Python Kılavuzu

Sürüm 3.x

Fırat Özgül (istihza)

17/04/2011

İÇİNDEKİLER

Uyarı: Aşağıdaki bilgiler Python'un 3.x sürümleri içindir. Eğer kullandığınız sürüm Python'un 2.x sürümlerinden biriyse şuradaki belgeleri inceleyebilirsiniz. Aşağıda gördüğünüz belgelendirme çalışması henüz tamamlanmamıştır. Bu bölüm sıklıkla güncellenecek, buraya yeni bölümler eklenecektir...

BÖLÜM 1

Temel Bilgiler

Python, Guido Van Rossum adlı Hollandalı bir programcı tarafından 90'lı yılların başında geliştirilmeye başlanmış bir programlama dilidir. Zannedildiğinin aksine bu programlama dilinin adı piton yılanından gelmez... Guido Van Rossum bu programlama dilini, "*The Monty Python*" adlı bir İngiliz komedi grubunun, "*Monty Python's Flying Circus*" adlı gösterisinden esinlenerek adlandırmıştır.

Python, pek çok dile kıyasla öğrenmesi kolay bir programlama dilidir. Bu yüzden, eğer daha önce hiç programlama deneyiminiz olmamışsa, programlama maceranıza Python'la başlamayı tercih edebilirsiniz.

Python programlarının en büyük özelliklerinden birisi, C ve C++ gibi dillerin aksine, derlenmeye gerek olmadan çalıştırılabilmeleridir. Python'da derleme işlemi ortadan kaldırıldığı için, Python'la oldukça hızlı bir şekilde program geliştirilebilir.

Ayrıca Python programlama dilinin basit ve temiz sözdizimi, onu pek çok programcı tarafından tercih edilen bir dil haline getirmiştir. Python'un sözdiziminin temiz ve basit olması sayesinde hem program yazmak, hem de başkası tarafından yazılmış bir programı okumak çok kolaydır.

Python'u kullanabilmek için öncelikle onu bilgisayarımıza kurmamız gerekiyor. İsterseniz sözü daha fazla uzatmadan Python'u nereden ve nasıl edinebileceğimizi öğrenelim.

1.1 Python Nereden İndirilir?

Python'un en yeni sürümü 3.x numaralıdır. Şu anda en çok kullanılan ve en yaygın sürümler ise 2.x numaralı olanlardır. Pek çok GNU/Linux dağıtımında Python kurulu olarak gelir. Eğer siz de bir GNU/Linux kullanıcısı iseniz muhtemelen sisteminizde Python zaten kuruludur. Ancak sisteminizde kurulu olan bu sürüm büyük ihtimalle Python'un 2.x numaralı bir sürümüdür.

Windows sistemlerinde ise herhangi bir Python sürümü kurulu olarak gelmez. Biz bu sitede Python'un 3.x sürümlerini belgelendireceğiz. Dilerseniz gelin şimdi Python'un 3.x sürümünün GNU/Linux ve Windows'ta nasıl kurulacağını ayrı ayrı inceleyelim:

GNU/Linux Kullanıcıları

Dediğim gibi, eğer GNU/Linux dağıtımlarından birini kullanıyorsanız sisteminizde muhtemelen Python'un 2.x sürümlerinden biri zaten kuruludur. Bunu şu şekilde kontrol edebilirsiniz:

Komut satırında:

```
python
```

yazıp enter tuşuna bastığınızda, eğer karşınıza şuna benzer bir ekran geliyorsa, kurulu sürüm 2.x'tir:

```
Python 2.6.1+ (r261:67515, Mar  2 2009, 13:10:18)
[GCC 4.3.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Burada **2.6.1+** olarak görünen rakam Python'un sürümünü gösteriyor. Bu çıktıya göre, sisteminizdeki Python sürümü 2.6.1. Yani sizde Python'un 2.x numaralı bir sürümü kurulu...

Sisteminizdeki Python sürümünü öğrenmek için şöyle bir yol da izleyebilirsiniz:

Yine komut satırında:

```
python -V
```

yazıp enter'e bastığınızda hangi Python sürümünün sisteminizde kurulu olduğu doğrudan görünecektir...

Eğer Python'un 2.x sürümleriyle çalışmak isterseniz, bu sitede 2.x sürümlerine ait belgeler de bulunmaktadır. İstediğiniz buysa sizi http://www.istihza.com/icindekiler_python.html adresine alalım...

Eğer Python'un 3.x sürümünü kullanmak isterseniz okumaya devam ediniz...

GNU/Linux kullanıcılarının, Python'un 3.x sürümünü elde etmek için tercih edebileceği iki yol vardır. Birincisi ve benim size önereceğim yol, öncelikle kullandığınız dağıtımın paket yöneticisini kontrol etmenizdir. Python 3.x sisteminizde kurulu olmasa bile, dağıtımınızın depolarında bu sürüm paketlenmiş halde duruyor olabilir. O yüzden sisteminize uygun bir şekilde paket yöneticinizi açıp orada "python" şeklinde bir arama yapmanızı öneririm. Örneğin Debian tabanlı bir GNU/Linux dağıtımı kullanıyorsanız komut satırında şu komutu vererek depolarınızdaki Python paketlerini listeleyebilirsiniz:

```
aptitude search python
```

Eğer bu komutun çıktısı içinde "python3" gibi bir şey görüyorsanız, kurmanız gereken paket odur... Yine Debian tabanlı sistemlerde bu paketi şu şekilde kurabilirsiniz:

```
sudo apt-get install python3
```

Örneğin Kubuntu, Debian tabanlı bir GNU/Linux dağıtımıdır. Eğer Kubuntu'nun "*Jaunty Jackalope*" veya daha üst sürümlerinden birini kullanıyorsanız, yukarıdaki komut yardımıyla python3 paketini sisteminize kurabilirsiniz.

Kullandığınız dağıtımın depolarında python3 paketini bulamazsanız, Python 3.x'i kaynaktan kurmanız gerekecektir. Bunun için yapmanız gereken işlemler şöyle:

1. Öncelikle şu adresi ziyaret ediyoruz: http://www.python.org/download
2. Bu adreste, üzerinde "*Python 3.x.x compressed source tarball (for Linux, Unix or OS X)*" yazan bağlantıya tıklıyoruz.
3. İlgili .tgz dosyasını bilgisayarımıza indiriyoruz.
4. Daha sonra, bu sıkıştırılmış dosyayı açıyoruz.
5. Açılan dosyanın içine girip, orada sırasıyla aşağıdaki komutları veriyoruz:

```
./configure
make
sudo make altinstall
```

Böylelikle Python 3.x'i sistemimize kurmuş olduk. Bu arada, yukarıdaki komutları vermeden önce tabii ki sistemimizde *gcc* ve *make* adlı araçların kurulu olması gerekiyor...

Yukarıdaki kodlarda "make install" yerine "make altinstall" komutunu kullandığımıza dikkat edin. Python'un kaynak kodlarıyla beraber gelen README dosyasında şöyle bir ibare bulunur:

> **Installing multiple versions**
>
> *On Unix and Mac systems if you intend to install multiple versions of Python using the same installation prefix (–prefix argument to the configure script) you must take care that your primary python executable is not overwritten by the installation of a different version. All files and directories installed using "make altinstall" contain the major and minor version and can thus live side-by-side. "make install" also creates ${prefix}/bin/python which refers to ${prefix}/bin/pythonX.Y. If you intend to install multiple versions using the same prefix you must decide which version (if any) is your "primary" version. Install that version using "make install". Install all other versions using "make altinstall".*
>
> *For example, if you want to install Python 2.5, 2.6 and 3.0 with 2.6 being the primary version, you would execute "make install" in your 2.6 build directory and "make altinstall" in the others.*
>
> **Birden fazla sürümü aynı anda kurmak**
>
> *Unix ve Mac sistemlerinde eğer birden fazla Python sürümünü aynı önek ile (configure betiğine verilen –prefix argümanı) kuracaksanız, farklı bir Python sürümüne ait çalıştırılabilir dosyanın, birincil Python sürümüne ait çalıştırılabilir dosyayı silip üzerine yazmamasına dikkat etmelisiniz. "make altinstall" ile kurulum yapıldığında bütün dosya ve dizinlerde ana ve alt sürüm numaraları da içerilecektir. Dolayısıyla farklı sürümler yan yana varolabilecektir. "make install" komutu, ${önek}/bin/pythonX.Y dosyasına bağlantı veren ${önek}/bin/python adlı bir dosya oluşturacaktır. Eğer aynı öneki kullanarak birden fazla sürüm kurmak istiyorsanız, hangi sürümün (eğer olacaksa) "birincil" sürümünüz olacağına karar vermelisiniz. Birincil sürümünüzü "make install" ile kurun. Öteki bütün sürümleri ise "make altinstall" ile...*
>
> *Örneğin Python 2.5, 2.6 ve 3.0 sürümlerini kuracaksanız ve eğer 2.6 sürümünün birincil sürüm olmasına karar vermişseniz, 2.6 sürümünün inşa dizini içinde "make install" komutunu çalıştırın. Öteki sürümleri ise "make altinstall" ile kurun.*

Bu noktada bir uyarı yapmadan geçmeyelim: Python özellikle bazı GNU/Linux dağıtımlarında pek çok sistem aracıyla sıkı sıkıya bağlantılıdır. Yani Python, kullandığınız dağıtımın belkemiği durumunda olabilir... Bu yüzden Python'u kaynaktan kurmak bazı riskler taşıyabilir. Eğer yukarıda anlatıldığı şekilde, sisteminize kaynaktan Python kuracaksanız, karşı karşıya olduğunuz risklerin farkında olmalısınız...

Kurduğumuz yeni Python'u nasıl çalıştıracağımızı biraz sonra göreceğiz. Ama önce Windows kullanıcılarının Python 3.x'i nasıl kuracaklarına bakalım.

Windows Kullanıcıları

Windows sürümlerinin hiçbirinde Python kurulu olarak gelmez. O yüzden Windows kullanıcıları, Python'u sitesinden indirip kuracak. Bunun için şu adımları takip ediyoruz:

1. http://www.python.org/download adresini ziyaret ediyoruz.

2. Orada, üzerinde *Python 3.x.x Windows installer (Windows binary – does not include source)* yazan bağlantıya tıklıyoruz.
3. .msi uzantılı dosyayı bilgisayarımıza indiriyoruz.
4. İnen dosyaya çift tıklayıp normal bir şekilde kurulumu gerçekleştiriyoruz.
5. Eğer ne yaptığınızdan emin değilseniz, kurulum sırasında varsayılan ayarları değiştirmemenizi öneririm...

Windows'ta Python kurulumu bu kadar basittir. Artık bilgisayarımıza kurduğumuz Python programını nasıl çalıştıracağımızı görebiliriz...

1.2 Python Nasıl Çalıştırılır?

Bir önceki bölümde, Python'u nasıl kuracağımızı farklı platformlara göre anlattık. Bu bölümde ise kurduğumuz bu Python programını hem GNU/Linux'ta hem de Windows'ta nasıl çalıştıracağımızı göreceğiz. Öncelikle GNU/Linux kullanıcılarının Python'u nasıl çalıştıracağına bakalım...

GNU/Linux Kullanıcıları

GNU/Linux kullanıcıları, eğer paket yöneticilerini kullanarak Python kurulumu gerçekleştirmiş iseler, komut satırında şu komutu vererek Python'u başlatabilirler:

```
python3
```

Bu komutun ardından şuna benzer bir ekranla karşılaşmış olmalısınız:

```
Python 3.0.1+ (r301:69556, Feb 24 2009, 13:51:44)
[GCC 4.3.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Gördüğünüz gibi, kullandığımız Python sürümü "3.0.1"...

Burada, paket yöneticisinden kurduğunuz paketin adının "python3" olduğunu varsayıyorum. Düşük bir ihtimal de olsa bu paketin adı sizde farklı olabilir. Örneğin Python3 sizin sisteminizde farklı bir ad altında geçiyor olabilir. Mesela, "python3.0" gibi...

Eğer paketin adı python3.0 ise komut satırında şu komutu vermelisiniz:

```
python3.0
```

Bu komut da benzer bir şekilde şöyle bir çıktı vermeli:

```
Python 3.0.1+ (r301:69556, Feb 24 2009, 13:51:44)
[GCC 4.3.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Python'u ne şekilde başlatacağınızı kesin olarak tespit etmek için */usr/bin* adlı klasörün içinde "python" programının hangi adla yer aldığına bakabilirsiniz. Bu klasörün içinde "python" dışında, adında "python" geçen hangi programların olduğunu kontrol edin. Mesela şunlar olabilir: "python3", "python3.0", "python3.0.1"

Eğer Python3'ü kaynaktan kuranlardansanız, vereceğiniz komut muhtemelen şu olacaktır:

```
python3.0
```

GNU/Linux'ta Python'u nasıl çalıştıracağımızı öğrendiğimize göre, Windows kullanıcılarının durumuna bakabiliriz...

Windows Kullanıcıları

Gelelim Windows kullanıcılarına...

Windows kullanıcıları Python3'ü iki şekilde başlatabilir:

1. Başlat > Programlar > Python3.x > Python (Command Line)* yolunu takip ederek.
2. Python3.x'i YOL'a (PATH) ekledikten sonra DOS ekranında "python" komutunu vererek...

Eğer birinci yolu tercih ederseniz, Python'un size sunduğu komut satırına ulaşırsınız. Ancak Python komut satırına bu şekilde ulaştığınızda bazı kısıtlamalarla karşı karşıya kalırsınız. O yüzden komut satırına bu şekilde ulaşmak yerine ikinci seçeneği tercih edebilirsiniz.

Eğer sisteminizde birkaç farklı Python sürümü kurulu ise ve eğer siz bu eski sürümlerden birini YOL'a eklemişseniz, Python3.0'ı YOL'a eklemeyin. Sisteminde sadece Python3.0 kurulu olan Windows kullanıcıları Python'u YOL'a nasıl ekleyeceklerini öğrenmek için http://www.istihza.com/py2/windows-path.html adresindeki makaleyi inceleyebilirler. (Orada Python26 yerine Python30 ifadesi gelecek...)

Sisteminizde sadece Python3.0 sürümünün kurulu olduğunu ve bu sürümü başarıyla YOL'a eklediğinizi varsayarak bir deneme yapalım. *Başlat > Çalıştır* yolunu takip ederek, açılan pencerede "cmd" komutunu verelim ve Windows komut satırına ulaşalım. Orada şu komutu verelim:

```
python
```

Eğer sisteminizde başka bir Python sürümü kurulu değilse veya siz kurulu olan sürümü daha önceden YOL'a eklememişseniz, yukarıdaki komutu verdiğinizde Python3.0 çalışmaya başlayacaktır. Şuna benzer bir çıktı almalısınız:

```
Python 3.0.1 (r301:69561, Feb 13 2009, 20:04:18) [MSC v.1500 32 bit (Intel)] on
win32 Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Eğer sisteminizde eski bir Python sürümü kuruluysa ve siz bu sürümü daha önceden YOL'a eklemişseniz, "python" komutunu verdiğinizde Python3 yerine, sistemimizdeki eski Python sürümü açılacaktır. Çünkü Windows'taki bütün Python sürümlerinin çalıştırılabilir (exe) dosyaları "python.exe" şeklinde adlandırıldığı için, DOS ekranında yazacağınız "python" komutu eski sürümü çalıştıracaktır. Dediğim gibi, eğer eski sürümü önceden YOL'a eklemişseniz, yeni kurduğunuz Python3'ü YOL'a eklemeyin. Bunun yerine, basit bir .bat dosyası yazmamız yeterli olacaktır. Nasıl mı?

Notepad yardımıyla boş bir metin belgesi açın ve bu metin belgesinin içine şu satırı ekleyin:

```
@c:\python30\python.exe %*
```

Şimdi bu dosyayı "python3.bat" adıyla kaydedin ve daha önce YOL'a eklediğiniz eski sürüm Python'un bulunduğu klasörün içine atın (mesela C:python26).

Artık "cmd" ile ulaştığınız komut satırında sadece "python3" yazıp enter'e basarak Python3'ü çalıştırabilirsiniz. Sadece "python" komutu verdiğinizde ise sisteminizdeki eski sürüm Python açılacaktır. Tabii ki bu eski sürümü daha önceden YOL'a eklemiş iseniz...

Bu arada, oluşturduğunuz .bat dosyasını Python26 klasörünün içine atmak yerine, Python'a ilişkin .bat dosyalarını özel bir klasörde toplayıp bu klasörü YOL'a ekleyerek de işinizi halledebilirsiniz. Örneğin benim Windows yüklü bilgisayarımda Python'un 2.5, 2.6 ve 3.0 sürümleri birlikte kurulu... Ben "yol" adını verdiğim bir klasör oluşturup bu klasörü "C:" dizininin içine

attım. Daha sonra “C:\yol” dizinini YOL’a ekledim. Ardından Python2.5 ve Python3.0 sürümleri için iki ayrı .bat dosyası oluşturdum. Bunlardan birini “python25.bat”, ötekini de “python3.bat” olarak adlandırdım. “python25.bat” dosyasının içeriği şöyle:

```
@c:\python25\python.exe %*
```

“python3.bat” dosyasının içeriği ise şöyle:

```
@c:\python30\python.exe %*
```

Bu şekilde, “cmd” ile ulaştığım komut satırında “python3” komutunu verdiğimde “Python3.0” sürümünün ekranı açılıyor:

```
Python 3.0.1 (r301:69561, Feb 13 2009, 20:04:18) [MSC v.1500 32 bit (Intel)] on
win32 Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Eğer “python25” komutunu verirsem, “Python2.5” sürümüne ait komut ekranı açılıyor:

```
Python 2.5.4 (r254:67916, Dec 23 2008, 15:10:54) [MSC v.1310 32 bit (Intel)] on
win32 Type "help", "copyright", "credits" or "license" for more information.
>>>
```

“Python2.6” sürümüne ise sadece “python” yazarak ulaşabiliyorum:

```
Python 2.6.1 (r261:67517, Dec  4 2008, 16:51:00) [MSC v.1500 32 bit (Intel)] on
win32 Type "help", "copyright", "credits" or "license" for more information.
>>>
```

İsterseniz siz de böyle bir çalışma şekli benimseyebilirsiniz...

1.3 FreeBSD’de Python

Yukarıda Python 3.x’in GNU/Linux ve Windows sistemlerine nasıl kurulacağını ve nasıl çalıştırılacağını anlattık. Python’un FreeBSD sistemine nasıl kurulacağını öğrenmek isterseniz http://www.istihza.com/denemeler/freebsd.7z adlı dosyayı indirerek, bu sıkıştırılmış klasör içindeki video dosyalarını izleyebilirsiniz. Klasör içinde ayrıca FreeBSD kurulumunu gösteren bir video da bulunuyor. FreeBSD üzerinde Python’un nasıl kurulup kullanılacağına ilişkin videoları hazırlayan **Mahmut Çetin**‘e teşekkür ederim.

Videoları 7z adlı yazılımla sıkıştırdım. Videoların özgün boyutu toplam 223 MB’dir. İndireceğiniz sıkıştırılmış klasörün boyutu 7z yardımıyla 2 MB’ye kadar azaltılmıştır.

Ubuntu GNU/Linux kullanıcıları bu dosyayı açabilmek için şu komut yardımıyla 7z yazılımını sistemlerine kurabilir:

```
sudo apt-get install p7zip-full
```

Yazılımı kurduktan sonra sıkıştırılmış dosyaya sağ tıklayıp “buraya aç” seçeneği yardımıyla dosyayı açabilirsiniz.

Windows kullanıcıları ise gerekli yazılımı http://www.7-zip.org/ adresinden indirebilir.

1.4 Python'un Etkileşimli Kabuğu ve print() Fonksiyonu

Bir önceki bölümde Python'un farklı sistem ve durumlarda nasıl çalıştırılacağını görmüş, Python'u anlattığımız şekilde çalıştırdığınız zaman karşınıza şuna benzer bir ekranın geleceğini söylemiştik:

```
Python 3.0.1+ (r301:69556, Feb 24 2009, 13:51:44)
[GCC 4.3.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Bu ekrana Python dilinde "etkileşimli kabuk" (interactive shell) adı verilir. Bu, bizim Python programlama dili ile ilişki kurabileceğimiz, yani onunla etkileşebileceğimiz bir üst katmandır. Etkileşimli kabuk, asıl programımız içinde kullanacağımız kodları deneme imkanı sunar bize... Burası bir nevi "test alanı" gibidir. Örneğin bir Python kodunun çalışıp çalışmadığını denemek veya nasıl çalıştığını, ne sonuç verdiğini görmek istediğimizde bu ekran son derece faydalı bir araç olarak karşımıza çıkar. İsterseniz konuyu daha fazla lafa boğmayalım. Zira etkileşimli kabuğu kullandıkça bunun ne büyük bir nimet olduğunu siz de anlayacaksınız. Özellikle derlenerek çalıştırılan programlama dilleri ile uğraşmış olan arkadaşlarım, etkileşimli kabuğun gücünü gördüklerinde, göz yaşlarına hakim olamayacaklar...

