(5,8)$'dir. Toplamlar $5, 10, 8, 13$; kuvvetler $2^3 = 8$, $2^8 = 256$, $5^3 = 125$ ve $5^8 = 390625$'tir.
Çıktı:

```
birinci: [5, 10, 8, 13]
ikinci: [8, 256, 125, 390625]
```

■

Alıştırma 12.16 (İşarete Göre İki Liste). Aşağıdaki programın çıktısını bulunuz.

```
sayilar = [1, -5, 6, -8, 3]
negatif = []
pozitif = []
for x in sayilar:
    if x < 0:
        negatif.append(x ** 2 + 1)
    else:
        pozitif.append(2 * x ** 2 + 1)
print("pozitif:", pozitif)
print("negatif:", negatif)
```

Çözüm.
Negatif sayılar $(-5, -8)$ için $x^2 + 1$ hesaplanır: 26 ve 65. Diğerleri $(1, 6, 3)$ için $2x^2 + 1$ hesaplanır: 3, 73 ve 19. Listelerin adları yalnız hangi sayıların nereye gittiğini anlatır; içlerindeki değerlerin hepsi pozitiftir.
Çıktı:

```
pozitif: [3, 73, 19]
negatif: [26, 65]
```

■

Alıştırma 12.17 (Her Zaman Doğru Bir Koşul). Aşağıdaki programın çıktısını bulunuz.

```
sayilar = [1, -5, 6, -8, 3]
negatif = []
pozitif = []
```

```
for x in sayilar:
    if (2 * x ** 2 + 1 > 5) or (2 * x ** 2 + 1 < 45):
        negatif.append(x ** 2 + 1)
    else:
        pozitif.append(2 * x ** 2 + 1)
print("negatif:", negatif)
print("pozitif:", pozitif)
```

Çözüm.
Her sayı için $2x^2 + 1$ değerleri $3, 51, 73, 129, 19$'dur. Bir sayı ya 5'ten büyüktür ya da 45'ten küçüktür; her gerçel sayı bu iki koşuldan en az birini sağlar. Bu yüzden `or` ile bağlanmış koşul her eleman için doğrudur, bütün sayılar `negatif` listesine $x^2 + 1$ olarak girer ve `pozitif` boş kalır: $2, 26, 37, 65, 10$.
Çıktı:

```
negatif: [2, 26, 37, 65, 10]
pozitif: []
```

■

Alıştırma 12.18 (Bir Aralığa Düşen Değer). Aşağıdaki programın çıktısını bulunuz.

```
sayilar = [1, -5, 6, -8, 3]
negatif = []
pozitif = []
for x in sayilar:
    if (2 * x ** 2 + 1 > 5) and (2 * x ** 2 + 1 < 45):
        negatif.append(x ** 2 + 1)
        print(x)
    else:
        pozitif.append(2 * x ** 2 + 1)
print("negatif:", negatif)
print("pozitif:", pozitif)
```

Çözüm.
Bu kez $2x^2 + 1$'in 5 ile 45 arasında olması gerekir. Değerler $3, 51, 73, 129, 19$'dur; yalnız 19 (yani $x = 3$) bu aralıktadır. $x = 3$ yazılır ve $3^2 + 1 = 10$ `negatif`'e eklenir; diğerlerinin $2x^2 + 1$ değerleri `pozitif`'e eklenir.
Çıktı:

```
3
negatif: [10]
pozitif: [3, 51, 73, 129]
```

■

Alıştırma 12.19 (Üç Liste ve pass). Aşağıdaki programın çıktısını bulunuz.

```
sayilar = [1, -5, 6, -8, 3]
birinci = []
ikinci = []
ucuncu = []
for x in sayilar:
    if (2 * x ** 2 > 5) and (2 * x ** 2 < 45):
        birinci.append(x ** 2 + 1)
```

 CC BY-NC-SA 4.0

```
    elif 2 * x ** 3 > 5:
        ikinci.append(2 * x ** 2 + 1)
    elif 2 * x ** 4 > 15:
        ucuncu.append(x)
    else:
        pass
        print("Vazgeçtim")
birinci.sort()
ikinci.sort()
ucuncu.sort()
print("birinci:", birinci)
print("ikinci:", ikinci)
print("ucuncu:", ucuncu)
```

Çözüm.
Her sayı için koşulları sırayla deneyelim:

- $x = 1$: $2x^2 = 2$, birinci koşul yanlış; $2x^3 = 2 > 5$ yanlış; $2x^4 = 2 > 15$ yanlış. `else` çalışır; `pass` hiçbir şey yapmaz ama sonraki `print` çalışır ve `Vazgeçtim` yazılır.
- $x = -5$: $2x^2 = 50$, 45'ten büyük; $2x^3 = -250$; $2x^4 = 1250 > 15$. `ucuncu`'ye -5 eklenir.
- $x = 6$: $2x^2 = 72$; $2x^3 = 432 > 5$. `ikinci`'ye $2 \cdot 36 + 1 = 73$ eklenir.
- $x = -8$: $2x^2 = 128$; $2x^3 = -1024$; $2x^4 = 8192 > 15$. `ucuncu`'ye -8 eklenir.
- $x = 3$: $2x^2 = 18$, 5 ile 45 arasında. `birinci`'ye 10 eklenir.

Son olarak listeler sıralanır; `ucuncu [-8, -5]` olur.
Çıktı:

```
Vazgeçtim
birinci: [10]
ikinci: [73]
ucuncu: [-8, -5]
```

`pass` bloğu boş bırakmamak için konur; ardından gelen komutların çalışmasını engellemez. ■

Alıştırma 12.20 (Koşullu Liste Üreteçleri). Aşağıdaki programın çıktısını bulunuz.

```
print([x for x in range(10) if 2 * x ** 2 + 1 >= 30])
liste = [-5, 3, -6, 1, 2, 9]
print([x for x in liste if x ** 2 + 1 >= 15 or x + 5 > 15])
print([x for x in liste if x ** 2 + 1 >= 15 and x + 5 > 15])
print([x for x in liste if (x ** 2 + 1) // 2 == 1])
print([x for x in liste if (x ** 2 + 1) % 2 == 1])
```

Çözüm.

- $2x^2 + 1 \geq 30$ ancak $x^2 \geq 14{,}5$, yani $x \geq 4$ ise sağlanır: $4, 5, \ldots, 9$.
- Listedeki sayılar için $x^2 + 1$ değerleri $26, 10, 37, 2, 5, 82$'dir. $x^2 + 1 \geq 15$ olanlar -5, -6 ve 9'dur; $x + 5 > 15$ ise hiçbir sayı için doğru değildir. `or` ile sonuç $-5, -6, 9$'dur.
- `and` ile iki koşulun birden sağlanması gerekir; ikinci koşul hiç sağlanmadığı için liste boştur.
- $(x^2 + 1)\,//\,2 = 1$ ancak $x^2 + 1$ değeri 2 ya da 3 ise doğrudur; yalnız $x = 1$ ($x^2 + 1 = 2$) sağlar.
- $x^2 + 1$ tek ancak x çift ise: -6 ve 2.

Çıktı:

```
[4, 5, 6, 7, 8, 9]
[-5, -6, 9]
[]
```

 CC BY-NC-SA 4.0

```
[1]
[-6, 2]
```

■

Alıştırma 12.21 (math Fonksiyonlarıyla Bir Toplam). Aşağıdaki programın çıktısını bulunuz.

```
import math
print(math.sqrt(25) + math.pow(2, 3) + math.fabs(-5.5))
x = 3.7
print(math.ceil(x) + math.floor(x))
```

Çözüm.
$\sqrt{25} = 5{,}0$, $2^3 = 8{,}0$ ve $|-5{,}5\ | = 5{,}5$'tir; üçü de `float`'tur ve toplamları $18{,}5$'tir. $\lceil 3{,}7 \rceil = 4$, $\lfloor 3{,}7 \rfloor = 3$; ikisi de tam sayıdır ve toplamları 7'dir.
Çıktı:

```
18.5
7
```

■

12.4 while Döngüleri

Bu kısımdaki her problemde, değişkenlerin her yinelemedeki değerlerini bir tabloya yazarak ilerleyin.