Şimdi kendi sistemimize uygun bir şekilde etkileşimli kabuğu çalıştırıyoruz. Bu ekrana nasıl ulaşacağımızı bir önceki konuda ayrıntılı olarak anlatmıştık. Etkileşimli kabuğa ulaşmak konusunda sıkıntı yaşıyorsanız bir önceki konuyu tekrar gözden geçirmenizi tavsiye ederim.

Etkileşimli kabuğu çalıştırdığımızda ekranda görünen ">>>" işareti Python'un bizden komut almaya hazır olduğunu gösteriyor. Python kodlarımızı bu ">>>" işaretinden hemen sonra, hiç boşluk bırakmadan yazacağız. İsterseniz basit bir deneme yapalım. ">>>" işaretinden hemen sonra, hiç boşluk bırakmadan şu komutu yazıyoruz:

```
print("Merhaba Zalim Dünya!")
```

Bu komutu yazıp "enter" tuşuna bastığımızda, eğer herhangi bir yazım hatası yapmamışsak, şöyle bir çıktı almış olmalıyız:

```
Merhaba Zalim Dünya!
```

Gördüğünüz gibi, `print()` adlı fonksiyon, ekrana çıktı vermemizi, yani yazı yazdırmamızı sağlıyor... Bu arada, bu "fonksiyon" kelimesine fazla takılmayın. İlerde bunun ne demek olduğunu ayrıntılı bir şekilde inceleyeceğiz. Şimdilik `print()` ifadesinin bir "fonksiyon" olduğunu bilmemiz yeterli olacaktır.

Burada dikkat etmemiz gereken bazı noktalar var:

Komutumuzu ">>>" işaretinden hemen sonra veriyoruz. Python'a yeni başlayanların en sık yaptığı hatalardan bir tanesi, `print()`'in "p"si ile ">>>" işareti arasında bir boşluk bırakmalarıdır. Eğer "p" harfi ile ">>>" işareti arasında boşluk bırakırsak, Python bize bir hata mesajı gösterecektir. O yüzden boşluklara dikkat ediyoruz...

`print()` fonksiyonunu oluşturan bütün harflerin küçük olduğuna dikkat edin. Python açısından, "print" ve "Print" kelimeleri birbirinden farklıdır. Bizim fonksiyonumuzun adı `print()`'tir. Başka bir şey değil...

`print()` fonksiyonunda, parantez içindeki ifadenin tırnak içinde gösterildiğine özellikle dikkat ediyoruz. Burada çift veya tek tırnak kullanmamız önemli değildir. Yani yukarıdaki kodu şöyle de yazabiliriz:

```
print('Merhaba Zalim Dünya!')
```

Yalnız dikkat etmemiz gereken şey, hangi tırnakla başladıysak onunla bitirmemiz gerektiğidir. Yani eğer başta çift tırnak kullandıysak, sonda da çift tırnak kullanmalıyız. Aynı şey tek tırnak için de geçerlidir. Yani Python açısından tek veya çift tırnak kullanmamızın değil, tutarlı olmamızın önemi vardır. Peki neden iki farklı tırnak çeşidi var?

Ekrana şöyle bir çıktı vermek istediğimizi varsayın:

> *Python programlama dilinin adı "piton" yılanından gelmez...*

Yukarıdaki çıktıyı alabilmek amacıyla etkileşimli kabukta ">>>" işaretinden hemen sonra aşağıdaki komutu verip enter tuşuna basın. Bakalım ne olacak?

```
print("Python programlama dilinin adı "piton" yılanından gelmez...")
```

Ne oldu? Bu komut bize sinir bozucu bir hata mesajı verdi, değil mi?

```
File "<stdin>", line 1
print("Python programlama dilinin adı "piton" yılanından gelmez...")
                                        ^
SyntaxError: invalid syntax
```

Dikkat ederseniz, yukarıdaki hata mesajında "piton" kelimesinin hemen altında minik bir ok işareti görünüyor. Bu ok işareti, ortaya çıkan hatanın nerede olduğu konusunda bize ipucu veriyor. Buradaki hata, tırnak işaretlerinin kullanımından kaynaklanıyor. Biz cümlemize çift tırnak ile başladık. Ama cümle içinde "piton" kelimesinde de birer adet çift tırnak kullandık. Ayrıca cümlemizi de çift tırnak ile bitirdik. İşte bu noktada Python'un kafasının karışmasına neden olduk. Python böyle bir yapı ile karşılaştığında, "piton" kelimesinin başındaki çift tırnak nedeniyle cümlenin nerede başlayıp nerede bittiğini anlayamıyor. İlk çift tırnaktan sonra gördüğü ikinci çift tırnağın cümle sonu olduğunu zannediyor. Ama "piton" kelimesinin sonunda da bir çift tırnak olduğunu görünce bir şeylerin ters gittiğini düşünüyor ve bize bir hata mesajı gösteriyor. Zaten gördüğünüz gibi, ok işareti de bu üçüncü çift tırnağın olduğu yerde duruyor. Yani Python'un kafası tam olarak o noktada karışmış... Bize düşen görev, Python'un kafa karışıklığını gidermek. Bunu şöyle yapabiliriz:

```
>>> print('Python programlama dilinin adı "piton" yılanından gelmez...')

Python programlama dilinin adı "piton" yılanından gelmez...
```

Gördüğünüz gibi, bu komutta, hatayı önlemek için cümlemize tek tırnak ile başladık ve cümlemizi tek tırnak ile bitirdik. Cümle içindeki "piton" kelimesini ise çift tırnak ile göstererek karışıklığı önledik. Böylece bu komutla istediğimiz çıktıyı almış olduk...

Demek ki, Python'da ekrana çıktı verirken, tırnak işaretlerini dikkatli kullanıyoruz.

Bu arada yukarıdaki kodlar içinde görünen ">>>" işaretini siz yazmayacaksınız. Bu işareti etkileşimli kabuğun görünümünü temsil etmek için yerleştirdik oraya... Siz ilk satırı yazdıktan sonra doğruca enter tuşuna basacaksınız.

Şimdi şu cümleye bakalım:

> *Python'u yazan kişi Guido Van Rossum adlı bir programcıdır...*

Acaba bu cümleyi, `print()` fonksiyonunu kullanarak ekrana nasıl yazdırmalıyız? Evet, tam tahmin ettiğiniz gibi:

```
>>> print("Python'u yazan kişi Guido Van Rossum adlı bir programcıdır...")

Python'u yazan kişi Guido Van Rossum adlı bir programcıdır...
```

“Python’u” kelimesinde geçen ayraç işareti nedeniyle cümlemize tek tırnakla başlamıyoruz. Eğer cümleye tek tırnakla başlarsak, Python o tek tırnak işareti yüzünden, cümlenin nerede başlayıp nerede bittiğini anlayamayacak ve bize bir hata çıktısı verecektir. İşte biz de bunu önlemek için yukarıdaki gibi bir önlem alıyoruz...

Gördüğünüz gibi, Python’un hem çift tırnağa hem de tek tırnağa müsaade etmesi bize bazı durumlarda büyük bir kolaylık sağlıyor. Demek ki birden fazla tırnak kullanılması boşuna değil...

Eğer ekrana herhangi bir şey yazdırmak yerine, bir satır boşluk bırakmak isterseniz, `print()` fonksiyonunu boş olarak kullanabilirsiniz. Yani şöyle:

```
>>> print()
```

Bu komutu verip enter’e bastığımızda etkileşimli kabuğun bir boşluk bırakıp alt satıra geçtiğini görüyoruz...

Etkileşimli kabukta çalışırken, `print()` fonksiyonunu kullanmasak da yazdığımız cümleler ekrana çıktı olarak verilecektir. Yani şöyle bir kullanım etkileşimli kabukta çalışırken mümkündür:

```
>>> "Lütfen kullanıcı adı ve parolanızı giriniz"

'Lütfen kullanıcı adı ve parolanızı giriniz'
```

Gördüğünüz gibi, `print()` fonksiyonunu kullanmasak da ekrana çıktı alabiliyoruz. Ama bu durum sizi yanıltmasın. Bu özellik sadece etkileşimli kabuğa mahsustur. İlerde kodlarımızı dosyalara yazıp kaydettiğimiz zaman, ekrana çıktı verebilmek için mutlaka `print()` fonksiyonunu kullanmamız gerekir. Eğer programlarımız içindeki cümleleri `print()` fonksiyonu ile kullanmazsak, programı çalıştırdığımızda, yazdığımız cümleyi Python görür, ama biz ve kullanıcılarımız göremeyiz!

Bu arada `print()` fonksiyonu ile veya onsuz yazdığımız cümlelerdeki Türkçe karakterlerin çıktıda herhangi bir soruna sebep olmadığına dikkat edin. `print()` fonksiyonu olmadan yazılan cümlelerdeki Türkçe karakterlerin düzgün görünmesi, Python 3.x ile gelen bir özelliktir. Eğer yukarıdaki cümleyi Python’un 2.x sürümlerinden birinde verseydik, şöyle bir tabloyla karşılaşacaktık:

```
>>> "Lütfen kullanıcı adı ve parolanızı giriniz"

'L\xc3\xbctfen kullan\xc4\xb1c\xc4\xb1 ad\xc4\xb1 ve parolan\xc4\xb1z\xc4\xb1 giriniz'
```

Bunun dışında, bu bölümde öğrendiğimiz `print()` fonksiyonu da Python 3.x’le birlikte mutasyon geçiren özelliklerden biridir. Bu yazıda öğrendiğimiz:

```
print("Merhaba Zalim Dünya!")
```

komutunu Python’un 3.x öncesi sürümlerinde:

```
print "Merhaba Zalim Dünya"
```

şeklinde yazıyorduk...

Böylece Python’da `print()` fonksiyonunun ne olduğunu ve ne işe yaradığını öğrenmiş olduk. İlerde kodlarımızı dosyalara kaydettiğimiz zaman bu fonksiyonu bol bol kullanacağız. Bu konuyu bitirdiğimize göre, artık yeni bir konuya geçebiliriz.

1.5 Python'da Basit Matematik İşlemleri

Bir önceki bölümde nasıl başlatacağımızı ve nasıl kullanacağımızı öğrendiğimiz etkileşimli kabuk üzerinde biraz daha çalışmaya devam edeceğiz. Etkileşimli kabuk aslında Python'da asıl çalışma ortamımız değildir. Kodlarımızı esas olarak dosyalara kaydedeceğiz. Ama etkileşimli kabuk bize, asıl programlarımızı yazmaya başlamadan önce Python'a aşinalık kazanma fırsatı verecek... Etkileşimli kabuk; kod alıştırması yapma ve kodları test etme gibi işlerimiz için harika bir araçtır.

Bu bölümde, Python'da basit matematik işlemlerini nasıl yapabileceğimizi öğreneceğiz. Python'la hiç bir şey yapamasak bile, onu basit bir hesap makinesi yerine kullanabiliriz. Şimdi etkileşimli kabuğu açıp çalışmaya başlayalım:

```
>>> 5 + 2

7
```

Gördüğünüz gibi, Python matematikten anlıyor.. Üstelik yukarıdaki kodda bize yabancı gelecek hiçbir öğe yok. Tıpkı bildiğimiz matematikte olduğu gibi, "+" işareti "toplama" anlamına geliyor. Bir de şuna bakalım:

```
>>> 3457 - 2456

1001
```

Burada da bir sıkıntı yok. Her şey sıradan... Peki çarpma ve bölme işlemleri için ne kullanacağız? Onlara da bakalım:

Çarpma işlemi için "*" işaretini kullanıyoruz:

```
>>> 6 * 5

30
```

Bölme işlemi için ise "/" işaretini:

```
>>> 5 / 2

2.5
```

Python için, işleme alınan sayıların büyüklüğü küçüklüğü önemli değildir. Python çok büyük (veya çok küçük) sayıları hiçbir sorun çıkarmadan çarpabilir, toplayabilir, çıkarabilir veya bölebilir...

Yalnız, birden fazla matematik işlemini aynı anda yaparken bir konuya dikkat etmemiz gerekir. Mesela size şöyle bir soru soralım: Sizce aşağıdaki işlemin sonucu kaçtır?:

```
>>> 5 * 2 + 4 / 2
```

Eğer yukarıdaki işlemin sonucu beklediğiniz gibi çıkmadıysa, "işlem önceliği" (operator precedence) denen kavramı gözardı etmişsiniz demektir. Matematik derslerinden hatırladığımız "işlem önceliği" kuralı Python'da da geçerlidir. Bu kurala göre; çarpma ve bölme işlemleri, toplama ve çıkarma işlemlerinden önce yapılır. Yani yukarıdaki ifadede önce "5 * 2" işlemi, ardından "4 / 2" işlemi yapılacak, daha sonra bu iki işlemin sonucu birbiriyle toplanacaktır. Python'un kullandığı işlem sırasını değiştirmek için parantez işaretlerinden yararlanabilirsiniz. Örneğin yukarıdaki işlemin "7" sonucunu vermesi için ifadeyi şöyle düzenlemeniz gerekir:

```
>>> (5 * 2 + 4) / 2

7.0
```

Öncelikli olarak yapılmasını istediğimiz işlemleri parantez içine aldığımıza dikkat edin. Bu şekilde, önce "5" ile "2" sayısı çarpılacak, ardından bu işlemin sonucuna "4" eklenecek ve çıkan değer "2"ye bölünecektir.

Matematik işlemleri yaparken, bu şekilde parantezler kullanarak işlemin istediğiniz gibi sonuç vermesini garanti edebilirsiniz.

Şimdiye kadar Python'da "+", "-", "*" ve "/" işaretlerini gördük. Bu işaretlere "işleç" (operator) adı verilir. Gelin isterseniz Python'da daha başka hangi faydalı işleçlerin olduğuna bir göz gezdirelim:

"%" işleci

Bir bölme işleminde, kalan sayıyı bulmak için "%" işaretinden yararlanıyoruz:

```
>>> 5 % 2

1
```

Demek ki "5" sayısını "2"ye böldüğümüzde, bölme işleminden artan sayı, yani "kalan", 1 oluyormuş...

Bir de şu işlemin sonucuna bakalım:

```
>>> 10 % 2

0
```

Kalan "0" olduğuna göre, demek ki "10" sayısı "2"ye tam bölünüyormuş... Peki bu bilgi bizim ne işimize yarar? Mesela "%" adlı işlecin verdiği sonuca bakarak, sayıların çift mi yoksa tek mi olduğunu denetleyebiliriz. "*herhangibirsayı % 2*" işleminin sonucu "0" ise o sayı çifttir. Eğer sonuç "1" ise o sayı tektir... Eğer ilkokul öğrencilerine basit matematik kavramlarını öğreten bir program yazmayı planlıyorsanız bu işleç işinize yarayacaktır. Hatta ileride bu işlecin hiç tahmin etmediğiniz yerlerde de işinize yarayacağını görürseniz şaşırmayın...

"//" işleci

Python'da kullanabileceğimiz başka bir işleç ise şudur: "//". Bu işleç, bir bölme işleminde sonucun sadece tamsayı kısmını almamızı sağlar. Hemen bir örnek vererek durumu somutlaştıralım:

```
>>> 9 // 2

4
```

Gördüğünüz gibi, sonuç tamsayı şeklinde. Yani sonucumuz ondalık kısmı içermiyor. Normalde "9" sayısı "2" sayısına bölündüğünde şu sonucu elde ederiz:

```
>>> 9 / 2

4.5
```

Aynı işlemi "9 // 2" şeklinde yaptığımızda ise ondalık kısım atılır, ekrana sadece tamsayı kısım verilir... İlk bakışta bu işleç çok manalı gelmeyebilir, ama ileride mutlaka, ondalık sayı yerine tamsayı elde etmek istediğiniz durumlarla karşılaşacaksınızdır.

"" işleci**

Bu işleç, bir sayının kuvvetlerini hesaplamak için kullanılır. Üslü sayıları bulmak için bu işleçten yararlanacağız. Örneğin:

```
>>> 2 ** 3

8
```

Demek ki "2" sayısının üçüncü kuvveti "8" imiş... Mesela 1453 sayısının karesini şöyle bulabiliriz:

```
>>> 1453 ** 2

2111209
```

Ya da 15 sayısının 3. kuvvetini şöyle bulabiliriz:

```
>>> 15 ** 3

3375
```

Python'da bunların dışında daha pek çok işleç bulunur. Ama içlerinde şu anda en çok işimize yarayacak, en temel işleçler bunlardır. İlerde öteki işleçleri de inceleyeceğiz.

Şu ana kadar Python'un temellerine ilişkin pek çok şey söyledik. Örneğin en temel öğelerden biri olan `print()` fonksiyonunu ve bu fonksiyonun ne işe yaradığını öğrendik. Bunun yanısıra Python'da sayıların kullanımına da şöyle bir göz gezdirdik. `print()` fonksiyonunu işlerken, bu fonksiyonun ekrana çıktı vermek için kullanıldığını, bu fonksiyonla birlikte kullandığımız cümleleri tırnak içine almamız gerektiğini söylemiştik. Yalnız bu bölümde dikkatinizi çekti mi bilmiyorum, ama farkettiyseniz yukarıdaki kodların hiçbirinde tırnak işareti kullanmadık. Şimdi gelin isterseniz bunun nedenlerini tartışalım...

1.6 Karakter Dizileri (strings)

Python'da çok önemli iki adet öğe vardır. Bunlardan biri "karakter dizileri" (strings), ikincisi ise "sayılar"dır (numbers). "Sayı"nın ne olduğu adından belli. Peki bu "karakter dizisi" denen şey de ne oluyor? Aslında `print()` fonksiyonunu anlatırken karakter dizilerini kullandık. Yalnız orada kafa karıştırmamak için "karakter dizisi" yerine "cümle" deyip geçmiştik. Şimdi ise sizlere hakikati söylemenin zamanı geldi dostlar! Evet, daha önce "cümle" dediğimiz şey esasında Python'cada "karakter dizisi"dir. İngilizce konuşanlar buna "string" diyor... Peki karakter dizisi denen şeyi gördüğümüzde nasıl tanıyacağız? Tabii ki tipine bakarak... Python'da karakter dizileri tırnak içinde gösterilir. Yani şu aşağıda gördüğümüz şey bir karakter dizisidir:

```
"Merhaba Zalim Dünya!"
```

Bu karakter dizisini ekrana yazdırmak için, bildiğiniz gibi şu komutu kullanıyoruz:

```
print("Merhaba Zalim Dünya!")
```

Yani "karakter dizisi"; içinde bir veya daha fazla sayıda karakter barındıran bir dizidir. Dolayısıyla bir "şey"in karakter dizisi olabilmesi için birden fazla karakter içeriyor olması şart değildir. Python'da tek bir karakter dahi, "karakter dizisi" sınıfına girer. Bu sebeple aşağıdaki "şey" de bir karakter dizisidir:

```
"c"
```

Hatta bir şeyin karakter dizisi olabilmesi için harf olması da gerekmez. Kabaca söylemek gerekirse, tırnak içinde gösterebileceğimiz her şey bir karakter dizisir. Boşluk karakteri de dahil...

Karakter dizilerini anladık sayılır. "Sayılar"ı ise zaten adından ötürü rahatlıkla anlayabiliyoruz. Mesela şu örnek bir sayıdır:

```
12354
```

Ama dikkat edin! Şu örnek bir sayı değildir:

```
"12354"
```

Tırnak içinde gösterilen "12354" bir karakter dizisidir. Dediğimiz gibi, Python'da tırnak içinde gösterilen her şey bir karakter dizisidir. Karakter dizilerini basitçe böyle ayırt edebiliriz. Bu durumu şu örneklerle teyit edelim:

```
>>> 12354 + 3444

15798
```

İsterseniz bunu bir de `print()` fonksiyonu ile gösterelim. Zaten normalde hep bu şekli kullanmamız yararlı olacaktır:

```
>>> print(12354 + 3444)

15798
```

Bir de şuna bakalım:

```
>>> print("12354 + 3444")

12354 + 3444
```

Gördüğünüz gibi, bu defa çıktımız farklı oldu. Neden? Çünkü dediğimiz gibi, tırnak işareti olmayan 12354 veya 3444 birer sayıdır, ama tırnak işareti olan "12354" veya "3444" birer karakter dizisidir. Aritmetik işlemleri sayılarla yapılır, karakter dizileriyle değil... Dolayısıyla Python, "*Merhaba Zalim Dünya*"ya nasıl davranıyorsa, "*12354 + 3444*"e de aynı şekilde davranıyor. Bunun için Python'u suçlayamayız...

Hatta şöyle ilginç bir örnek de verebiliriz:

```
>>> print("12354" + "3444")

123543444
```

Gördüğünüz gibi, iki tane karakter dizisiyle karşılaşan Python ("12354" ve "3444"), "artı" (+) işaretini bu iki karakter dizisini birleştirmek için kullandı. Eğer artı işaretinin beraber kullanıldığı öğeler birer sayı olsaydı, Python bunları yan yana yazmak yerine, birbirleriyle toplayacaktı.

Yukarıdaki örneklerin bize gösterdiği gibi, bazı işleçler, birlikte kullanıldıkları öğelerin tipine göre farklı anlamlar taşıyabilir. Yukarıda da şahit olduğumuz gibi, eğer söz konusu olan şey sayılar ise, "artı" işareti bir aritmetik işlem yapılmasını sağlayacaktır. Ama eğer söz konusu olan şey karakter dizileri ise, "artı" işareti bu karakter dizilerini bir araya getirme görevi görecektir. Mesela daha önce çarpma işlemlerinde kullandığımız "*" işareti de birlikte kullanıldığı öğelerin tipine göre farklı anlamlar taşıyabilir:

```
>>> print("yavaş"*2)

yavaşyavaş
```

Python burada “yavaş” karakter dizisini iki kez tekrar etti. Ama dikkat ederseniz, “yavaş” adlı karakter dizisini tekrar ederken araya boşluk koymadı. Tabii ki Python bizim ne istediğimizi bilemez. Python’un istediğimizi yapabilmesi için bizim ona yardımcı olmamız gerekir. Yukarıdaki kodları şöyle yazarsak, çıktı daha düzgün görünecektir:

```
>>> print("yavaş "*2)

yavaş yavaş
```

“yavaş” adlı karakter dizisinin kapanış tırnağını koymadan önce bir boşluk bırakarak emelimize ulaştık... Python çıktıda o boşluk karakterini de göreceği için, iki tane “yavaş” kelimesi çıktıda boşluklu olarak görünecektir.

Yalnız bazı durumlarda karakter dizisinin sonuna böyle boşluk eklemek mümkün olmayabilir. Eğer öyle bir durumla karşılaşırsak, şöyle bir şey de yapabiliriz:

```
>>> print(("yavaş" + " ")*2)

yavaş yavaş
```

Gördüğünüz gibi, burada sanki bir matematik işlemi yapar gibi, parantezleri kullanarak Python’a yol gösterdik. Matematikteki işlem önceliği kuralının burada nasıl işlediğine dikkat edin.

Karakter dizileri (strings) tabii ki yukarıda anlattıklarımızla sınırlı değildir. Ama şimdilik bizim bilgimiz sınırlı olduğu için, Python’daki karakter dizilerinin bütün imkanlarını burada önünüze seremiyoruz. Python’la ilgili birkaç şey daha öğrendikten sonra karakter dizilerini etkili bir şekilde kullanmayı da öğreneceğiz. Zira karakter dizileri, Python’un en güçlü olduğu alanlardan birisidir.

Bu konuyu da böylece tamamlamış olduk. Henüz söylenmesi gereken her şeyi söyleyemedik, ama Python’da sağlam bir temel atmamımızı sağlayacak pek çok önemli bilgiyi konular arasına serpiştirdik. Bu ilk bölümleri sindire sindire çalışmak, ilerde kemikli konuları daha kolay öğütmemizi sağlayacaktır.

Bir sonraki bölümde, Python’da hareket kabiliyetimizi bir hayli artıracak bir konuyu inceleyeceğiz: Değişkenler

1.7 Değişkenler

Bir önceki bölümün sonunda da belirttiğimiz gibi, değişkenler Python’daki esnekliğimizi, hareket kabiliyetimizi bir hayli artıracak olması bakımından epey önemli bir konudur. Bu konuyu işledikten sonra, artık asıl çalışma alanımız olan dosyalara geçebilecek kadar bilgi sahibi olmuş olacağız. Yani bu konuyu da atlattıktan sonra gerçek anlamda ilk programlarımızı yazmaya başlayabileceğiz.

Biz burada “değişken” kavramını tanımlamaya uğraşmakla vakit kaybetmeyeceğiz. Bir kısmımız bu kavrama zaten pek de yabancı değiliz. Öbür kısmımız ise verdiğimiz ilk örnekte bunun ne olduğunu, ne işe yaradığını derhal anlayacaktır... Dolayısıyla bir an önce örneklerimize geçelim:

Mesela şu örneğe bir bakalım:

```
>>> n = 10
```

Burada “n” adlı bir ifadeye, “10” değerini atadık. Yani, değeri 10 olan “n” adlı bir değişken tanımladık. Artık “n” değişkeninin değerine şu şekilde ulaşabiliriz:

```
>>> print(n)

10
```

Gördüğünüz gibi, 10 değerini elde etmek için “n” değişkenini ekrana yazdırmamız yeterli oluyor.

Bu şekilde bir değişken tanımladıktan sonra, bu değişkeni türlü şekillerde kullanabiliriz. Örneğin bu değişkenle matematik işlemleri yapabiliriz:

```
>>> print(n * 5)

50
```

Tabii ki yukarıdaki değişkenle matematik işlemleri yapabilmemiz, bu değişkenin değerinin bir sayı olmasından kaynaklanıyor. Eğer “n” değişkeninin değeri bir karakter dizisi olsaydı yukarıdaki komuttan alacağımız çıktı çok farklı olacaktı. Bakalım:

```
>>> n = "10"
```

Böylece “n” değerini yeniden tanımlamış olduk. Artık “n” değişkeninin değeri 10 değil, “10”... Yani sayı değil, karakter dizisi. Bunu şu şekilde teyit edebiliriz:

```
>>> print(n * 5)

1010101010101010101010
```

Gördüğünüz gibi, bu defa Python 5 adet 10’u yan yana dizdi. Bunun sebebini biliyorsunuz. Aritmetik işlemleri sayılarla yapılır, karakter dizileriyle değil...

Gelin isterseniz birkaç tane daha değişken tanımlayarak elimizi alıştıralım:

```
>>> isim = "istihza"
```

Burada, değeri “istihza” olan, isim adlı bir değişken tanımladık. Buna şöyle ulaşabiliriz:

```
>>> print(isim)

istihza
```

Bu değişkeni istersek başka karakter dizileriyle birlikte kullanarak daha karışık işlemler de yapabiliriz. Mesela:

```
>>> print("Benim adım", isim)

Benim adım istihza
```

Burada, “Benim adım” adlı karakter dizisinden sonra bir virgül koyduğumuza ve “isim” adlı değişkeni ne şekilde kullandığımıza dikkat edin. Bununla ilgili benzer bir örnek daha yapalım:

```
>>> konu = "değişkenler"

>>> print("Bu dersimizin konusu", konu)

Bu dersimizin konusu değişkenler
```

Elbette, değişkenimizi sadece en sonda kullanmak zorunda değiliz. Bunu aralara da yerleştirebiliriz. Örneğin şöyle bir kod parçası yazdığımızı düşünün:

```
tarih = "12 Ekim 2007"

ziyaret_sayısı = "123456789"

print("Bu siteye", tarih, "tarihinden bu yana", ziyaret_sayısı, "defa tıklanmıştır.")
```

Bu kodların çıktısı şöyle olacaktır:

```
Bu siteye 12 Ekim 2007 tarihinden bu yana 123456789 defa tıklanmıştır.
```

Burada değişkenlerle ilgili olarak dikkatimizi çeken bazı noktalar var:

Değişken adı belirlerken Türkçe karakter kullanabiliyoruz. Gördüğünüz gibi, "ziyaret_sayisi" demek yerine, "ziyaret_sayısı" diyebildik... Bu özellik Python3.0 ile gelen bir güzelliktir. Python'un 2.x sürümlerinde aynı değişkeni "ziyaret_sayisi" şeklinde tanımlamamız gerekecekti.

Eğer değişken adı olarak birden fazla kelime kullanacaksak, kelimeler arasında boşluk bırakmıyoruz. Kelimeleri bitişik olarak yazabileceğimiz gibi, burada gördüğümüz şekilde kelimeler arasına alt çizgi işareti de koyabiliriz.

Ayrıca yine virgülleri nasıl kullandığımıza özellikle dikkat ediyoruz...

Bunların dışında değişken adları ile ilgili birkaç kural daha vardır...

Değişken adları asla bir sayıyla başlamaz. Kelimenin ortasında veya sonunda sayı bulunabilir, ama başında bulunamaz... Mesela "3sayı" geçerli bir değişken adı değildir. Ama "s3ayı" veya "sayı3" geçerli birer değişken adıdır.

Ayrıca, bazı özel kelimeler vardır ki, bunları değişken adı olarak kullanamayız. Bunlar şöyle listelenebilir:

> `and`, `del`, `from`, `not`, `while`, `as`, `elif`, `global`, `or`, `with`, `assert`, `else`, `if`, `pass`, `yield`, `break`, `except`, `import`, `print`, `class`, `exec`, `in`, `raise`, `continue`, `finally`, `is`, `return`, `def`, `for`, `lambda`, `try`

Elbette bu listeyi ezberlemenize gerek yok. Programınız değişkenlerle ilgili anlamsız hatalar verdiğinde gelip bu listeyi kontrol edebilirsiniz... Ayrıca etkileşimli kabukta şu komutu vererek de yukarıdaki listeyi elde edebilirsiniz:

```
>>> help("keywords")
```

Gerçi eğer programlarınızı Türkçe olarak yazıyorsanız, yukarıdaki yasaklı kelimelerden birine toslamanız düşük bir ihtimaldir. Ama yine de Python'da bu tür "yasaklı kelimeler"in olduğunu bilmenin ilerde bize faydası dokunacaktır.

Değişkenler özellikle kullanıcıyla etkileşen, yani onlardan veri alıp onlara veri veren programlar yazdığımızda daha çok işimize yarayacaktır.

Gördüğünüz gibi, komut satırında yazdığımız kodlar artık tek satırı geçmeye başladı. Bunları bir yere kaydedip oradan çalıştırsak daha iyi olacak. Hem zaten etkileşimli kabukta yazdığımız kodlar kalıcı olmuyor. Etkileşimli kabuğu kapattığımız anda bunlar hafızadan siliniyor. Tabii ki yazdığımız programları başkalarıyla paylaşabilmek için bunları önce bir yere uygun şekilde kaydetmiş olmamız gerekiyor. Dolayısıyla artık yavaş yavaş asıl çalışma ortamımıza geçebiliriz. Çünkü bunu yapabilecek kadar temel bilgiye sahibiz şu anda.

1.8 Python Programlarını Kaydetmek

Buraya kadar olan tecrübelerimizden gördüğümüz gibi, etkileşimli kabuk gerçekten de oldukça pratik ve güçlü bir araçtır. Ama burada yazdığımız kodlar kalıcı olmuyor. Bizim istediğimiz şey ise, bir kod yazdığımızda o anı ölümsüzleştirmek... İşte bunun için, yazdığımız programları bir yere kaydetmemiz gerekiyor. Bu bölümde bunu nasıl yapacağımızı öğreneceğiz.

Aslında bizim bu aşamada ihtiyacımız olan tek şey basit bir metin düzenleyicidir. Eğer GNU/Linux üzerinde KDE masaüstü ortamını kullanıyorsanız Kwrite veya Kate işinizi görecektir. Eğer kullandığınız sistem GNU/Linux üzerinde GNOME masaüstü ortamı ise Gedit sizin için yeterli olacaktır. Windows kullanıcıları, bilgisayarlarına kurdukları Python programı ile birlikte gelen IDLE adlı yazılımı kullanabilirler.

Windows kullananlar, IDLE'ye *Başlat > Programlar > Python 3.x > IDLE (Python GUI)* yolunu takip ederek ulaşabilir. IDLE'yi ilk başlattığınızda karşınıza "Python Shell" başlıklı bir ekran gelecektir. Aslında bu daha önce bahsettiğimiz ve şimdiye kadar hep üzerinde çalıştığımız etkileşimli kabuğun kendisidir... Burayı da etkileşimli kabuğu kullandığımız gibi kullanabiliriz. Ama bizim şimdi bahsedeceğimiz özellik bu değildir. Bizim ihtiyacımız olan şey bir metin düzenleyici. IDLE'nin metin düzenleyicisine ulaşmak için *File > New Window* yolunu takip etmeliyiz. Veya kısaca "CTRL+N" tuşlarına basarak da metin düzenleyiciyi açabiliriz. *File > New Window* yolunu takip ederek veya kısaca "CTRL+N" tuşlarına basarak ulaştığımız ekran, Python kodlarımızı yazacağımız alandır. Bu alanı nasıl kullanacağımızı biraz sonra göreceğiz. Ama önce GNU/Linux kullanıcılarının neler yapacağına bir bakalım...

Dediğim gibi, eğer GNU/Linux üzerindeyseniz Kwrite, Kate veya Gedit programlarını kullanabilirsiniz. Bu metin düzenleyici programlara ulaşmanın en kolay yolu "komut çalıştır" penceresinde bu programların ismini yazıp enter'e basmaktır... Muhtemelen bunun nasıl yapılacağını biliyorsunuz, ama ben yine de kısaca anlatayım:

Önce ALT+F2 tuşlarına basıyoruz. Eğer KDE kullanıyorsak, Kwrite'yi çalıştırmak için, açılan pencerede şu komutu veriyoruz:

```
kwrite
```

Kate için şu komutu:

```
kate
```

Eğer bir GNOME kullanıcısıysak, Gedit'i çalıştırmak için şöyle bir komut veriyoruz:

```
gedit
```

Yukarıdaki komutlar arasından kendinize uygun olanı verdiğinizde karşınıza boş bir metin düzenleyici gelecek. İşte Python kodlarımızı buraya yazacağız.

Aslında IDLE, Kwrite, Kate veya Gedit dışında, Python programlarımızı yazıp çalıştırmak için kullanabileceğimiz pek çok uygulama bulunur. Programlarımızı yazmak ve çalıştırmak için kullanabileceğimiz bu özel uygulamalara "IDE" (Geliştirme Ortamı) adı verilir. Bazı IDE'ler bize oldukça gelişmiş özellikler sunar. Python için geliştirilmiş IDE'lerin bir listesi için şu adresi ziyaret edebilirsiniz: http://wiki.python.org/moin/IntegratedDevelopmentEnvironments.

Her bir IDE'nin kendine özgü bir çalışma mantığı ve şekli vardır. Biz burada bu IDE'leri tek tek incelemeyeceğiz. Bizim için şu aşamada, yazılan kodları bilgisayarımıza kaydetmemizi sağlayacak herhangi bir program yeterli olacaktır. Hatta Notepad uygulaması bile bu iş için yeterlidir... Ama tabii ki Notepad'dan biraz daha gelişmiş bir metin düzenleyici kullanmak hayatımızı kolaylaştıracaktır. Dolayısıyla, Kwrite, Kate, Gedit veya IDLE şu aşamada bizim için yeter de artar bile... Neyse... Lafı çok fazla uzattık. Asıl konumuza gelmeliyiz artık.

Şimdi kendi sistemimize uygun olarak Kwrite, Kate, Gedit veya IDLE programlarından birini başlatalım. (IDLE kullanıcıları *File > New Window* yolunu takip ederek boş bir sayfa açmayı unutmuyorlar...)

Açtığımız boş metin sayfasına şu satırı yazıyoruz:

```
#!/usr/bin/env python3.0
```

Bu satır sadece GNU/Linux kullanıcıları içindir. Windows kullanıcıları bu satırı yazmasa da olur... Bu satırı yazmamızın amacı, GNU/Linux'ta programımızı çalıştıracağımız zaman Python'un hangi sürümünü kullanmak istediğimizi açıkça belli etmektir... Bu satırın önemini biraz sonra ayrıntılı olarak anlatacağız. Ama şimdi programımızı yazmaya devam edelim. İkinci satırımız şöyle:

```
print("Merhaba Zalim Dünya!")
```

İlk programımız bu kadar. İsterseniz bu kodları bir arada görelim:

```
#!/usr/bin/env python3.0

print("Merhaba Zalim Dünya!")
```

Programımızı yazdığımıza göre sıra geldi bu programı bilgisayarımıza kaydetmeye... Kullandığımız metin düzenleyicinin kaydetme özelliğini kullanarak bu dosyayı "ilk.py" adıyla (tırnaklar olmadan) bilgisayarımıza kaydedelim. Gelin isterseniz ilk programımızı masaüstüne kaydedelim, ki ulaşması kolay olsun. Programımızı kaydettiğimize göre, artık bu programı çalıştırabiliriz. Kaydettiğimiz bu programı nasıl çalıştıracağımızı bir sonraki bölüme bırakalım...