Alıştırma 12.22 (Fibonacci Benzeri Bir Dizi). Aşağıdaki programın çıktısını bulunuz.

```
n = 5
a, b = 3, 2
sayac = 0
while sayac < n:
    a, b = b, a + b
    sayac += 1
    print("a:", a, ", b:", b)
print(b)
```

Çözüm.
Her adımda yeni `a` eski `b`, yeni `b` eski iki değerin toplamı olur; bu, Fibonacci dizisinin kuralıdır ama başlangıç değerleri farklıdır:

sayac	a	b
başta	3	2
1	2	5
2	5	7
3	7	12
4	12	19
5	19	31

Çıktı:

 CC BY-NC-SA 4.0

```
a: 2 , b: 5
a: 5 , b: 7
a: 7 , b: 12
a: 12 , b: 19
a: 19 , b: 31
31
```

■

Alıştırma 12.23 (Ara Toplamların Ortalaması). Aşağıdaki programın çıktısını bulunuz.

```
N = 10
X = N - (N % 3)
T = 0
sayac = 0
Tlist = []
while X > 0:
    T = T + X
    X = X - 3
    sayac = sayac + 1
    Tlist.append(T)
    print("X:", X, ", T:", T, ", sayac:", sayac, ", Tlist:", Tlist)
print("Ortalama =", sum(Tlist) / sayac)
```

Çözüm.
X başta $10 - 1 = 9$'dur ve 3'er azalır. Her adımda T'ye o anki X eklenir, sonra X azaltılır:

adım	eklenen X	T	yeni X	Tlist
1	9	9	6	[9]
2	6	15	3	[9, 15]
3	3	18	0	[9, 15, 18]

$X = 0$ olunca döngü biter. Ortalama, 3'ün katlarının değil **ara toplamların** ortalamasıdır: $(9 + 15 + 18)/3 = 42/3 = 14{,}0$. 3'ün katlarının ortalaması isteniyor olsaydı $T/\text{ sayac} = 18/3 = 6{,}0$ hesaplanmalıydı.
Çıktı:

```
X: 6 , T: 9 , sayac: 1 , Tlist: [9]
X: 3 , T: 15 , sayac: 2 , Tlist: [9, 15]
X: 0 , T: 18 , sayac: 3 , Tlist: [9, 15, 18]
Ortalama = 14.0
```

■

Alıştırma 12.24 (continue Öncesi Güncelleme). Aşağıdaki programın çıktısını bulunuz.

```
s = 7
a = 1
b = 3
while s < 10:
    s = s + 1
    a = a + b
```

 CC BY-NC-SA 4.0

```
    b = b + 2
    print("s:", s, " a:", a, " b:", b)
    if s == 8 or s == 5:
        a = b - 2
        b = b + 2
        continue
print("a =", a, " b =", b)
```

Çözüm.
Adımları izleyelim:

- Birinci yineleme: $s = 8$, $a = 1 + 3 = 4$, $b = 5$; bu değerler yazılır. $s = 8$ olduğu için `if` çalışır: $a = 5 - 2 = 3$, $b = 7$; `continue` döngünün başına döner.
- İkinci yineleme: $s = 9$, $a = 3 + 7 = 10$, $b = 9$; yazılır. Koşul yanlıştır.
- Üçüncü yineleme: $s = 10$, $a = 10 + 9 = 19$, $b = 11$; yazılır. Koşul yanlıştır. $10 < 10$ yanlış olduğundan döngü biter.

`if` bloğundaki değişiklikler, `print`'ten sonra yapıldığı için yazdırılan satırlarda görünmez; ama bir sonraki yinelemenin hesabını etkiler. `continue` döngünün son satırı olduğundan burada hiçbir şeyi atlamaz.
Çıktı:

```
s: 8  a: 4  b: 5
s: 9  a: 10  b: 9
s: 10  a: 19  b: 11
a = 19  b = 11
```

■

Alıştırma 12.25 (Azalan Çarpanlarla Bir Liste). Aşağıdaki programın çıktısını bulunuz.

```
x = 1
y = 1
z = 5
k = []
while True:
    y = y * z
    k.append(y)
    z -= 1
    if z < x:
        break
print(k)
print(sum(k))
print(k[1] + k[3])
```

Çözüm.
`y` sırayla 5, 4, 3, 2, 1 ile çarpılır ve her ara çarpım listeye eklenir: $5,\ 5 \cdot 4 = 20,\ 20 \cdot 3 = 60,\ 60 \cdot 2 = 120,\ 120 \cdot 1 = 120$. `z` 0 olduğunda $0 < 1$ doğru olur ve `break` döngüyü bitirir. Toplam $5 + 20 + 60 + 120 + 120 = 325$; `k[1] + k[3]` $20 + 120 = 140$'tır.
Çıktı:

```
[5, 20, 60, 120, 120]
325
140
```

■

 CC BY-NC-SA 4.0

12.5 Demetler

Bu kısımda dilimlerin ve işlemlerin sonucunun bir demet mi yoksa bir liste mi olduğuna dikkat edin.