1.9 Python Programlarını Çalıştırmak

Bir önceki bölümde, bir Python programını nasıl yazıp kaydedebileceğimizi öğrendik. Bu bölümde ise yazıp kaydettiğimiz bu programı nasıl çalıştıracağımızı inceleyeceğiz.

Önce kendi sistemimize uygun olarak bir konsol veya DOS ekranı açıyoruz. GNU/Linux KDE kullanıcıları ALT+F2 ile gelen pencerede "konsole" komutunu kullanarak komut satırına ulaşabilir. GNOME kullanıcılarının aynı iş için kullanması gereken komut "gnome-terminal"dir. Windows kullanıcıları ise *Başlat > Çalıştır* yolunu takip ederek açılan pencerede "cmd" komutuyla DOS ekranına erişebilir.

Komut satırını ilk açtığınızda bulunduğunuz dizin muhtemelen masaüstü olmayacaktır. Ama biz programımızı masaüstüne kaydettiğimiz için, öncelikle dizinimizi masaüstü olarak değiştirmemiz gerekiyor. Bütün platformlarda şu komutu vererek masaüstüne geçebilirsiniz:

```
cd Desktop
```

Masaüstüne geldikten sonra GNU/Linux kullanıcıları şu komutu veriyor:

```
python3.0 ilk.py
```

Windows kullanıcıları ise şu komutu:

```
python ilk.py
```

Windows kullanıcıları bu komutu verdiklerinde program muhtemelen çalışacaktır. Ama eğer sistemde başka bir Python sürümü kuruluysa, programı o sürüm çalıştırmış olabilir. Lütfen okumaya devam ediniz...

Eğer her şey yolunda gitmişse konsol veya DOS ekranına şu yazının döküldüğünü göreceğiz:

```
Merhaba Zalim Dünya!
```

Tebrikler! Böylece gerçek anlamda ilk Python programınızı yazmış oldunuz. Henüz programımız önemli bir iş yapmıyor. Ama bir Python programının sahip olması gereken bütün temel özellikleri taşıyor. Şimdilik bizim için önemli olan tek şey, bu basit programı çalıştırabiliyor olmaktır... Yalnız bu noktada bir-iki ufak not düşelim:

Temel kural şudur: Python'un etkileşimli kabuğunu çalıştırmak için hangi komutu kullanıyorsanız, yazdığınız programı çalıştırmak için de o komutu kullanacaksınız. Yukarıda gördüğünüz gibi, Windows için "python ilk.py" komutunu kullandık. Tabii bu komut, sistemlerinde sadece Python 3.x sürümü kurulu olan kullanıcılar için geçerlidir. Eğer sisteminizde Python 3.x ile birlikte Python'un eski sürümleri de kuruluysa (mesela Python2.6), bu komutu verdiğinizde aslında programınız Python2.6 ile çalışmış olabilir. Ama siz aynen bu sitede anlatıldığı şekilde .bat dosyaları hazırlayarak Python sürümlerini birbirinden ayırdıysanız, programınızın Python3.x sürümüyle çalışmasını garanti etmek için vermeniz gereken komut şu olacaktır:

```
python3 ilk.py
```

Tabii ki, eğer siz .bat dosyasının adını farklı bir şey yapmışsanız o ismi kullanmanız gerekir. Mesela benim Windows yüklü bilgisayarımda, daha önce de söylediğim gibi, Python2.5, Python2.6 ve Python3.0 sürümleri bir arada kurulu. Windows'taki Python'a ait bütün çalıştırılabilir dosyaların adı "python.exe" olduğu için, ben Python 2.6 sürümünü YOL'a ekledim. Öteki sürümler için ise, "python25.bat" ve "python3.bat" adlı iki ayrı .bat dosyası yazdım. Dolayısıyla ben herhangi bir Python programını Python2.5 sürümüyle çalıştırmak istersem şu komutu veriyorum:

```
python25 program_adı.py
```

Veya bir programı Python2.6 ile çalıştırmak istediğimde sadece şu komutu veriyorum:

```
python program_adı.py
```

Çünkü Python2.6'yı daha önceden YOL'a eklemiştim... Eğer bir programı Python 3.x ile çalıştırmam gerekirse de şu komutu kullanıyorum:

```
python3 program_adı.py
```

GNU/Linux kullanıcıları için ise şu komutu kullandık:

```
python3.0 ilk.py
```

Tabii eğer siz Python'un etkileşimli kabuğunu çalıştırmak için "python3" komutunu kullanıyorsanız ilk.py adlı programı şöyle de çalıştırabilirsiniz:

```
python3 ilk.py
```

Dediğim gibi, işin özü şu: Python'un etkileşimli kabuğunu nasıl başlatıyorsanız, programlarınızı da öyle çalıştıracaksınız. Yani eğer etkileşimli kabuğu:

```
python
```

komutuyla başlatıyorsanız, ilk.py adlı programımızı çalıştırmak için şu komutu vereceksiniz:

```
python ilk.py
```

Eğer etkileşimli kabuğu:

```
python3
```

komutuyla çalıştırıyorsanız, programımızı şu komutla çalıştıracaksınız:

```
python3 ilk.py
```

Eğer etkileşimli kabuğu çalıştırmak için verdiğiniz komut:

```
python3.0
```

ise, ilk.py'yi şöyle çalıştırıyoruz:

```
python3.0 ilk.py
```

Bu noktada IDLE kullanıcıları için ufak bir not düşelim: IDLE kullananlar, programı bilgisayara kaydettikten sonra, sadece F5 tuşuna basarak programlarını çalıştırabilir. Ayrıca şunu da söyleyelim: IDLE programı, .py uzantılı dosyaların sağ tık menüsüne yerleşecektir. Yani bilgisayarınızdaki herhangi bir Python dosyasına sağ tıkladığınızda, menüde "Edit with IDLE" adlı bir seçenek de göreceksiniz. Bu seçeneği kullanarak .py uzantılı dosyayı IDLE ile düzenlemek üzere açabilirsiniz. Bu arada ufak bir not daha düşelim: IDLE adlı program Python'u yazan kişi olan Guido Van Rossum tarafından geliştirilmiştir...

Peki bir soru soralım: Acaba Python programlarını çalıştırmak için hep başlarına böyle "python" diye yazmak zorunda mıyız? Elbette hayır. Programlarımızı isimleriyle de çağırabiliriz:

Aşağıda söyleyeceklerim GNU/Linux kullanıcıları için geçerli:

Burada, ilk.py adlı programımızın ilk satırına yerleştirdiğimiz "*#!/usr/bin/env python3.0*" satırının önemi ortaya çıkıyor. Programlarımızı isimleriyle çalıştırabilmek için bir defa bu satırı mutlaka yazmalıyız. İkincisi, program dosyamızı "çalıştırılabilir" olarak ayarlamalıyız. Bunu yapmak için program dosyasına sağ tıklayıp "özellikler" menüsü içindeki "izinler" sekmesine geldikten sonra, oradaki "çalıştırılabilir" adlı kutucuğu işaretlememiz gerekir. Aynı işlemi konsoldan şu komutu vererek de yapabiliriz:

```
chmod a+x ilk.py
```

Böylece dosyamıza çalıştırma yetkisi vermiş olduk. Bundan sonra programımızı şu komutla çalıştırabiliriz:

```
./ilk.py
```

Artık programımızın .py uzantısını silip, adını "ilk" olarak değiştirdikten sonra konsolda doğrudan şu komutu vererek de programımızı çalıştırabiliriz:

```
./ilk
```

Peki ya biz programımızı tıpkı öteki programlar gibi sadece adını kullanarak çalıştırmak istiyorsak ne yapacağız?

Bunun için programımızı */usr/bin/* klasörünün içine atmamız gerekir. Programımızı */usr/bin/* içine attıktan sonra sadece şu komutu vermemiz yeterli olacaktır:

```
ilk
```

Gelelim Windows kullanıcılarına:

Windows kullanıcıları, yazdıkları programı sadece ismiyle çalıştırmak için DOS ekranında "cd Desktop" komutuyla masaüstüne geldikten sonra şu komutu verebilir:

```
deneme.py
```

Bu komut, programı otomatik olarak sistemdeki en son Python sürümüyle çalıştıracaktır.

Windows kullanıcıları ayrıca program dosyasına çift tıklayarak da programlarını çalıştırabilir. Çünkü Python sisteme kurulurken kendini Windows kütüğüne (registry) kaydeder. Dolayısıyla Windows, .py uzantılı bir dosyayı hangi programla çalıştırması gerektiğini bilir. Ancak dosyaya çift tıklandığında Windows bu programı Python'un yine en son sürümüyle çalıştıracaktır.

Windows'ta, yazdığınız programa çift tıkladığınızda bir DOS ekranının hızla yanıp söndüğünü göreceksiniz. Aslında programınız çalışıyor ve ekrana "Merhaba Zalim Dünya!" yazısını yazdırıyor, ama çalışma süreci çok hızlı olduğu için bu çıktı ekrana verilir verilmez, programın sonuna gelindiğinden ötürü DOS ekranı hemen kapanıyor. İsterseniz ilk.py adlı dosyayı şu şekilde yazarak DOS ekranının hemencecik kapanmasını engelleyebilirsiniz:

```
print("Merhaba Zalim Dünya!")

input()
```

Böylelikle, program dosyasına çift tıkladığınızda DOS ekranı açılacak, "Merhaba Zalim Dünya!" çıktısı ekrana verilecek ve siz enter tuşuna basana kadar da DOS ekranı açık kalacaktır... Ancak bu yöntem her zaman tercih edilmeyebilir. Çünkü bu şekilde, eğer programda bir hata varsa programın verdiği hataları göremezsiniz. Hataları görebilmek için en doğru yol programı komut satırından çalıştırmaktır. Böylece eğer programda bir hata varsa komut ekranında bunları görebilir ve hataları düzeltebilirsiniz. Buna bir örnek verelim. Diyelim ki programımızda şöyle bir hata yaptık:

```
print("Merhaba Zalim Dünya!"

input()
```

Gördüğünüz gibi, "*print("Merhaba Zalim Dünya!"* satırında kapanış parantezini koymayı unuttuk ve programımızı bu şekilde kaydettik. Şimdi program dosyamıza çift tıkladığımızda DOS ekranı `input()` satırına rağmen anlık olarak ekranda görünüp kaybolacaktır. Bu şekilde sorunun nereden kaynaklandığını anlayamazsınız. Ama eğer bu programı komut satırından çalıştırırsanız, en azından programdaki hata konusunda bir fikir sahibi olabilir, buna göre programa müdahale edebilirsiniz.

Böylece Python programlarımızı nasıl çalıştırabileceğimiz öğrenmiş olduk. Artık hızla yolumuza devam edebiliriz.

1.10 Python'da İlk Örnekler...

Buraya kadar Python'la ilgili birkaç şey öğrendik... Öğrendiklerimiz henüz yeterli değil elbette. Ama en azından elimizi alıştırmak için dahi olsa birkaç örnek yapabiliriz. Bu bölüm aynı zamanda bir nevi, önceki konuların tekrarı gibi de olacaktır. Daha önce öğrendiğimiz konuları burada tekrar etme fırsatı bulmanın yanısıra, bu derste yeni şeyler de öğreneceğiz. O halde hemen işe koyulalım.

İlk iş olarak, daha önceki derslerde öğrendiğimiz şekilde boş bir metin sayfası açıyoruz. Bu iş için Kwrite, Kate, Gedit, IDLE ve hatta Notepad gibi programlardan herhangi birini kullanabiliriz. Şimdi, tercih ettiğimiz metin düzenleyiciyi kullanarak boş bir sayfa açalım ve içine şunları yazalım:

```
#!/usr/bin/env python3.0

a = "kalem"
b = "pergel"
c = "çikolata"

print("Bir", a, "bir", b, "bir de", c, "alacağım...")
```

Buradaki ilk satır sadece GNU/Linux kullanıcıları içindir. Windows kullanıcıları bu satırı yazmayabilirler. Burada "a", "b" ve "c" adlı değişkenlere sırasıyla "kalem", "pergel" ve "çikolata" değerlerini nasıl atadığımıza dikkat edin. Değerlerimizi tırnak içine almayı unutmuyoruz. Çünkü bunlar birer karakter dizisidir.

En son satırda ise, ilk derslerde öğrendiğimiz `print()` fonksiyonunu görüyoruz. Daha önce tanımladığımız değişkenleri, `print()` fonksiyonu içine tek tek yerleştirdik. Değişken adlarını tırnak içine almadığımıza dikkat edin. Ayrıca bu kodları yazarken, virgülleri de yanlış yerlere koymamaya özen gösteriyoruz.

Şimdi artık bu kodları .py uzantısıyla kaydedebiliriz. Mesela "deneme.py" adıyla kaydedelim kodlarımızı... Böylelikle elimizde, Python'la yazılmış bir program örneği olmuş oldu. Şimdi bu programı çalıştıracağız. Bunun için, kullandığımız işletim sistemine uygun olarak hemen bir komut satırı açıyoruz. Komut satırını nasıl açacağımızı daha önceki derslerimizde anlatmıştık. Ama isterseniz yine de kısaca değinelim:

GNU/Linux'ta KDE masaüstü ortamını kullananlar "ALT+F2" tuşlarına basıp, açılan pencerede "konsole" komutunu vererek; GNOME masaüstü ortamını kullananlar ise aynı pencerede "gnome-terminal" komutunu vererek bir komut satırı açabilirler. Windows kullanıcıları ise *Başlat > Çalıştır* yolunu takip ederek, açılan pencerede "cmd" komutunu verdikten sonra MS-DOS ekranına ulaşabilir. Komut satırına ulaştıktan sonra, orada şu komutu verelim:

```
python3 deneme.py
```

Tabii eğer program dosyanızı masaüstüne kaydettiyseniz:

```
cd Desktop
```

komutu yardımıyla öncelikle masaüstünün bulunduğu dizine gelmeniz gerekir...

Bir de ben burada dosyayı "deneme.py" adıyla kaydettiğinizi varsaydım. Eğer farklı bir dosya adı kullandıysanız, burada o adı yazmalısınız. Ayrıca dosya adının önüne "python3" yazdığıma dikkat edin. Temel kuralımız şuydu: Python'un etkileşimli kabuğuna ulaşmak için hangi komutu kullanıyorsanız burada da o komutu kullanacaksınız. Yani eğer etkileşimli kabuğu "python3.0" diyerek açıyorsanız şu komutu vermeniz gerekir:

```
python3.0 deneme.py
```

Yok eğer Python 3.x'in etkileşimli kabuğunu sadece "python" komutuyla açıyorsanız, tabii ki vereceğiniz komut şu olacaktır:

```
python deneme.py
```

Eğer daha önceki derslerde açıkladığımız şekilde gerekli ayarları yaparsanız, programımızı sadece adıyla veya üzerine çift tıklayarak da çalıştırabileceğimizi biliyorsunuz.

Kendimize uygun olan komutu verip yukarıdaki programı çalıştırdığımızda şöyle bir çıktı elde edeceğiz:

```
Bir kalem bir pergel bir de çikolata alacağım...
```

Gelin isterseniz bu kodlara yeni bir satır daha ekleyelim. Dosyanın en son satırından sonra şunu yazalım:

```
print("Fişini almayacak mısın oğlum?")
```

Yani kodlarımız şöyle olmuş oldu:

```
#!/usr/bin/env python3.0

a = "kalem"
b = "pergel"
c = "çikolata"

print("Bir", a, "bir", b, "bir de", c, "alacağım...")
print("Fişini almayacak mısın oğlum?")
```

Bu kodları çalıştırdığımızda şöyle bir çıktı alırız:

```
Bir kalem bir pergel bir de çikolata alacağım...
Fişini almayacak mısın oğlum!
```

Gördüğünüz gibi, ikinci `print()` fonksiyonu ile verdiğimiz çıktı otomatik olarak bir alt satırda görünüyor. Çoğu zaman istediğimiz davranış bu olacaktır. Ama bazen istediğimiz şey bu olmayabilir. Bunun yerine, yeni eklenen satırın bir öncekiyle aynı satırda görünmesini isteyebiliriz. O zaman, Python'un bize sunduğu "end" adlı ifadeden yararlanabiliriz:

```
#!/usr/bin/env python3.0

a = "kalem"
b = "pergel"
c = "çikolata"

print("Bir", a, "bir", b, "bir de", c, "alacağım...",
       end="Fişini almayacak mısın oğlum?")
```

Bu kodların çıktısı şöyle olacaktır:

```
Bir kalem bir pergel bir de çikolata alacağım... Fişini almayacak mısın
oğlum!istihza@istihza:~/Desktop$
```

Gördüğünüz gibi, çıktımız hiçbir şekilde biçimlendirilmedi. Öyle ki, çıktımız komut satırındaki "istihza@istihza:~/Desktop$" ifadesinin dahi gerisinde kaldı! Bu kullanımda ne bir boşluk karakteri otomatik olarak eklenir, ne de otomatik olarak yeni satıra geçilir. Arzu ettiğimiz işaretleri kendimiz koymamız lazım. Mesela bu kodları şu şekilde yazarak daha düzgün bir çıktı alabiliriz:

```
#!/usr/bin/env python3.0

a = "kalem"
b = "pergel"
c = "çikolata"

print("Bir", a, "bir", b, "bir de", c, "alacağım...",
end="Fişini almayacak mısın oğlum?\n")
```

Bu defa çıktımız şöyle olacak:

```
Bir kalem bir pergel bir de çikolata alacağım... Fişini almayacak mısın oğlum!
```

Bu kez hem üç noktadan sonra bir boşluk var, hem de çıktımız verildikten sonra bir alt satıra geçiliyor. Bunu nasıl becerdiğimize bakalım:

“end=” ifadesinin ardından bir boşluk bırakarak çıktının da bir adet boşluk karakteri içermesini sağladık. “end=” ile eklediğimiz karakter dizisinin en sonuna şöyle bir işaret koyduk: “\n”. Buna programlama dillerinde “yeni satır karakteri” adı verilir. Bir cümleyi herhangi bir yerinden bölüp alt satıra geçmek istediğimizde bu karakteri kullanabiliriz. İsterseniz bununla ilgili birkaç örnek yapalım:

Diyelim ki elimizde şöyle bir karakter dizisi var:

> *Guido Van Rossum Python’u geliştirmeye 1990 yılında başlamış... Yani aslında Python için nispeten yeni bir dil denebilir. Ancak Python’un çok uzun bir geçmişi olmasa da, bu dil öteki dillere kıyasla kolay olması, hızlı olması, ayrı bir derleyici programa ihtiyaç duymaması ve bunun gibi pek çok nedenden ötürü çoğu kimsenin gözdesi haline gelmiştir. Ayrıca Google’nin de Python’a özel bir önem ve değer verdiğini, çok iyi derecede Python bilenlere iş olanağı sunduğunu da hemen söyleyelim. Mesela bundan kısa bir süre önce Python’un yaratıcısı Guido Van Rossum Google’de işe başladı...*

Bizim amacımız bu uzun cümleyi ekrana yazdırmak, ama yazdırırken de belli noktalarından bölmek olsun. Bunun için şöyle bir şey yazabiliriz:

```
#!/usr/bin/env python3.0

print("\nPython Hakkında:")
print()
print("Guido Van Rossum Python’u geliştirmeye 1990 yılında başlamış...\nYani \
aslında Python için nispeten yeni bir dil denebilir.\nAncak Python’un çok uzun\
bir geçmişi olmasa da,\nbu dil öteki dillere kıyasla kolay olması, hızlı olması,\nayrı\
bir derleyici programa ihtiyaç duymaması\nve bunun gibi pek çok nedenden ötürü\
çoğu kimsenin\ngözdesi haline gelmiştir. Ayrıca Google’nin de Python’a özel bir\
önem\nve değer verdiğini, çok iyi derecede Python bilenlere iş olanağı\nsunduğunu\
da hemen söyleyelim. Mesela bundan kısa bir süre önce\nPython’un yaratıcısı Guido\
Van Rossum Google’de işe başladı...")
```

Gördüğünüz gibi, yeni bir satırda yer almasını istediğimiz kısımlara “\n” karakterini yerleştirdik. Ayrıca bu uzun karakter dizisini belli noktalarda enter tuşuna basarak bölmek istediğimizde de “\” işaretinden yararlanıyoruz. Eğer enter tuşuna basacağımız noktalarda bu “ters bölü” işaretini kullanmazsak programımız hata verecektir... Bu ters bölü işaretleri çıktıda görünmez, hatta bu işaretlerin çıktı üzerinde hiçbir etkisi yoktur. Bu işaret yardımıyla Python’a şöyle bir şey söylemiş oluyoruz:

> “*enter tuşuna bastığıma bakma! Aslında satır devam ediyor...*“

Bu kodları çalıştırdığımızda şuna benzer bir çıktı alırız:

```
istihza@istihza:~/Desktop$ python3 deneme.py

Python Hakkında:

Guido Van Rossum Python’u geliştirmeye 1990 yılında başlamış...
Yani aslında Python için nispeten yeni bir dil denebilir.
Ancak Python’un çok uzun bir geçmişi olmasa da,
bu dil öteki dillere kıyasla kolay olması, hızlı olması,
ayrı bir derleyici programa ihtiyaç duymaması
ve bunun gibi pek çok nedenden ötürü çoğu kimsenin
gözdesi haline gelmiştir. Ayrıca Google’nin de Python’a özel bir önem
ve değer verdiğini, çok iyi derecede Python bilenlere iş olanağı
```

```
sunduğunu da hemen söyleyelim. Mesela bundan kısa bir süre önce
Python'un yaratıcısı Guido Van Rossum Google'de işe başladı...
```

Gördüğünüz gibi, satırlar "\" işaretinin bulunduğu yerlerden değil, "\n" işaretinin olduğu noktalardan bölünüyor... Ayrıca burada "\n" karakterini kelimeler arasına yerleştirirken boşluk bırakmadığımıza da dikkat edin. Çünkü "\n" karakteri kendisinden sonra gelen kelimeyi bir alt satıra alacağı için, eğer kelimeler arasında boşluk bırakırsak, alt satıra geçen kelime, yeni satırda girintili olarak görünecektir...

Yukarıda da ifade ettiğimiz gibi, böyle uzun cümlelerle çalışırken, eğer kendiniz enter tuşuna basarak alt satıra geçmeye çalışırsanız, programınız hata verecektir. Bunu önlemek için "\" işaretinden yararlanmamız gerekir. Bu durum Python'da tek ve çift tırnak işaretlerinin özelliğidir.

Eğer karakter dizilerini yazarken alt satıra geçmek istiyorsanız bu işi "\n" adlı karakterle yapmanız gerekir. Dediğim gibi, aksi halde hata alırsınız...

Gördüğünüz gibi yukarıdaki gibi bir kod yazabilmek için epey takla atmamız gerekti... Ama Python'da çareler tükenmez! Bunun gibi uzun cümlelerle çalışırken işimize yarayacak başka bir tırnak tipi daha vardır Python'da: "üç tırnak"

"üç tırnak" işaretini kullanarak, çok uzun cümleleri sorunsuz ve dertsiz bir şekilde ekrana yazdırabiliriz. Mesela yukarıdaki örnek üzerinden gidelim:

```
#!/usr/bin/env python3.0

print("""
Python Hakkında:
""")

print("""Guido Van Rossum Python'u geliştirmeye 1990 yılında başlamış...
Yani aslında Python için nispeten yeni bir dil denebilir. Ancak
Python'un çok uzun bir geçmişi olmasa da, bu dil öteki dillere kıyasla
kolay olması, hızlı olması, ayrı bir derleyici programa ihtiyaç duymaması
ve bunun gibi pek çok nedenden ötürü çoğu kimsenin gözdesi haline gelmiştir.
Ayrıca Google'nin de Python'a özel bir önem ve değer verdiğini, çok iyi derecede
Python bilenlere iş olanağı sunduğunu da hemen söyleyelim. Mesela bundan kısa
bir süre önce Python'un yaratıcısı Guido Van Rossum Google'de işe başladı...""")
```

Burada """ (üç tırnak) işaretini nasıl kullandığımıza dikkat edin. Bir tane bile "\n" karakteri kullanmadan, klavyedeki enter tuşunu kullanarak, istediğimiz yerden böldük cümlelerimizi. Aynı şeyi çift veya tek tırnak ile yapmaya çalıştığımızda programımızın hata verdiğini görürüz...

Buraya kadar öğrendiğimiz bilgilere göre Python'da üç adet tırnak tipi kullanılabiliyor:

1. Tek tırnak (')
2. Çift tırnak (")
3. Üç tırnak (""")

Tek ve çift tırnaklar kısa karakter dizilerinde kullanılmaya oldukça uygundur. Elbette kısa karakter dizileriyle de üç tırnağı kullanabilirsiniz.Ama kısa karakter dizileri için üç tırnak kullanımı pek yaygın değildir. Zaten mesela "elma" gibi kısa bir karakter dizisini """elma""" şeklinde yazmak en basitinden "çirkin" bir görünüm sunacak, hiç de pratik olmayacaktır...

Yukarıdaki kodları çok dikkatli bir şekilde inceleyip, hangi kodun ne işe yaradığını anlamaya çalışmanızı öneririm. Hatta yukarıdaki kodlarda kendinize göre birtakım değişiklikler yaparak, kodların işlevini daha iyi kavrayabilirsiniz.

İsterseniz Python'da tırnak işaretlerinin işlevini daha iyi kavrayabilmek için birkaç örnek daha yapalım. Mesela şöyle bir cümleyi ekrana nasıl yazdırırsınız:

```
*Bugün Adana'ya gidiyoruz!*
```

Bu cümleyi en kolay şu şekilde yazdırabiliriz:

```
>>> print("Bugün Adana'ya gidiyoruz!")

Bugün Adana'ya gidiyoruz!
```

"Adana'ya" kelimesinde geçen kesme işareti nedeniyle tek tırnak kullanmıyoruz. Aksi halde Python'un kafası karışacak, tırnakların nerede başlayıp nerede bittiğini anlayamayacaktır.

Bir de şu cümleyi yazdırmayı deneyelim:

```
"Bugün günlerden Perşembe," dedi Bihter Hanım
```

Tahmin ettiğiniz gibi, cümle içinde çift tırnaklar olduğu için, bu cümleyi en kolay şu şekilde yazdırırız:

```
>>> print('"Bugün günlerden Perşembe," dedi Bihter Hanım')

Bugün günlerden Perşembe," dedi Bihter Hanım
```

Burada başta ve sonda tek tırnak işaretlerini kullanıyoruz...

Peki ya şu cümle nasıl yazdırılır?

```
Ahmet, "Bugün Adana'ya gidiyoruz," dedi.
```

Burada hem tek tırnak hem de çift tırnak var. Acaba bu sorunu nasıl aşacağız?

Bu sorunu en kolay, üç tırnak işaretlerini kullanarak çözebiliriz:

```
>>> print("""Ahmet, "Bugün Adana'ya gidiyoruz," dedi.""")
```

Gördüğünüz gibi sorun kolayca çözüldü. Peki ya biz burada üç tırnak işaretini kullanmak istemezsek ne olacak? Diyelim ki biz bu cümleyi mutlaka çift tırnak kullanarak yazdırmak istiyoruz. O zaman ne yapacağız?

Burada, Python'un özel işaretleri devreye girecek. Bu özel işaretlere programlama dilinde "kaçış dizisi" (escape sequence) adı verilir. Mesela "\" işareti bunlardan biridir ve bizim yukarıdaki gibi bir problemden "kaçmamızı" sağlar. Hemen görelim nasıl kullanıldığını bu işaretin:

```
>>> print("Ahmet, \"Bugün Adana'ya gidiyoruz,\" dedi.")
```

"\" adlı kaçış dizisini nasıl kullandığımızı görüyorsunuz. Başlangıç ve bitiş tırnakları dışında kalan bütün çift tırnak işaretlerinin önüne "\" adlı kaçış dizisini getiriyoruz.

Eğer yukarıdaki cümleyi çift tırnak yerine mutlaka tek tırnak kullanarak yazdırmak istiyor olsaydık, şöyle yapacaktık:

```
>>> print('Ahmet, "Bugün Adana\'ya gidiyoruz," dedi.')
```

Burada da, kaçmamız gereken bir adet kesme işareti var. Cümlemize tek tırnakla başladığımız için cümle içinde geçen çift tırnakların zararı olmayacak. Ama "Adana'ya" kelimesindeki kesme işareti bize sorun çıkaracak. Bu işaretin sorun yaratmaması için "\" adlı kaçış dizisini kullanarak sorundan "kaçıyoruz"...

Python'da "\" karakterinden başka pek çok kaçış dizisi vardır. Bunları ilerleyen derslerde tek tek inceleyeceğiz.

Karakter dizileri üzerinde yeterince çalıştık sayılır. İsterseniz biraz da "sayılar"la ilgili örnekler yapalım...

Önceki derslerden hatırladığımız gibi, karakter dizileriyle sayılar arasında iki temel fark bulunuyordu:

1. Karakter dizileri tırnak içinde gösterilir, sayılar tırnaksız...
2. Sayılar, adlarından anlaşılacağı gibi sayı değerli öğelerdir. Karakter dizileri ise sayı değerli olabilecekleri gibi, olmayabilirler de...

Önceki bilgilerimize dayanarak şu örnekleri verebiliriz. Mesela aşağıdaki ifade bir sayıdır:

```
13
```

Ama şu ifade bir sayı değildir:

```
"13"
```

Ancak şöyle bir yanılgıya düşmeyelim. Sayı olup olmama özelliği yalnızca tırnak işaretleriyle ilgili bir durum değildir. Örneğin aşağıdaki karakter dizisinin tırnaklarını kesip aldığımızda bir sayı elde etmeyiz:

```
"elma"
```

Yani:

```
elma
```

gibi bir kullanım hata verecektir. Çünkü bir ifadenin sayı olabilmesi için, tırnak işareti taşımamasının yanısıra, sayı değerli olması da gerekir...

İsterseniz sayıları (ve karakter dizilerini) kullanarak şöyle bir örnek yapalım:

```
#!/usr/bin/env python3.0

km = 10
m = 50000

print(m, "metre", m/1000, "kilometreye eşittir.")
print(km, "kilometre", km*1000, "metreye eşittir.")
```

Bu program şöyle bir çıktı verecektir:

```
50000 metre 50.0 kilometreye eşittir.
10 kilometre 10000 metreye eşittir.
```

Eğer "km" ve "m" değişkenlerinin değeri sayı değil de karakter dizisi olsaydı, "*m/1000*" veya "*km*1000*" işlemlerini yapamazdık.

Bu arada, buradaki çıktıda "50.0" gibi ondalık değer taşıyan bir sayı elde ettiğimize dikkat edin. Eğer ondalık sayı yerine tamsayı elde etmek isterseniz, bölme işlemini şu şekilde yapabilirsiniz:

```
m//1000
```

Önceki derslerimizde "//" işlecinden söz etmiştik. Bu işleç yukarıdaki gibi durumlarda oldukça faydalı bir araçtır.

Böylelikle bir konuyu daha bitirmiş olduk. Bu derste, şimdiye kadar öğrendiğimiz konularla ilgili pek çok örnek yaptık. Burada anlatılanları daha iyi kavrayabilmek için kendi kendinize alıştırmalar yapmanızı tavsiye ederim.

1.11 Kullanıcıyla İletişim: input() fonksiyonu

Python'da artık bazı şeyleri yapabiliyoruz. Ama şimdiye kadar yaptıklarımız, kendimiz çalıp kendimiz oynamaktan pek farklı değil... Artık bir yolunu bulup kullanıcıyla iletişime geçebilmemiz lazım. Aksi halde tek yönlü bir programlama deneyiminin çok sıkıcı olacağı bariz...

Bu bölümde kullanıcıyla etkileşeceğiz. Yani ondan bir takım veriler alıp programımız içinde bu verileri işleyeceğiz. Her zamanki gibi, lafı hiç uzatmadan konunun özüne dalalım...

Python'da kullanıcıyla veri alış-verişi yapabilmek için `input()` adlı bir fonksiyondan yararlanacağız.

Hemen bir örnek yapalım.

Boş bir metin belgesi açıp içine şunları yazıyoruz:

```
#!/usr/bin/env python3.0

print("Merhaba, ben Python. Monty Python")
input("Senin adın nedir? ")
```

Bu programı çalıştırdığımız zaman, ekrana şöyle bir çıktı verilecektir:

```
Merhaba, ben Python. Monty Python
Senin adın nedir?
```

Burada Python kullanıcıya adını sordu ve cevap bekliyor. Kullanıcı kendi adını yazıp "enter" tuşuna bastığında programımız kapanacaktır. Tabii ki program bu haliyle bizi tatmin etmekten çok uzak. En azından, kullanıcının adını öğrenip ona kendi adıyla hitap edebilsek çok iyi olur... Şu kodlar bu dileğimizi yerine getirecektir:

```
#!/usr/bin/env python3.0

print("Merhaba, ben Python. Monty Python")
isim = input("Senin adın nedir? ")
print("Merhaba", isim)
```

Bu programı çalıştırdığımızda, kullanıcıya kendi adıyla hitap edebildiğimizi görüyoruz. Bunu yapabilmek için, öncelikle `input()` fonksiyonunun kendisini bir değişken içinde depoladık. Kullanıcı ekrana ismini yazdığında `input()` fonksiyonu bu bilgiyi alıp "isim" adlı değişken içinde saklayacaktır. Böylece bu veriyi daha sonra tekrar kullanabileceğiz. Hemen bir sonraki satırda da bu veriyi kullandığımıza dikkat edin. Daha önceki derslerde öğrendiğimiz bilgilerden pek farklı değil. Bu kodlardaki tek yenilik, araya bir input() fonksiyonunun yerleştirilmiş olması. Gördüğünüz gibi, `input()` fonksiyonunun kullanımı `print()` fonksiyonunun kullanımına biraz benziyor.

İsterseniz kullanıcıyla biraz sohbet edelim, ne dersiniz?

```
#!/usr/bin/env python3.0

print("Merhaba, ben Python. Monty Python")
isim = input("Senin adın nedir? ")
print("Merhaba", isim)
```

```
yaş = input("Peki yaşın kaç diye sorsam?")
print("Hmmm... Yaşıt değiliz seninle!")
```

...bu böyle gider...

Benzer bir örnek daha yaparak konuyu pekiştirelim:

```
#!/usr/bin/env python3.0

kare = input("Bir sayı gir, ben sana o sayının karesini söyleyeyim:")
print(kare, "sayısının karesi: ", kare**2)
```

Ne oldu? Python hata verdi, değil mi? Gayet normal. Çünkü `input()` fonksiyonu çıktı olarak bir karakter dizisi verir. Hatırlarsanız önceki bölümlerde şöyle bir şey demiştik:

> "tırnak işareti taşımayan "13" bir sayıdır, ama tırnak işareti taşıyan "13" bir sayı değildir..."

İşte bu `input()` fonksiyonu kullanılarak elde edilen değer de bir sayı değildir. Dolayısıyla kullanıcı ekrana bir sayı giriyormuş gibi görünse de aslında `input()` fonksiyonunun çıktı türü bir karakter dizisidir... Biz görmesek de, `input()` fonksiyonu ekrana yazılan ifadeleri tırnak içine alır... Bu bilgi önemlidir. Bunu aklımızda tutmaya çalışalım. Zira daha önce de dediğimiz gibi, aritmetik işlemleri ancak sayılar arasında yapılabilir. Karakter dizileri ile aritmetik işlemi yapılamaz.

Peki ne yapacağız? Böyle elimiz kolumuz bağlı oturacak mıyız? Elbette hayır! Yapacağımız işlem çok basit. `input()` ile elde edilen çıktıyı sayıya dönüştüreceğiz. Bu noktada biraz Python'daki sayı tiplerinden bahsetmekte fayda var:

Python'da temel olarak iki farklı sayı tipi vardır:

1. Tamsayılar (Integers)
2. Ondalık Sayılar (Floats)

Tamsayılar, ondalık bir kısım içermeyen sayılardır. Mesela "5", "20", "17" gibi sayılara tamsayı adı verilir.

Ondalık sayılar ise, içinde ondalık bir kısım barındıran sayılardır. Mesela, "12.7", "5.4", "56.8", "0.5" gibi sayılar ondalık sayılardır.

Bunların dışında Python'da bir de karmaşık sayılar (complex numbers) vardır. Karmaşık sayılar; bir gerçek ve bir sanal kısımdan oluşan sayılardır. Python'da karmaşık sayılar şu şekilde gösterilir: (9+3j), (1+2j), gibi...

Eğer matematikle çok içli dışlı değilseniz karmaşık sayılar pek işinize yaramayacaktır...

Birkaç küçük örnek ile yolumuza devam edelim. Aşağıdaki küçük örnekleri, uzun uzun metin dosyalarına yazmak yerine Python'un etkileşimli kabuğunu kullanarak yapalım. Zira etkileşimli kabuk bu tür ufak kod denemeleri için birebirdir:

```
>>> a = 56
```

Bu sayı bir "tamsayı"dır. İngilizce olarak ifade etmek gerekirse, "integer"... Bunun bir tamsayı olduğunu şu şekilde teyit edebiliriz:

```
>>> type(a)

<class 'int'>
```

Burada aldığımız "class int" çıktısı, bize "a" değişkeninin tuttuğu sayının bir "tamsayı" olduğunu söylüyor. "int" ifadesi, "integer" (tamsayı) kelimesinin kısaltmasıdır.

Bir de şu sayıya bakalım:

```
>>> b = 34.5
>>> type(b)

<class 'float'>
```

Bu çıktı ise bize "34.5" sayısının bir ondalık sayı olduğunu söylüyor. "float" kelimesi "Floats" veya "Floating Point Number" ifadesinin kısaltmasıdır. Yani "ondalık sayı" demektir...

Bu arada, yeni öğrendiğimiz bu `type()` adlı fonksiyonu sadece sayılara değil, başka şeylere de uygulayabiliriz. Mesela bir örnek vermek gerekirse:

```
>>> meyve = "karpuz"
>>> type(meyve)

<class 'str'>
```

Gördüğünüz gibi, `type()` fonksiyonu bize meyve adlı değişkenin değerinin bir "str" yani "string" yani "karakter dizisi" olduğunu bildirdi...

Şimdi de etkileşimli kabuktan çıkıp aşağıdaki gibi bir program yazalım:

```
#!/usr/bin/env python3.0

kare = input("Bir sayı giriniz lütfen:")
print(kare)
print("Girdiğiniz verinin tipi şudur:")
print(type(kare))
```

Burada, kullanıcıdan bir sayı girmesini istedik. Kullanıcı bir veri girip enter tuşuna bastığında, önce kullanıcının girdiği verinin kendisi ekrana yazdırılacak, ardından da bu verinin tipi ekrana çıktı olarak verilecektir.