Alıştırma 12.26 (Demet Dilimleri). Aşağıdaki programın çıktısını bulunuz.

```
demet = (10, 51, 66, 13, 48, 15)
print(demet[-3:])
print(demet[::-1])
print(demet[-4:-1])
print(demet[2:6:3])
print(demet[::2])
```

Çözüm.
Demetler listelerle aynı kurallarla dilimlenir; sonuç yine bir demettir.

- `demet[-3:]`: son üç eleman.
- `demet[::-1]`: ters sıra.
- `demet[-4:-1]`: indeks -4 (yani 2) ile -2 (yani 4) arası.
- `demet[2:6:3]`: indeks 2 ve 5: 66 ve 15.
- `demet[::2]`: indeks 0, 2, 4: 10, 66, 48.

Çıktı:

```
(13, 48, 15)
(15, 48, 13, 66, 51, 10)
(66, 13, 48)
(66, 15)
(10, 66, 48)
```

■

Alıştırma 12.27 (Demeti ve Listeyi Çoğaltmak). Aşağıdaki programın çıktısını bulunuz.

```
t = (121, 352, 431, 245, 567)
print(t[1:3])
print(len(t))
print(t * 2)
print(list(t) * 2)
```

Çözüm.
`t[1:3]` indeks 1 ve 2'deki elemanlardan bir demettir. `t * 2` demeti iki kez yan yana yazar ve bir demet verir; `list(t) * 2` aynı işi önce listeye çevirerek yapar ve bir liste verir. Çıktıdaki fark yalnız parantezlerin türündedir.
Çıktı:

```
(352, 431)
5
(121, 352, 431, 245, 567, 121, 352, 431, 245, 567)
[121, 352, 431, 245, 567, 121, 352, 431, 245, 567]
```

■

Alıştırma 12.28 (Paketlenmiş Bir Demet). Aşağıdaki programın çıktısını bulunuz.

 CC BY-NC-SA 4.0

```
x = 3, 7, 10
print("Demet:", x)
for i in x:
    if i % 2 == 0:
        print(f"{i} çift")
    else:
        print(f"{i} tek")
```

Çözüm.
Parantezsiz yazılan `3, 7, 10` bir demete paketlenir (Tanım 9.1). Döngü elemanları sırayla alır: 3 ve 7 tek, 10 çifttir.
Çıktı:

```
Demet: (3, 7, 10)
3 tek
7 tek
10 çift
```

■

12.6 Algoritmalar

Bu kısımdaki programlar küçük ama tam birer algoritmadır; önce ne yaptıklarını anlamaya, sonra çıktılarını bulmaya çalışın.

Alıştırma 12.29 (En Uzun Collatz Zinciri). Bir n sayısı çiftse yarıya bölünür, tekse $3n + 1$ yapılır; bu kural 1'e ulaşılana kadar tekrarlanır. Aşağıdaki program 1'den 30'a kadar başlangıç değerleri arasında 1'e en çok adımda ulaşanı bulur. Ne yaptığını açıklayınız ve çıktısını bulunuz.

```
en_uzun = 0
en_iyi_sayi = 0
for baslangic in range(1, 31):
    n = baslangic
    adim = 0
    while n != 1:
        if n % 2 == 0:
            n //= 2
        else:
            n = 3 * n + 1
        adim += 1
    if adim > en_uzun:
        en_uzun = adim
        en_iyi_sayi = baslangic
print(f"En uzun zincir: {en_iyi_sayi} sayısı, {en_uzun} adım")
```

Çözüm.
Dıştaki `for` her başlangıç değerini dener. İçteki `while`, kaç adım süreceği önceden bilinmediği için kullanılmıştır: kural uygulanır ve `n` 1 olana kadar adımlar sayılır. Örneğin

$$6 \to 3 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$$

zinciri 8 adımdır. Her başlangıç değerinin adım sayısı şimdiye kadarki en büyük adım sayısıyla karşılaştırılır; daha büyükse en uzun zincir güncellenir.
27'den başlayan zincir bu aralıktaki en uzun zincirdir: 9232'ye kadar yükseldikten sonra 111 adımda 1'e iner.
Çıktı:

 CC BY-NC-SA 4.0

```
En uzun zincir: 27 sayısı, 111 adım
```

Her başlangıç değerinden bu kuralla sonunda 1'e ulaşıldığı tahmin edilir; ama bu tahmin, **Collatz sanısı** adıyla bilinen, henüz kanıtlanmamış bir problemdir. ■

Alıştırma 12.30 (Kabarcık Sıralaması). Aşağıdaki program bir listeyi hazır `sort` metodunu kullanmadan sıralar. Nasıl çalıştığını açıklayınız ve listenin her turdan sonraki hâlini bulunuz.

```
sayilar = [64, 34, 25, 12, 22, 11, 90]
n = len(sayilar)
degisti = True
tur = 0
while degisti:
    degisti = False
    for i in range(n - 1 - tur):
        if sayilar[i] > sayilar[i + 1]:
            sayilar[i], sayilar[i + 1] = sayilar[i + 1], sayilar[i]
            degisti = True
    tur += 1
    print(tur, sayilar)
print("Sıralı liste:", sayilar)
```

Çözüm.
İçteki `for` yan yana duran iki elemanı karşılaştırır; soldaki büyükse ikisini takasla yer değiştirir. Bir tur boyunca en büyük eleman, suyun içindeki bir kabarcık gibi adım adım sağa taşınır ve sonunda listenin en sağına yerleşir. Bu yüzden yönteme **kabarcık sıralaması** (bubble sort) denir. Her turdan sonra sondaki bir eleman daha yerini bulduğu için içteki döngünün aralığı `tur` kadar kısalır.
`degisti` değişkeni bir turda hiç takas yapılıp yapılmadığını tutar. Hiç takas yapılmayan bir tur, listenin sıralı olduğunu gösterir; o zaman `while` döngüsü biter. Burada beşinci turdan sonra liste sıralıdır; altıncı turda hiç takas yapılmaz ve döngü durur.
Çıktı:

```
1 [34, 25, 12, 22, 11, 64, 90]
2 [25, 12, 22, 11, 34, 64, 90]
3 [12, 22, 11, 25, 34, 64, 90]
4 [12, 11, 22, 25, 34, 64, 90]
5 [11, 12, 22, 25, 34, 64, 90]
6 [11, 12, 22, 25, 34, 64, 90]
Sıralı liste: [11, 12, 22, 25, 34, 64, 90]
```

■

Alıştırma 12.31 (İç İçe for ile Sıralama). Aşağıdaki programın çıktısını bulunuz ve dıştaki döngünün her adımda neyi garanti ettiğini açıklayınız.

```
dizi = [5, 2, 9, 1]
n = len(dizi)
for i in range(n - 1):
    for j in range(i + 1, n):
        if dizi[i] > dizi[j]:
            dizi[i], dizi[j] = dizi[j], dizi[i]
    print(i, dizi)
```

Çözüm.

 CC BY-NC-SA 4.0

Dıştaki döngü `i` indeksli yeri doldurur. İçteki döngü `i`'den sonraki bütün elemanları `dizi[i]` ile karşılaştırır ve daha küçük bir eleman bulduğunda takas eder; içteki döngü bittiğinde `i` indeksinde kalan elemanların en küçüğü bulunur.

- `i = 0`: $5 > 2$ takas, $[2, 5, 9, 1]$; $2 > 9$ değil; $2 > 1$ takas, $[1, 5, 9, 2]$.
- `i = 1`: $5 > 9$ değil; $5 > 2$ takas, $[1, 2, 9, 5]$.
- `i = 2`: $9 > 5$ takas, $[1, 2, 5, 9]$.

Çıktı:

```
0 [1, 5, 9, 2]
1 [1, 2, 9, 5]
2 [1, 2, 5, 9]
```

Her `i` adımından sonra ilk `i + 1` eleman listenin en küçük `i + 1` elemanıdır ve sıralıdır. ■

Alıştırma 12.32 (Bir Kümeden Seçilenlerin Toplamı). Aşağıdaki programın çıktısını bulunuz.

```
rakamlar = {1, 5, 7, 9, 2, 4, 6, 8, 3, 0}
secilenler = {1, 2, 3, 4, 5}
print(sum(r for r in rakamlar if r in secilenler))
print(sum(s for s in range(1, 101) if s % 2 == 0))
sayilar = [-3, -2, -1, 0, 1, 2, 3, 4]
print(sum(x * x for x in sayilar if x > 0))
```

Çözüm.
Birinci satır `rakamlar` kümesinde olup `secilenler` kümesinde de bulunan sayıları toplar: $1 + 2 + 3 + 4 + 5 = 15$. Kümelerin sırasız olması toplamı etkilemez. İkinci satır

$$2 + 4 + \cdots + 100 = 2(1 + 2 + \cdots + 50) = 2 \cdot 1275 = 2550$$

'yi hesaplar. Üçüncü satır pozitif sayıların karelerini toplar: $1 + 4 + 9 + 16 = 30$.
Çıktı:

```
15
2550
30
```

■

Alıştırma 12.33 (İkinci Dereceden Bir Koşul). Aşağıdaki programın çıktısını bulunuz.

```
sayilar = [-3, -2, -1, 0, 1, 2, 3, 4]
print(sum(x * x for x in sayilar if x ** 2 - 5 * x > 0))
print(max(x * x for x in sayilar if x ** 2 - 5 * x > 0))
```

Çözüm.
$x^2 - 5x = x(x - 5) > 0$ eşitsizliği $x < 0$ ya da $x > 5$ için sağlanır. Listede 5'ten büyük sayı olmadığından seçilen sayılar $-3, -2, -1$'dir. Karelerinin toplamı $9 + 4 + 1 = 14$, en büyüğü 9'dur.
Çıktı:

```
14
9
```

■

 CC BY-NC-SA 4.0

Alıştırma 12.34 (Geçer Notların Ortalaması). Aşağıdaki programın çıktısını bulunuz.

```
notlar = [45, 78, 92, 36, 67, 81, 53, 29]
gecer = [n for n in notlar if n >= 50]
print(gecer)
ortalama = sum(gecer) / len(gecer) if gecer else 0
print(f"Geçer not ortalaması: {ortalama:.2f}")
```

Çözüm.
50 ve üstündeki notlar $78, 92, 67, 81, 53$'tür. İkinci satırdaki koşullu ifade, liste boşsa sıfıra bölmemek için yazılmıştır: boş bir liste yanlış sayılır ve o durumda ortalama 0 alınır. Burada liste boş değildir; ortalama

$$(78 + 92 + 67 + 81 + 53)/5 = 371/5 = 74{,}2$$

'dir.
Çıktı:

```
[78, 92, 67, 81, 53]
Geçer not ortalaması: 74.20
```

■

Alıştırma 12.35 (Derin İç İçe Koşullar). Aşağıdaki programın çıktısını bulunuz.

```
x, y, z = 12, 5, 8
toplam = x + y + z
fark = x - y - z
if toplam > 20:
    print("Toplam 20'den büyük!")
    if x > y:
        print("x, y'den büyük")
        carpim = x * y
        if carpim % 2 == 0:
            print(f"Çarpım çift: {carpim}")
            if z > 0:
                print("z pozitif")
                sonuc = (x * y) - (y * z) + (x % z)
                print(f"Matematiksel sonuç: {sonuc}")
                if sonuc > 30:
                    print("Sonuç 30'dan büyük")
                else:
                    print("Sonuç 30'dan küçük")
        else:
            print(f"Çarpım tek: {carpim}")
elif toplam > 15:
    print("Toplam 15-20 arası")
else:
    print("Toplam 15'ten küçük")
    if fark < 0:
        print("Fark negatif")
```

Çözüm.
toplam $= 25 > 20$ olduğundan ilk kola girilir. $12 > 5$ doğrudur; çarpım 60 çifttir; $z = 8 > 0$'dır.

$$\text{sonuc} = 60 - 40 + (12 \bmod 8) = 60 - 40 + 4 = 24$$

 CC BY-NC-SA 4.0

bulunur ve $24 > 30$ yanlış olduğu için son `else` çalışır. Her koşul bir öncekinin içinde olduğu için girintiler her seviyede biraz daha sağa kayar.
Çıktı:

```
Toplam 20'den büyük!
x, y'den büyük
Çarpım çift: 60
z pozitif
Matematiksel sonuç: 24
Sonuç 30'dan küçük
```