Yukarıdaki programdan aldığımız çıktının gösterdiği gibi, `input()` fonksiyonu ile alınan verinin tipi "str", yani karakter dizisidir. Bizim yapmamız gereken şey, kullanıcıdan aldığımız bu karakter dizisini sayıya dönüştürmek olacaktır. Peki bu dönüştürme işlemi nasıl olacak? Hemen bakalım:

Diyelim ki elimizde şöyle bir sayı var (Bu arada, artık aşağıdaki kod parçalarını denemek için etkileşimli kabuğu kullanmanın daha pratik olduğunu söylememe gerek yok...):

```
>>> a = 45
```

"sayı" adlı değişkenin tuttuğu verinin değeri bir "tamsayı"dır. İsterseniz bunu type() fonksiyonunu kullanarak teyit edebileceğinizi biliyorsunuz... Biz bu tamsayıyı ondalık sayıya dönüştürmek istiyoruz. Yapacağımız işlem çok basit:

```
>>> float(a)

45.0
```

Gördüğünüz gibi, "45" adlı tamsayıyı, "45.0" adlı bir ondalık sayıya dönüştürdük. Şimdi *type(45.0)* komutu bize "*<class 'float'>*" çıktısını verecektir... Eğer ondalık bir sayıyı tamsayıya çevirmek istersek şu komutu veriyoruz. Mesela ondalık sayımız, "56.5" olsun:

```
>>> int(56.5)

56
```

Yukarıdaki örneği tabii ki şöyle de yazabiliriz:

```
>>> a = 56.5
>>> type(a)

56
```

Bu arada tekrar bir hatırlatma yapalım. Yukarıdaki ufak kod parçalarını etkileşimli kabukta çalıştırdığımız için `print()` fonksiyonunu kullanmamıza gerek kalmadan ekrana çıktı alabiliyoruz. Daha önce de dediğimiz gibi, bu durum etkileşimli kabuğun bir özelliğidir. Yukarıdaki kodları şu şekilde bir metin dosyasına yazarsak çıktı almamız mümkün olmaz:

```
#!/usr/bin/env python3.0

sayı = 56.5
type(sayı)
```

Bu programı çalıştırdığımızda ekranda hiçbir çıktı görünmeyecektir. Bu programın çalışması için `print()` fonksiyonunu kullanmamız lazım:

```
#!/usr/bin/env python3.0

sayı = 56.5
print(type(sayı))
```

Dönüştürme işlemini sayılar arasında yapabileceğimiz gibi, sayılar ve karakter dizileri arasında da yapabiliriz. Örneğin şu bir karakter dizisidir:

```
>>> nesne = "45"
```

Yukarıdaki değeri tırnak içinde belirttiğimiz için bu değer bir karakter dizisidir. Şimdi bunu bir tamsayıya çevireceğiz:

```
>>> int(nesne)

45
```

Dilersek, aynı karakter dizisini ondalık sayıya da çevirebiliriz:

```
>>> float(nesne)

45.0
```

Hatta bir sayıyı karakter dizisine de çevirebiliriz. Bunun için “string” (karakter dizisi) kelimesinin kısaltması olan “str” parçacığını kullanacağız:

```
>>> s = 6547
>>> str(s)

'6547'
```

Bir örnek de ondalık sayılarla yapalım:

```
>>> s = 65.7
>>> str(s)

'65.7'
```

Yalnız şunu unutmayın: Bir karakter dizisinin sayıya dönüştürülebilmesi için o karakter dizisinin sayı değerli olması lazım. Yani “45” değerini sayıya dönüştürebiliriz. Çünkü “45” değeri, tırnaklardan ötürü bir karakter dizisi de olsa, neticede sayı değerli bir karakter dizisidir. Ama

mesela “elma” karakter dizisi böyle değildir. Dolayısıyla, şöyle bir maceraya girişmek bizi hüsrana uğratacaktır:

```
>>> nesne = "elma"
>>> int(nesne)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'elma'
```

Gördüğünüz gibi, Python bu numarayı yutmadı!...

Neyse, olmayacak duaya amin demeyi bırakıp yolumuza devam edelim...

Gelin şimdi daha büyük bir işlem yapalım. Kullanıcıdan aldığımız veriyi tamsayıya çevirelim... Aşağıdaki kodları bir metin dosyasına yazıyoruz:

```
#!/usr/bin/env python3.0

kare = int(input("Bir sayı giriniz lütfen:"))
print(kare)

print("Girdiğiniz verinin tipi şudur:")
print(type(kare))
```

Gördüğünüz gibi, “type(kare)” fonksiyonu bu defa “class ‘int’” çıktısı verdi. Yukarıdaki kodlar kafanızı karıştırmasın. İstersek bu kodları şöyle de yazabiliriz:

```
#!/usr/bin/env python3.0

kare = input("Bir sayı giriniz lütfen:")
sayı = int(kare)

print(sayı)

print("Girdiğiniz verinin tipi şudur:")
print(type(sayı))
```

Bu örnekten anlıyoruz ki, kullanıcıdan gelecek veriyi daha almadan sayıya dönüştürebileceğimiz gibi, önce kullanıcıdan veriyi alıp daha sonra bunu sayıya da dönüştürebiliyoruz...

Kullanıcıdan aldığımız veriyi nasıl sayıya dönüştüreceğimizi öğrendiğimize göre artık en başa dönebiliriz... En yukarıda yazdığımız ve hata verdiğini gördüğümüz kodları tekrar ele alalım:

```
#!/usr/bin/env python3.0

kare = input("Bir sayı gir, ben sana o sayının karesini söyleyeyim:")
print(kare, "sayısının karesi: ", kare**2)
```

Bu kodları bu şekilde çalıştırırsak hata vereceğini biliyoruz artık. O yüzden yukarıdaki kodlarda şu değişikliği yapıyoruz:

```
#!/usr/bin/env python3.0

kare = int(input("Bir sayı gir, ben sana o sayının karesini söyleyeyim:"))
print(kare, "sayısının karesi: ", kare**2)
```

Böylece, Python kullanıcıdan aldığımız veriyi başarıyla sayıya dönüştürüp, bu veri üzerinde aritmetik bir işlem yapabildi. Yalnız, tabii ki kullanıcının yanlışlıkla sayı değerli bir veri yerine sayı değerli olmayan bir veri girmesi durumunda programımız yine yere çakılacaktır... Yani

mesela kullanıcı “12” yazmak isterken yanlışlıkla veya bilerek bir harf girerse programımız beklediğimiz çıktıyı vermeyecektir... Henüz bu tür durumlara karşı önlem almayı bilmiyoruz. Ama zamanı geldiğinde beklenmedik durumlara karşı önlem almayı da öğreneceğiz... Şimdilik kullanıcılarımızın istediğimiz ve beklediğimiz verileri uslu uslu girmesi için dua ediyoruz...

Böylece bu konunun da sonuna geldik... Şimdi isterseniz bu bölümde öğrendiklerimizi şöyle kısaca bir özetleyelim.

Özet

Bu bölümde pek çok yeni şey öğrendik. Bu bölümün en önemli getirisi `input()` fonksiyonunu öğrenmemiz oldu. Bu fonksiyon sayesinde kullanıcıyla iletişim kurmayı başardık. Artık kullanıcıdan veri alıp, bu verileri programlarımız içinde işleyebiliyoruz. Yine bu bölümde dikkatinizi çektiğimiz başka bir konu da sayılar ve karakter dizileri arasındaki ilişkiydi. `input()` fonksiyonuyla elde edilen çıktının bir karakter dizisi olduğunu öğrendik. Bildiğimiz gibi, aritmetik işlemleri ancak sayılar arasında yapılabilir. Dolayısıyla `input()` fonksiyonuyla gelen karakter dizisini bir sayıyla çarpmaya kalkarsak hata alıyoruz. Burada yapmamız gereken şey, elimizdeki verileri dönüştürmek. Yani `input()` fonksiyonundan gelen karakter dizisini bir sayıyla çarpmak istiyorsak, öncelikle aldığımız karakter dizisini sayıya dönüştürmemiz gerekiyor. Dönüştürme işlemleri için kullandığımız fonksiyonlar şunlardı:

int() Sayı değerli bir karakter dizisini veya ondalık sayıyı tamsayıya (integer) çevirir.

float() Sayı değerli bir karakter dizisini veya tamsayıyı ondalık sayıya (float) çevirir.

str() Bir tamsayı veya ondalık sayıyı karakter dizisine (string) çevirir.

Ayrıca bu bölümde şöyle önemli bir tespitte de bulunduk:

Her tamsayı ve/veya ondalık sayı bir karakter dizisine dönüştürülebilir. Ama her karakter dizisi tamsayıya ve/veya ondalık sayıya dönüştürülemez...

Örneğin, 5654 gibi bir tamsayıyı veya 543.34 gibi bir ondalık sayıyı `str()` fonksiyonu yardımıyla karakter dizisine dönüştürebiliriz:

```
>>> str(5654)
>>> str(543.34)
```

“5654” veya “543.34” gibi bir karakter dizisini `int()` veya `float()` fonksiyonu yardımıyla tamsayıya ya da ondalık sayıya da dönüştürebiliriz:

```
>>> int("5654")
>>> int("543.34")

>>> float("5654")
>>> float("543.34")
```

Ama “elma” gibi bir karakter dizisini ne `int()` ne de `float()` fonksiyonuyla tamsayıya veya ondalık sayıya dönüştürebiliriz!

Bu bilgileri, önemlerinden ötürü aklımızda tutmaya çalışalım. Buraya kadar anlatılan konular hakkında zihnimizde belirsizlikler varsa veya bazı noktaları tam olarak kavrayamadıysak, şimdiye kadar öğrendiğimiz konuları tekrar gözden geçirmemiz bizim için epey faydalı olacaktır. Zira bundan sonraki bölümlerde, yeni bilgilerin yanısıra, buraya kadar öğrendiğimiz şeyleri de yoğun bir şekilde pratiğe dökeceğiz. Bundan sonraki konuları takip edebilmemiz açısından, buraya kadar verdiğimiz temel bilgileri iyice sindirmiş olmak işimizi bir hayli kolaylaştıracaktır.

Önemli bir konuyu daha geride bıraktığımıza göre artık yeni bir bölüme geçebiliriz...

BÖLÜM 2

Python'da Koşula Bağlı Durumlar

2.1 Giriş

Bu bölümde yine çok önemli bir konuya değineceğiz: Python'da koşula bağlı durumlar.

Bu bölüm sonunda, programlarımıza "karar vermeyi" öğretmiş olacağız. Bu bölümde öğreneceğimiz bilgileri kullanarak, yazdığımız bir programın, kullanıcıdan gelen verinin niteliğine göre farklı işlemler yapabilmesini sağlayacağız.

Python programlama dilinde, koşullu durumları belirtmek için üç adet deyimden yararlanıyoruz:

- if deyimi
- elif deyimi
- else deyimi

Yukarıdaki deyimler, bir önceki bölümde gördüğümüz, kullanıcıdan veri almamızı sağlayan `input()` fonksiyonuyla kol kola gider. Bu fonksiyon ve yukarıdaki deyimler esasında bir program içinde birbirini tamamlayan unsurlar olarak karşımıza çıkar.

Gelin isterseniz bir an önce bu önemli konuya giriş yapalım.

2.2 if deyimi

Eğer daha önceden herhangi bir programlama dilini az da olsa kurcalama fırsatınız olduysa, bir programlama dilinde `if` deyimlerinin ne işe yaradığını az çok biliyorsunuzdur. Daha önceden hiç bir programcılık deneyiminiz olmamışsa da ziyanı yok. Zira bu bölümde `if` deyimlerinin ne işe yaradığını ve nerelerde kullanıldığını enine boyuna tartışacağız.

İngilizce bir kelime olan "if", Türkçe'de "eğer" anlamına gelir. Anlamından da çıkarabileceğimiz gibi, bu kelime bir koşul bildiriyor. Yani "eğer bir şey filanca ise..." ya da "eğer bir şey falanca ise..." gibi... İşte biz Python'da bir koşula bağlamak istediğimiz durumları `if` deyimi aracılığıyla göstereceğiz.

Gelin isterseniz bu deyimi nasıl kullanacağımıza dair ufacık bir örnek vererek işe başlayalım:

Öncelikle elimizde şöyle bir değişken olsun:

```
n = 255
```

Yukarıda verdiğimiz değişkenin değerinin bir karakter dizisi değil, aksine bir sayı olduğunu görüyoruz. Şimdi bu değişkenin değerini sorgulayalım:

```
if n > 10:
```

Burada sayının 10'dan büyük olup olmadığına bakıyoruz.

Burada gördüğümüz ">" işaretinin ne demek olduğunu açıklamaya gerek yok sanırım. Hepimizin bildiği "büyüktür" işareti Python'da da aynen bildiğimiz şekilde kullanılıyor. Mesela "küçüktür" demek isteseydik, "<" işaretini kullanacaktık. İsterseniz hemen şurada araya girip bu işaretleri yeniden hatırlayalım:

> büyüktür

< küçüktür

>= büyük eşittir

<= küçük eşittir

== eşittir

!= eşit değildir

Gördüğünüz gibi hiçbiri bize yabancı gelecek gibi değil... Yalnızca en sondaki "eşittir" (==) işareti ve "eşit değildir" (!=) işaretleri biraz değişik gelmiş olabilir. Burada "eşittir" işaretinin "=" olmadığına dikkat edin. Python'da "=" işaretini değer atama işlemleri için kullanıyoruz (*a = 25* gibi...). "==" işaretini ise "eşittir" anlamında... Neyse konuyu dağıtmayalım... Ne diyorduk?

```
if n > 10:
```

Bu ifadeyle Python'a şöyle bir şey demiş oluyoruz:

> "Eğer sayının değeri 10'dan büyükse..."

Burada kullandığımız işaretlere dikkat edin. En sonda bir adet ":" işaretinin olduğunu gözden kaçırmıyoruz... Bu tür işaretler Python için çok önemlidir. Bunları yazmayı unutursak Python gözümüzün yaşına bakmadan bize hata mesajı gösterecektir.

Dedik ki, "*if n > 10:*" ifadesi, "*eğer bir sayının değeri 10'dan büyükse...*" anlamına gelir. Bu ifadenin eksik olduğu apaçık ortada. Yani belli ki bu cümlenin bir de devamı olması gerekiyor. O halde biz de devamını getirelim:

```
if n > 10:
    print("sayı 10'dan büyüktür!")
```

Burada çok önemli bir durumla karşı karşıyayız. Dikkat ederseniz, ikinci satırı ilk satıra göre girintili yazdık. Elbette bunu şirinlik olsun diye yapmadık. Python programlama dilinde girintiler çok büyük önem taşır. Hatta ne kadarlık bir girinti verdiğiniz bile önemlidir. Eğer Python kodlarına duyarlı bir metin düzenleyici kullanıyorsanız, kullandığınız metin düzenleyici çoğu durumda sizin yerinize uygun bir şekilde girintilemeyi yapacaktır. Mesela IDLE adlı geliştirme ortamını kullananlar, ilk satırdaki ":" işaretini koyup "enter" tuşuna bastıklarında otomatik olarak girinti verildiğini farkedeceklerdir. Eğer kullandığınız metin düzenleyici, satırları otomatik olarak girintilemiyorsa sizin bu girintileme işlemini elle yapmanız gerekecektir. Yalnız elle girintilerken, ne kadar girinti vereceğimize dikkat etmeliyiz. Genel kural olarak 4 boşlukluk bir girintileme uygun olacaktır. Girintileme işlemini klavyedeki sekme (tab) tuşuna basarak da yapabilirsiniz. Ama aynı program içinde sekmelerle boşlukları karıştırmayın. Yani eğer girintileme işlemini klavyedeki boşluk (space) tuşuna basarak yapıyorsanız, program boyunca aynı şekilde

yapın. Kısaca söylemek gerekirse; Python'da girintileme ve girintilemede tutarlılık çok önemlidir. Özellikle büyük programlarda, girintilemeler açısından tutarsızlık gösterilmesi programın çalışmamasına sebep olabilir... Neyse... Konumuza devam edelim.

Eğer yukarıdaki `if` bloğunu bir metin düzenleyici içine değil de doğrudan etkileşimli kabuğa yazmışsanız bazı şeyler dikkatinizi çekmiş olmalı... Etkileşimli kabukta "*if sayı > 10:*" satırını yazıp enter tuşuna bastığınızda şöyle bir görüntüyle karşılaşmış olmalısınız:

```
>>> if n > 10:
...
```

Dikkat ederseniz, ">>>" işareti, "..." işaretine dönüştü. Eğer bu noktada herhangi bir şey yazmadan enter tuşuna basacak olursanız Python size şöyle bir hata mesajı verecektir:

```
File "<stdin>", line 2
    ^
IndentationError: expected an indented block
```

Hata mesajında da söylendiği gibi, Python bizden girintilenmiş bir blok beklerken, biz onun bu beklentisini karşılamamışız. Dolayısıyla bize yukarıdaki hata mesajını göstermiş. "..." işaretini gördükten sonra yapmamız gereken şey, dört kez boşluk (space) tuşuna basarak girinti oluşturmak ve `if` bloğunun devamını yazmak olmalıydı. Yani şöyle:

```
>>> if n > 10:
...     print("sayı 10'dan büyüktür!")
...
```

Gördüğünüz gibi, `print()` fonksiyonunu yazıp enter'e bastıktan sonra yine "..." işaretini gördük. Python burada bizden yeni bir satır daha bekliyor. Ama bizim yazacak başka bir kodumuz olmadığı için tekrar enter tuşuna basıyoruz ve nihai olarak şöyle bir görüntü elde ediyoruz:

```
>>> if n > 10:
...     print("sayı 10'dan büyüktür!")
...
sayı 10'dan büyüktür!
>>>
```

Demek ki 250 sayısı 10'dan büyükmüş! Ne büyük bir buluş! Merak etmeyin, daha çok şey öğrendikçe daha mantıklı programlar yazacağız... Burada amacımız işin temelini kapmak. Bunu da en iyi, saçma sapan ama basit programlar yazarak yapabiliriz.

Şimdi metin düzenleyicimizi açarak daha mantıklı şeyler yazmaya çalışalım. Zira yukarıdaki örnekte değişkeni kendimiz belirlediğimiz için, bu değişkenin değerini `if` deyimleri yardımıyla denetlemek pek akla yatkın görünmüyor. Ne de olsa değişkenin değerinin ne olduğunu biliyoruz. Dolayısıyla bu değişkenin 10 sayısından büyük olduğunu da biliyoruz! Bunu `if` deyimiyle kontrol etmek çok gerekli değil... Ama şimdi daha makul bir iş yapacağız. Değişkeni biz belirlemek yerine kullanıcıya belirleteceğiz:

```
#!/usr/bin/env python3.0

sayı = int(input("Bir sayı giriniz: "))

if sayı > 10:
    print("Girdiğiniz sayı 10'dan büyüktür!")

if sayı < 10:
    print("Girdiğiniz sayı 10'dan küçüktür!")
```

```
if sayı == 10:
    print("Girdiğiniz sayı 10'dur!")
```

Gördüğünüz gibi, art arda üç adet `if` bloğu kullandık. Bu kodlara göre, eğer kullanıcının girdiği sayı 10'dan büyükse, ilk `if` bloğu işletilecek; eğer sayı 10'dan küçükse ikinci `if` bloğu işletilecek; eğer sayı 10'a eşit ise üçüncü `if` bloğu işletilecektir... Peki ya kullanıcı muziplik yapıp sayı yerine harf yazarsa ne olacak? Böyle bir ihtimal için programımıza herhangi bir denetleyici yerleştirmedik. Dolayısıyla eğer kullanıcı sayı yerine harf girerse programımız hata verecek, yani çökecektir. Bu tür durumlara karşı nasıl önlem alacağımızı ilerleyen derslerimizde göreceğiz. Biz şimdilik bildiğimiz yolda yürüyelim...