■

Alıştırma 12.36 (Aynı Koşullar, Farklı Değerler). Bir önceki problemdeki programın ilk satırı `x, y, z = 2, 1, 8` olarak değiştirilirse çıktı ne olur?

```
x, y, z = 2, 1, 8
toplam = x + y + z
fark = x - y - z
if toplam > 20:
    print("Toplam 20'den büyük!")
elif toplam > 15:
    print("Toplam 15-20 arası")
else:
    print("Toplam 15'ten küçük")
    if fark < 0:
        print("Fark negatif")
```

Çözüm.
Bu kez toplam $= 11$'dir; ilk iki koşul yanlıştır ve `else` kolu çalışır. İlk koldaki iç içe koşullara hiç girilmediği için onları yukarıdaki kodda yazmadık; çıktıyı değiştirmezler. fark $= 2 - 1 - 8 = -7 < 0$ olduğundan ikinci mesaj da yazılır.
Çıktı:

```
Toplam 15'ten küçük
Fark negatif
```

■

12.7 İç İçe Toplamlar

Bu kısımda matematikteki iki katlı toplamları iç içe döngülerle hesaplayacağız; dıştaki toplamın değişkeni dıştaki döngüye, içteki toplamın değişkeni içteki döngüye karşılık gelir.

Alıştırma 12.37 (Sınırları Sabit Bir Çift Toplam).

$$\sum_{x=3}^{7}\sum_{y=1}^{3}(x^2+y)$$

toplamını iç içe döngülerle hesaplayınız ve elle doğrulayınız.

```
toplam = 0
for x in range(3, 8):
    for y in range(1, 4):
        toplam += x ** 2 + y
```

 CC BY-NC-SA 4.0

```
print("Toplam:", toplam)
print(sum(x ** 2 + y for x in range(3, 8) for y in range(1, 4)))
```

Çözüm.
Üst sınırlar dahil olduğu için `range(3, 8)` ve `range(1, 4)` yazılır. Elle hesaplamak için içteki toplamı her x için açalım:

$$\sum_{y=1}^{3}(x^2+y) = 3x^2 + (1+2+3) = 3x^2 + 6$$

. Dıştaki toplam terim terim

$$(3\cdot 9+6)+(3\cdot 16+6)+(3\cdot 25+6)+(3\cdot 36+6)+(3\cdot 49+6)$$

olur, yani $33+54+81+114+153 = 435$. Program aynı toplamı önce iç içe döngüyle, sonra iki `for`'lu bir üreteç ifadesiyle hesaplar.
Çıktı:

```
Toplam: 435
435
```

■

Alıştırma 12.38 (İç Sınırı Dış Değişkene Bağlı Toplam).

$$\sum_{x=3}^{7}\sum_{y=1}^{x}(x^2+y)$$

toplamını iç içe döngülerle hesaplayınız ve elle doğrulayınız.

```
toplam = 0
for x in range(3, 8):
    for y in range(1, x + 1):
        toplam += x ** 2 + y
print(toplam)
print(sum(x ** 2 + y for x in range(3, 8) for y in range(1, x + 1)))
```

Çözüm.
Bu kez içteki toplamın üst sınırı x'tir; içteki döngü `range(1, x + 1)` olur ve her x için farklı sayıda terim toplanır. Elle:

$$\sum_{y=1}^{x}(x^2+y) = x\cdot x^2 + \frac{x(x+1)}{2} = x^3 + \frac{x(x+1)}{2}$$

. Terimler

- $x=3$: $27+6=33$,
- $x=4$: $64+10=74$,
- $x=5$: $125+15=140$,
- $x=6$: $216+21=237$,
- $x=7$: $343+28=371$

olur ve toplam $33+74+140+237+371 = 855$'tir.
Çıktı:

 CC BY-NC-SA 4.0

```
855
855
```

■

Bu problemlerle dersin ilk yarısının konularını tamamladık. Değişkenlerden döngülere kadar öğrendiğimiz araçlar, ilerleyen bölümlerde fonksiyonlar ve daha büyük programlar yazmanın temelini oluşturacak. Notların başına dönmek için Hazırlık: Python ve Çalışma Ortamı bölümüne bakılabilir.

 CC BY-NC-SA 4.0