`if` deyimlerini kullanıcı adı veya parola denetlerken de kullanabiliriz. Mesela şöyle bir program taslağı yazabiliriz:

```
#!/usr/bin/env python3.0

print("""
Dünyanın en gelişmiş e.posta hizmetine
hoşgeldiniz... Yalnız hizmetimizden
yararlanmak için önce sisteme giriş
yapmalısınız..
""")

parola = input("Parola: ")

if parola == "bilmiyorum":
    print("Sisteme Hoşgeldiniz!")
```

Gördüğünüz gibi, programın başında üç tırnak işaretlerinden yararlanarak uzun bir metni kullanıcıya gösterdik. Bu bölümü, kendiniz göze hoş gelecek bir şekilde süsleyebilirsiniz de... Eğer kullanıcı, kendisine parola sorulduğunda cevap olarak "bilmiyorum" yazarsa kullanıcıyı sisteme alıyoruz.

Daha önce de söylediğimiz gibi, `if` deyimi dışında Python'da koşullu durumları belirtmek için kullandığımız, `elif` ve `else` adlı iki deyim daha vardır. Bunlar `if` ile birlikte kullanılırlar. Gelin isterseniz bu iki deyimden, adı `elif` olana bakalım...

2.3 elif deyimi

Python'da, `if` deyimleriyle birlikte kullanılan ve yine koşul belirten bir başka deyim de `elif` deyimidir. Buna şöyle bir örnek verebiliriz:

```
#!/usr/bin/env python3.0

yaş = int(input("Yaşınız: "))

if yaş == 18:
    print("18 iyidir!")

elif yaş < 0:
    print("Yok canım, daha neler!...")

elif yaş < 18:
    print("Genç bir kardeşimizsin!")
```

```
elif yaş > 18:
    print("Eh, artık yaş yavaş yavaş kemale eriyor!")
```

Yukarıdaki örneği elbette şöyle de yazabiliriz:

```
#!/usr/bin/env python3.0

yaş = int(input("Yaşınız: "))

if yaş == 18:
    print("18 iyidir!")

if yaş < 0:
    print("Yok canım, daha neler!...")

if yaş < 18:
    print("Genç bir kardeşimizsin!")

if yaş > 18:
    print("Eh, artık yaş yavaş yavaş kemale eriyor!")
```

Bu iki program da aynı işlevi görecektir. O halde şöyle bir soru geliyor akla: `if` varken `elif`'e ne lüzum var? İlk bakışta pek belli olmasa da `if` ile `elif` arasında çok önemli bir fark vardır. `if` bize olası bütün sonuçları listeler, `elif` ise sadece doğru olan ilk sonucu verir. Bu soyut tanımlamayı biraz somutlaştıralım:

```
#!/usr/bin/env python3.0

a = int(input("Bir sayı giriniz: "))

if a < 100:
    print("verdiğiniz sayı 100'den küçüktür.")

if a < 50:
    print("verdiğiniz sayı 50'den küçüktür.")

if a == 100:
    print("verdiğiniz sayı 100'dür.")

if a > 100:
    print("verdiğiniz sayı 100'den büyüktür.")

if a > 150:
    print("verdiğiniz sayı 150'den büyüktür.")
```

Yukarıdaki kodları çalıştırdığımızda, doğru olan bütün sonuçlar listelenecektir. Yani mesela kullanıcı 40 sayısını girmişse, ekrana verilecek çıktı şöyle olacaktır:

```
verdiğiniz sayı 100'den küçüktür.
verdiğiniz sayı 50'den küçüktür.
```

Burada "40" sayısı hem "100"den, hem de "50"den küçük olduğu için iki sonuç da çıktı olarak verilecektir. Ama eğer yukarıdaki kodları şöyle yazarsak:

```
#!/usr/bin/env python3.0

a = int(input("Bir sayı giriniz: "))
```

```
if a < 100:
    print("verdiğiniz sayı 100'den küçüktür.")

elif a < 50:
    print("verdiğiniz sayı 50'den küçüktür.")

elif a == 100:
    print("verdiğiniz sayı 100'dür.")

elif a > 150:
    print("verdiğiniz sayı 150'den büyüktür.")

elif a > 100:
    print("verdiğiniz sayı 100'den büyüktür.")
```

Kullanıcının "40" sayısını girdiğini varsayarsak, bu defa programımımız yalnızca şu çıktıyı verecektir:

```
verdiğiniz sayı 100'den küçüktür.
```

Gördüğünüz gibi, `elif` deyimlerini kullandığımız zaman, ekrana yalnızca doğru olan ilk sonuç veriliyor. Yukarıda "40" sayısı hem 100'den hem de 50'den küçük olduğu halde, Python bu sayının 100'den küçük olduğunu görür görmez sonucu ekrana basıp, kodların geri kalanını incelemeyi bırakıyor. `if` deyimlerini arka arkaya sıraladığımızda ise, Python bütün olasılıkları tek tek değerlendirip, geçerli olan bütün sonuçları ekrana döküyor...

Bir sonraki bölümde `else` deyimini öğrendiğimiz zaman, `elif`'in tam olarak ne işe yaradığını çok daha iyi anlamanızı sağlayacağını düşündüğüm bir örnek vereceğim.

`elif`'i de incelediğimize göre, koşul bildiren deyimlerin sonuncusuna göz atabiliriz: `else`

2.4 else deyimi

Şimdiye kadar Python'da koşul bildiren iki deyimi öğrendik. Bunlar "if" ve "elif" idi. Bu bölümde ise koşul deyimlerinin sonuncusu olan "else"yi göreceğiz. Öğrendiğimiz şeyleri şöyle bir gözden geçirecek olursak, temel olarak şöyle bir durumla karşı karşıya olduğumuzu görürüz:

```
if falanca:
    bu işlemi yap

if filanca:
    şu işlemi yap
```

Veya şöyle bir durum:

```
if falanca:
    bu işlemi yap

elif filanca:
    şu işlemi yap
```

`if` ile `elif` arasındaki farkı biliyoruz. Eğer `if` deyimlerini art arda sıralayacak olursak, Python doğru olan bütün sonuçları listeleyecektir. Ama eğer `if` deyiminden sonra `elif` deyimini kullanırsak, Python doğru olan ilk sonucu listelemekle yetinecektir.

Bu bölümde göreceğimiz `else` deyimi, yukarıdaki tabloya bambaşka bir boyut kazandırıyor. Dikkat ederseniz şimdiye kadar öğrendiğimiz deyimleri kullanabilmek için ilgili bütün durumları tanımlamamız gerekiyor. Yani:

```
eğer böyle bir durum varsa:
    bunu yap

eğer şöyle bir durum varsa:
    şunu yap

eğer filancaysa:
    şöyle git

eğer falancaysa:
    böyle gel
```

gibi...

Ancak her durum için bir `if` bloğu yazmak bir süre sonra yorucu ve sıkıcı olacaktır. İşte bu noktada devreye `else` deyimi girecek. `else`'nin anlamı kabaca şudur:

> *"eğer yukarıdaki koşulların hiçbiri gerçekleşmezse..."*

Gelin isterseniz bununla ilgili şöyle bir örnek verelim:

```
#!/usr/bin/env python3.0

soru = input("Bir meyve adı söyleyin bana:")

if soru == "elma":
    print("evet, elma bir meyvedir...")

elif soru == "karpuz":
    print("evet, karpuz bir meyvedir...")

elif soru == "armut":
    print("evet, armut bir meyvedir...")

else:
    print(soru, "gerçekten bir meyve midir?")
```

Eğer kullanıcı soruya "elma", "karpuz" veya "armut" cevabı verirse, "evet, ... bir meyvedir" çıktısı verilecektir. Ama eğer kullanıcı bu üçü dışında bir cevap verirse, "... gerçekten bir meyve midir?" çıktısını görürüz... Burada `else` deyimi, programımıza şu anlamı katıyor:

> *"Eğer kullanıcı yukarıda belirlenen meyve adlarından hiç birini girmez, bunların yerine bambaşka bir şey yazarsa, o zaman ''else'' bloğu içinde belirtilen işlemi gerçekleştir."*

Dikkat ederseniz yukarıdaki kodlarda `if` deyimlerini art arda sıralamak yerine ilk `if`'ten sonra `elif` ile devam ettik. Peki şöyle bir şey yazarsak ne olur?

```
#!/usr/bin/env python3.0

soru = input("Bir meyve adı söyleyin bana:")

if soru == "elma":
    print("evet, elma bir meyvedir...")

if soru == "karpuz":
```

```
    print("evet, karpuz bir meyvedir...")

if soru == "armut":
    print("evet, armut bir meyvedir...")

else:
    print(soru, "gerçekten bir meyve midir")
```

Bu kodlar beklediğiniz sonucu vermeyecektir. İsterseniz yukarıdaki kodları çalıştırıp ne demek istediğimi daha iyi anlayabilirsiniz. Python'un doğru sonucu vermesi için bu tür durumlarda `else` deyiminden önce `elif` deyimlerinden yararlanmalıyız...

Yalnız bu dediğimizden, `else` ifadesi `if` ile birlikte kullanılmaz, anlamı çıkarılmamalı. Mesela şöyle bir örnek yapılabilir:

```
#!/usr/bin/env python3.0

soru = input("Programdan çıkmak istediğinize emin misiniz? \
Eminseniz 'e' harfine basın : ")

if soru == "e":
    print("Güle güle!")

else:
    print("Peki, biraz daha sohbet edelim!")
```

Bir önceki bölümde `elif` deyiminin tam olarak ne işe yaradığını anlamamızı sağlayacak bir örnek vereceğimizi söylemiştik. Şimdi bu örneğe bakalım:

```
#!/usr/bin/env python3.0

boy = int(input("boyunuz kaç cm?"))

if boy < 170:
    print("boyunuz kısa")

elif boy < 180:
    print("boyunuz normal")

else:
    print("boyunuz uzun")
```

Yukarıda sekiz satırla hallettiğimiz işi sadece `if` deyimleriyle yapmaya çalışırsanız bunun ne kadar zor olduğunu göreceksiniz. Diyelim ki kullanıcı "165" cevabını verdi. Python bu 165 sayısının 170'ten küçük olduğunu görünce "boyunuz kısa" cevabını verecek, öteki satırları değerlendirmeyecektir. 165 sayısı, `elif` ile gösterdiğimiz koşullu duruma da uygun olduğu halde (*165 < 180*), koşul ilk satırda karşılandığı için ikinci satır değerlendirmeye alınmayacaktır.

Kullanıcının "175" cevabını verdiğini varsayalım: Python 175 sayısını görünce önce ilk koşula bakacak, verilen 175 sayısının ilk koşulu karşılamadığını görecektir (*175 > 170*). Bunun üzerine Python kodları incelemeye devam edecek ve `elif` bloğunu değerlendirmeye alacaktır. 175 sayısının 180'den küçük olduğunu görünce de çıktı olarak "boyunuz normal" cevabını verecektir.

Peki ya kullanıcı "190" cevabını verirse ne olacak? Python yine önce ilk `if` bloğuna bakacak ve 190 cevabının bu bloğa uymadığını görecektir. Dolayısıyla ilk bloğu bırakıp ikinci bloğa bakacaktır. 190 cevabının bu bloğa da uymadığını görünce, bir sonraki bloğu değerlendirmeye

alacaktır. Bir sonraki blokta ise `else` deyimimiz var. Bu bölümde öğrendiğimiz gibi, `else` deyimi, "eğer kullanıcının cevabı yukarıdaki koşulların hiçbirine uymazsa bu bloğu çalıştır," anlamına geliyor. Kullanıcının girdiği "190" cevabı ne birinci ne de ikinci bloktaki koşula uyduğu için, normal bir şekilde `else` bloğu işletilecek, dolayısıyla da ekrana "boyunuz uzun" çıktısı verilecektir.

Python'da koşullu durumları nasıl göstereceğimizi öğrendiğimize göre, bu öğrendiklerimiz yardımıyla küçük bir alıştırma yapabiliriz. Bir sonraki yazımızın konusu, Python'da basit bir hesap makinesi nasıl yapılır?

2.5 Basit bir Hesap Makinesi

Şu ana kadar Python'da pek çok şey öğrendik. Bu öğrendiğimiz şeylerle artık kısmen yararlı bazı programlar yazabiliriz. Elbette henüz yazacağımız programlar pek yetenekli olamayacak olsa da, en azından bize öğrendiklerimizle pratik yapma imkanı sağlayacak... Bu bölümde, özellikle `if`, `elif` ve `else` yapılarını kullanarak çok basit bir hesap makinesi yapmayı deneyeceğiz. Bu arada, bu derste yeni şeyler öğrenerek ufkumuzu ve bilgimizi genişletmeyi de ihmal etmeyeceğiz.

İsterseniz önce kullanıcıya bazı seçenekler sunarak işe başlayalım:

```
#!/usr/bin/env python3.0

giriş = """
(1) topla
(2) çıkar
(3) çarp
(4) böl
(5) karesini hesapla
"""

print(giriş)
```

İlk satırın sadece GNU/Linux kullanıcılarını ilgilendirdiğini biliyoruz artık. Bu satırı Windows kullanıcıları yazmasa da olur...

İlk satırın ardından kullanıcıya bazı seçenekler sunuyoruz. Bu seçenekleri ekrana yazdırmak için üç tırnaktan yararlandığımıza dikkat edin. Birden fazla satıra yayılmış bu tür ifadeleri en kolay üç tırnak yardımıyla yazdırabiliriz. Esasında her bir seçenek için ayrı bir değişken tanımlamak da mümkündür. Yani aslında yukarıdaki kodları şöyle de yazabiliriz:

```
#!/usr/bin/env python3.0

seçenek1 = "(1) topla"
seçenek2 = "(2) çıkar"
seçenek3 = "(3) çarp"
seçenek4 = "(4) böl"
seçenek5 = "(5) karesini hesapla"

print(seçenek1, seçenek2, seçenek3, seçenek4, seçenek5)
```

Yalnız burada dikkat ederseniz, seçenekler hep yan yana diziliyor. Eğer programınızda yukarıdaki şekli kullanmak isterseniz, bu seçeneklerin yan yana değil de, alt alta görünmesini sağlamak için Python'un bize sunduğu çok faydalı bir parçacıktan yararlanabilirsiniz. Bu parçacığın adı "sep"tir. "sep", İngilizce "separator" (ayraç) kelimesinin kısaltmasıdır. Kullanımı tıpkı daha önce öğrendiğimiz "end" parçacığına benzer:

```
#!/usr/bin/env python3.0

seçenek1 = "(1) topla"
seçenek2 = "(2) çıkar"
seçenek3 = "(3) çarp"
seçenek4 = "(4) böl"
seçenek5 = "(5) karesini hesapla"

print(seçenek1, seçenek2, seçenek3, seçenek4, seçenek5, sep="\n")
```

Burada “sep” parçacığının değeri olarak “\n” kaçış dizisini belirlediğimize dikkat edin. “\n” kaçış dizisinin ne işe yaradığını hatırlıyorsunuz. Bu dizi, “yeni satır” oluşturmamızı sağlıyordu... Burada, “ayraç” olarak “yeni satır” kaçış dizisini belirlediğimiz için her bir seçenek yan yana değil, alt alta görünecektir. Elbette “sep” parçacığı için istediğiniz değeri belirleyebilirsiniz. Mesela her bir seçeneği “yeni satır işaretiyle” ayırmak yerine, “–” gibi bir işaretle ayırmayı da tercih edebilirsiniz:

```
print(seçenek1, seçenek2, seçenek3, seçenek4, seçenek5, sep="--")
```

Programınızda nasıl bir giriş paragrafı belirleyeceğiniz konusunda özgürsünüz. Gelin isterseniz biz birinci şekille yolumuza devam edelim:

```
#!/usr/bin/env python3.0

giriş = """
(1) topla
(2) çıkar
(3) çarp
(4) böl
(5) karesini hesapla
"""

print(giriş)
```

Burada “giriş” adlı bir değişken oluşturduk. Bu değişkenin içinde barındırdığı değeri kullanıcıların görebilmesi için `print()` fonksiyonu yardımıyla bu değişkeni ekrana yazdırıyoruz. Devam edelim:

```
soru = input("Yapmak istediğiniz işlemin numarasını girin: ")
```

Bu kod yardımıyla kullanıcıya bir soru soruyoruz. Kullanıcıdan yapmasını istediğimiz şey, yukarıda belirlediğimiz giriş seçenekleri içinden bir sayı seçmesi... Kullanıcı “1”, “2”, “3”, “4” veya “5” seçeneklerinden herhangi birini seçebilir. Kullanıcıyı, seçtiği numaranın karşısında yazan işleme yönlendireceğiz. Yani mesela eğer kullanıcı klavyedeki “1” tuşuna basarsa hesap makinemiz toplama işlemi yapacaktır... “2” tuşu ise kullanıcıyı “çıkarma” işlemine yönlendirir...

`input()` fonksiyonunu işlediğimiz bölümde, bu fonksiyonun değer olarak bir “karakter dizisi” (string) verdiğini söylemiştik. Yukarıdaki kodun çıktısı da doğal olarak bir karakter dizisi olacaktır. Bizim şu aşamada kullanıcıdan karakter dizisi almamızın bir sakıncası yok. Çünkü kullanıcının gireceği “1”, “2”, “3”, “4” veya “5” değerleriyle herhangi bir matematik işlemi yapmayacağız. Kullanıcının gireceği bu değerler, yalnızca bize onun hangi matematik işlemini yapmak istediğini belirtecek. Dolayısıyla `input()` fonksiyonunu yukarıdaki şekilde kullanıyoruz...

İsterseniz şimdiye kadar gördüğümüz kısma topluca bakalım:

```
#!/usr/bin/env python3.0
```

```
giriş = """
(1) topla
(2) çıkar
(3) çarp
(4) böl
(5) karesini hesapla
"""

print(giriş)

soru = input("Yapmak istediğiniz işlemin numarasını girin: ")
```

Bu kodları çalıştırdığımızda, ekranda giriş paragrafımız görünecek ve kullanıcıya, yapmak istediği işlemin ne olduğu sorulacaktır. Henüz kodlarımız eksik olduğu için, kullanıcı hangi sayıyı girerse girsin, programımız hiç bir iş yapmadan kapanacaktır. O halde yolumuza devam edelim:

```
if soru == "1":
```

Böylece ilk `if` deyimimizi tanımlamış olduk. Buradaki yazım şekline çok dikkat edin. Bu kodlarla Python'a şu emri vermiş oluyoruz:

> *"Eğer "soru" adlı değişkenin değeri "1" ise, yani eğer kullanıcı klavyede "1" tuşuna basarsa..."*

`if` deyimlerinin en sonuna ":" işaretini koymayı unutmuyoruz. Python'a yeni başlayanların en çok yaptığı hatalardan birisi, sondaki bu ":" işaretini koymayı unutmalarıdır. Bu işaret bize çok ufak bir ayrıntıymış gibi görünse de Python için manevi değeri çok büyüktür! Python'un bize öfkeli mesajlar göstermesini istemiyorsak bu işareti koymayı unutmayacağız. Bu arada, burada "==" işaretini kullandığımıza da dikkat edin. Bunun ne anlama geldiğini önceki derslerimizde öğrenmiştik. Bu işaret, iki şeyin aynı değere sahip olup olmadığını karşılaştırmamızı sağlıyor. Biz burada "1"in değeri ile "soru" adlı değişkenin değerinin aynı olup olmadığını sorguladık. "1"in değeri tabii ki "1"dir, ama "soru" değişkeninin değeri kullanıcı tarafından belirleneceği için henüz bu değişkenin değerinin ne olduğunu bilmiyoruz. Bizim programımızda kullanıcı klavyeden "1", "2", "3", "4" veya "5" değerlerinden herhangi birini seçebilir... Biz yukarıdaki kod yardımıyla, eğer kullanıcı klavyede "1" tuşuna basarsa ne yapılacağını belirleyeceğiz. O halde devam edelim:

```
if soru == "1":
    sayı1 = int(input("Toplama işlemi için ilk sayıyı girin: "))
    sayı2 = int(input("Toplama işlemi için ikinci sayıyı girin: "))
    print(sayı1, "+", sayı2, "=", sayı1 + sayı2)
```

Böylece ilk `if` bloğumuzu tanımlamış olduk.

`if` deyimimizi yazdıktan sonra ne yaptığımız çok önemli. Buradaki girintileri, programımız güzel görünsün diye yapmıyoruz. Bu girintilerin Python için bir anlamı var. Eğer bu girintileri vermezsek programımız çalışmayacaktır. Eğer Python kodlarına duyarlı bir metin düzenleyici kullanıyorsanız, ":" işaretini koyup "enter"e bastıktan sonra otomatik olarak girinti verilecektir. Eğer kullandığınız metin düzenleyici size böyle bir güzellik sunmuyorsa "enter"e bastıktan sonra klavyedeki boşluk (space) tuşunu kullanarak dört vuruşluk bir girinti oluşturabilirsiniz. Bu girintiler, ilk satırda belirlediğimiz `if` deyimiyle gösterilecek işlemlere işaret ediyor. Dolayısıyla burada yazılan kodları Pythonca'dan Türkçe'ye çevirecek olursak şöyle bir şey elde ederiz:

```
eğer sorunun değeri "1" ise:
    Toplama işlemi için ilk sayı girilsin ve biz bu değere "sayı1" diyelim
    Ardından toplama işlemi için ikinci sayı girilsin ve biz bu değere
    "sayı2" diyelim
```

```
    Son olarak, "sayı1", "+" işareti, "=" işareti, "sayı2" ve "sayı1 + sayı2"
    ekrana yazdırılsın...
```

Gelin isterseniz buraya kadar olan bölümü yine topluca görelim:

```
#!/usr/bin/env python3.0

giriş = """
(1) topla
(2) çıkar
(3) çarp
(4) böl
(5) karesini hesapla
"""

print(giriş)

soru = input("Yapmak istediğiniz işlemin numarasını girin: ")

if soru == "1":
    sayı1 = int(input("Toplama işlemi için ilk sayıyı girin: "))
    sayı2 = int(input("Toplama işlemi için ikinci sayıyı girin: "))
    print(sayı1, "+", sayı2, "=", sayı1 + sayı2)
```

Bu kodları çalıştırıp, klavyede “1” tuşuna bastığımızda, bizden bir sayı girmemiz istenecektir. İlk sayımızı girdikten sonra bize tekrar bir sayı girmemiz söylenecek. Bu emre de uyup “enter” tuşuna basınca, girdiğimiz bu iki sayının toplandığını göreceğiz... Fena sayılmaz, değil mi?

Şimdi programımızın geri kalan kısmını yazıyoruz. İşin temelini kavradığımıza göre birden fazla kod bloğunu aynı anda yazabiliriz:

```
elif soru == "2":
    sayı3 = int(input("Çıkarma işlemi için ilk sayıyı girin: "))
    sayı4 = int(input("Çıkarma işlemi için ikinci sayıyı girin: "))
    print(sayı3, "-", sayı4, "=", sayı3 - sayı4)

elif soru == "3":
    sayı5 = int(input("Çarpma işlemi için ilk sayıyı girin: "))
    sayı6 = int(input("Çarpma işlemi için ikinci sayıyı girin: "))
    print(sayı5, "x", sayı6, "=", sayı5 * sayı6)

elif soru == "4":
    sayı7 = int(input("Bölme işlemi için ilk sayıyı girin: "))
    sayı8 = int(input("Bölme işlemi için ikinci sayıyı girin: "))
    print(sayı7, "/", sayı8, "=", sayı7 / sayı8)

elif soru == "5":
    sayı9 = int(input("Karesini hesaplamak istediğiniz sayıyı giriniz: "))
    print(sayı9, "sayısının karesi =", sayı9 ** 2)
```

Bunlarla birlikte kodlarımızın büyük bölümünü tamamlamış oluyoruz. Bu bölümdeki tek fark, ilk `if` bloğunun aksine, burada `elif` bloklarını kullanmış olmamız... Eğer burada bütün blokları `if` kullanarak yazarsanız, biraz sonra kullanacağımız `else` bloğu etkisiz kalacaktır. Yukarıdaki kodlarda az da olsa farklılık gösteren tek yer en son `elif` bloğumuz. Esasında buradaki fark da pek büyük bir fark sayılmaz. Neticede tek bir sayının karesini hesaplayacağımız için, kullanıcıdan yalnızca tek bir giriş istiyoruz.

Şimdi de son bloğumuzu yazalım. Az evvel çıtlattığımız gibi, bu son blok bir `else` bloğu olacak:

```
else:
    print("Yanlış giriş.")
    print("Aşağıdaki seçeneklerden birini giriniz:", giriş)
```

Çok basit bir `else` bloğu ile işimizi bitirdik. Bu bloğun ne işe yaradığını biliyorsunuz. "Eğer kullanıcının girdiği değer yukarıdaki bloklardan hiç birine uymuyorsa bu else bloğunu işlet," gibi bir komut vermiş oluyoruz bu `else` bloğu yardımıyla... Mesela kullanıcımız "1", "2", "3", "4" veya "5" seçeneklerini girmek yerine "6" yazarsa, bu blok işletilecek...

Gelin isterseniz son kez kodlarımızı topluca bir görelim:

```
#!/usr/bin/env python3.0

giriş = """
(1) topla
(2) çıkar
(3) çarp
(4) böl
(5) karesini hesapla
"""

print(giriş)

soru = input("Yapmak istediğiniz işlemin numarasını girin: ")

if soru == "1":
    sayı1 = int(input("Toplama işlemi için ilk sayıyı girin: "))
    sayı2 = int(input("Toplama işlemi için ikinci sayıyı girin: "))
    print(sayı1, "+", sayı2, "=", sayı1 + sayı2)

elif soru == "2":
    sayı3 = int(input("Çıkarma işlemi için ilk sayıyı girin: "))
    sayı4 = int(input("Çıkarma işlemi için ikinci sayıyı girin: "))
    print(sayı3, "-", sayı4, "=", sayı3 - sayı4)

elif soru == "3":
    sayı5 = int(input("Çarpma işlemi için ilk sayıyı girin: "))
    sayı6 = int(input("Çarpma işlemi için ikinci sayıyı girin: "))
    print(sayı5, "x", sayı6, "=", sayı5 * sayı6)

elif soru == "4":
    sayı7 = int(input("Bölme işlemi için ilk sayıyı girin: "))
    sayı8 = int(input("Bölme işlemi için ikinci sayıyı girin: "))
    print(sayı7, "/", sayı8, "=", sayı7 / sayı8)

elif soru == "5":
    sayı9 = int(input("Karesini hesaplamak istediğiniz sayıyı giriniz: "))
    print(sayı9, "sayısının karesi =", sayı9 ** 2)

else:
    print("Yanlış giriş.")
    print("Aşağıdaki seçeneklerden birini giriniz:", giriş)
```

Genel olarak baktığımızda, bütün programın aslında basit bir "if, elif, else" yapısından ibaret olduğunu görüyoruz. Ayrıca bu kodlardaki simetriye de dikkatinizi çekmek isterim. Gördüğünüz gibi her "paragraf" bir `if`, `elif` veya `else` bloğundan oluşuyor ve her blok kendi içinde girintili bir yapı sergiliyor. Temel olarak şöyle bir şeyle karşı karşıyayız:

```
Eğer böyle bir durum varsa:
    şöyle bir işlem yap

Yok eğer şöyle bir durum varsa:
    böyle bir işlem yap

Eğer bambaşka bir durum varsa:
    şöyle bir şey yap
```

Böylelikle şirin bir hesap makinesine sahip olmuş olduk!... Hesap makinemiz pek yetenekli değil, ama olsun... Henüz bildiklerimiz bunu yapmamıza müsaade ediyor. Yine de başlangıçtan bu noktaya kadar epey yol katettiğimizi görüyorsunuz.

Şimdi bu programı çalıştırın ve neler yapabildiğine göz atın. Bu arada kodları da iyice inceleyin. Programı yeterince anladıktan sonra, program üzerinde kendinize göre bazı değişiklikler yapın, yeni özellikler ekleyin. Eksikliklerini, zayıf yönlerini bulmaya çalışın. Böylece bu dersten azami faydayı sağlamış olacaksınız.

Bu arada bu bölümde yazdığımız basit hesap makinesinin Python 2.x için olan bir benzerini zamanında http://www.istihza.com/kosul_elif.html adresinde vermiştim. O kodlarla bu kodları karşılaştırarak, Python 2.x ve 3.x sürümleri arasındaki bazı farkları görebilirsiniz...

BÖLÜM 3

Python'da Döngüler (loops)

Şimdiye kadar Python'la bazı ufak programlar yazabilecek düzeye geldik. Mesela öğrendiğimiz bilgiler yardımıyla bir önceki bölümde çok basit bir hesap makinesi yapabilmiştik. Yalnız o hesap makinesinde farkettiyseniz çok önemli bir eksiklik vardı. Hesap makinemizle hesap yaptıktan sonra programımız kapanıyor, yeni hesap yapabilmek için programı yeniden başlatmamız gerekiyordu...

Mesela bununla ilgili şöyle bir örnek daha verelim:

```
#!/usr/bin/env python3.0

tuttuğum_sayı = 23

bilbakalım = int(input("Aklımdan bir sayı tuttum. Bil bakalım kaç tuttum?? "))

if bilbakalım == tuttuğum_sayı:
    print("Tebrikler! Bildiniz...")

else:
    print("Ne yazık ki tuttuğum sayı bu değildi...")
```

Burada "tuttuğum_sayı" adlı bir değişken belirledik. Bu değişkenin değeri "23". Kullanıcıdan tuttuğumuz sayıyı tahmin etmesini istiyoruz. Eğer kullanıcının verdiği cevap "tuttuğum_sayı" değişkeninin değeriyle aynıysa (yani 23 ise), ekrana "Tebrikler!..." yazısı dökülecektir. Aksi halde (`else`) "Ne yazık ki..." cümlesi ekrana dökülecektir.

Bu program iyi, hoş, ama çok önemli bir eksiği var. Bu programı yalnızca bir kez kullanabiliyoruz. Yani kullanıcı yalnızca bir kez tahminde bulunabiliyor. Eğer kullanıcı bir kez daha tahminde bulunmak isterse programı yeniden çalıştırması gerekecek... Bunun hiç iyi bir yöntem olmadığı ortada. Programımız sürekli olarak başa dönse ne iyi olurdu, değil mi? Yani kullanıcı bir sayı tahmin ettikten sonra, eğer bu sayı bizim tuttuğumuz sayıyla aynı değilse, kullanıcıya tekrar tahmin etme fırsatı verebilsek çok hoş olurdu...

İşte bu bölümde, programlarımızı nasıl "döngü içine sokabileceğimizi", yani nasıl "sürekli olarak çalışmalarını sağlayabileceğimizi" öğreneceğiz.

3.1 while Döngüsü (while loop)

İngilizce bir kelime olan “while”, Türkçe’de ”... iken, ... olduğu sürece” gibi anlamlara gelir. Python’da `while` bir döngüdür. Bir önceki bölümde söylediğimiz gibi, döngüler sayesinde programlarımızın sürekli olarak çalışmasını sağlayabiliriz. Bu dersimizde Python’da `while` döngüsünün ne olduğunu ve ne işe yaradığını öğrenmeye çalışacağız. Öncelikle `while` döngüsünün temellerini kavrayarak işe başlayalım.

Basit bir `while` döngüsü kabaca şuna benzer:

```
#!/usr/bin/env python3.0

a = 1

while a == 1:
```

Burada “a” adlı bir değişken oluşturduk. Bu değişkenin değeri “1”. Bir sonraki satırda ise “*while a == 1:*” gibi bir ifade yazdık. En başta da söylediğimiz gibi “while” kelimesi, “iken, olduğu sürece” gibi anlamlar taşıyor. Python programlama dilindeki anlamı da buna oldukça yakındır. Burada “*while a == 1*” ifadesi programımıza şöyle bir anlam katıyor:

a değişkeninin değeri “1” olduğu sürece...

Gördüğünüz gibi cümlemiz henüz eksik. Yani belli ki bunun bir de devamı olacak. Ayrıca `while` ifadesinin sonundaki ”:” işaretinden anladığımız gibi, bundan sonra gelecek satır girintili yazılacak. Devam edelim:

```
#!/usr/bin/env python3.0

a = 1

while a == 1:
    print("bilgisayar çıldırdı!")
```

Burada Python’a şu emri vermiş olduk:

> *“a değişkeninin değeri “1” olduğu sürece, ekrana “bilgisayar çıldırdı!” yazısını dök!”*

Bu programı çalıştırdığımızda Python verdiğimiz emre sadakatle uyacak ve a değişkeninin değeri “1” olduğu müddetçe de bilgisayarımızın ekranına “bilgisayar çıldırdı!” yazısını dökecektir. Programımızın içinde a değişkeninin değeri “1” olduğu ve bu değişkenin değerini değiştirecek herhangi bir şey bulunmadığı için Python hiç sıkılmadan ekrana “bilgisayar çıldırdı!” yazısını basmaya devam edecektir. Eğer siz durdurmazsanız bu durum sonsuza kadar devam edebilir. Bu çılgınlığa bir son vermek için klavyenizde CTRL+C veya CTRL+Z tuşlarına basarak programı durmaya zorlayabilirsiniz...

Burada programımızı sonsuz bir döngüye sokmuş olduk (infinite loop). Esasında sonsuz döngüler genellikle bir program hatasına işaret eder. Yani çoğu durumda programcının arzu ettiği şey bu değildir. O yüzden doğru yaklaşım, döngüye soktuğumuz programlarımızı durduracak bir ölçüt belirlemektir. Yani öyle bir kod yazmalıyız ki, “a” değişkeninin “1” olan değeri bir noktadan sonra artık “1” olmasın ve böylece o noktaya ulaşıldığında programımız dursun. Kullanıcının CTRL+C tuşlarına basarak programı durdurmak zorunda kalması pek hoş olmuyor... Gelin isterseniz bu soyut ifadeleri biraz somutlaştıralım. Öncelikle şu iki satırı yazarak işe başlıyoruz:

```
#!/usr/bin/env python3.0

a = 1
```

İlk satırı geçelim. İkinci satırda, normal bir şekilde "a" değişkenine "1" değerini atadık. Şimdi devam ediyoruz:

```
#!/usr/bin/env python3.0

a = 1

while a < 10:
```

`while` ile verdiğimiz ilk örnekte "*while a == 1*" gibi bir ifade kullanmıştık. Bu ifade, "a'nın değeri 1 olduğu müddetçe..." gibi bir anlama geliyordu. "*while a < 10*" ifadesi ise, "a'nın değeri 10'dan küçük olduğu müddetçe..." anlamına gelir. İşte burada programımızın "sonsuz döngüye" girmesini engelleyecek bir ölçüt koymuş olduk. Buna göre, "a" değişkeninin şimdiki değeri "1"dir. Biz, "a"nın değeri "10"dan küçük olduğu müddetçe bir işlem yapacağız... Devam edelim:

```
#!/usr/bin/env python3.0

a = 1

while a < 10:
    print("bilgisayar yine çıldırdı!")
```

Ne oldu? İstediğimizi elde edemedik, değil mi? Programımız yine sonsuz döngüye girdi... Bu sonsuz döngüyü kırmak için CTRL+C (veya CTRL+D'ye) basmamız gerekecek yine...

Sizce buradaki hata nereden kaynaklandı? Yani neyi eksik yaptık da programımız sonsuz döngüye girmekten kurtulamadı? Aslında bunun cevabı çok basit. Biz yukarıdaki kodları yazarak Python'a şu emri vermiş olduk:

> *"a'nın değeri 10'dan küçük olduğu müddetçe ekrana "bilgisayar yine çıldırdı!" yazısını bas!"*

a değişkeninin değeri 1. Yani 10'dan küçük... Dolayısıyla Python'un ekrana o çıktıyı basmasını engelleyecek herhangi bir şey yok... Şimdi bu problemi nasıl aşacağımızı göreceğiz:

```
#!/usr/bin/env python3.0

a = 1

while a < 10:
    a = a + 1
    print("bilgisayar yine çıldırdı!")
```

Burada "*a = a + 1*" diye bir şey ekledik kodlarımızın arasına. Bu satır, a'nın değerine her defasında "1" ekliyor. En sonunda a'nın değeri 10'a ulaşınca da, Python ekrana "bilgisayar yine çıldırdı!" cümlesini yazmayı bırakıyor. Çünkü `while` ifadesi içinde belirttiğimiz ölçüte göre, programımızın devam edebilmesi için a'nın değerinin 10'dan küçük olması gerekiyor. a'nın değeri 10'a ulaştığı anda bu ölçüt bozulacaktır.... Gelin isterseniz bu kodları Python'un nasıl algıladığına bir bakalım:

- Python öncelikle *a = 1* satırını görüyor.
- Daha sonra "a'nın değeri 10'dan küçük olduğu müddetçe..." satırını görüyor.
- Ardından a'nın değerini, 1 artırıyor (*a = a + 1*) ve a'nın değeri 2 oluyor.
- a'nın değeri (yani 2) 10'dan küçük olduğu için Python ekrana ilgili çıktıyı veriyor.
- İlk döngüyü bitiren Python başa dönüyor ve a'nın değerinin 2 olduğunu görüyor.

- a'nın değerini yine 1 artırıyor ve a'yı 3 yapıyor.
- a'nın değeri hâlâ 10'dan küçük olduğu için ekrana yine ilgili çıktıyı veriyor.
- İkinci döngüyü de bitiren Python yine başa dönüyor ve a'nın değerinin 3 olduğunu görüyor.
- Yukarıdaki adımları tekrar eden Python, a'nın değerini 9 yapana kadar geliyor.
- a'nın değeri 9'a ulaştığında Python a'nın değerini bir kez daha artırınca bu değer 10'a ulaşıyor.
- Python a'nın değerinin artık 10'dan küçük olmadığını görüyor ve programdan çıkıyor...

Yukarıdaki kodları şöyle yazarsak belki durum daha anlaşılır olabilir:

```
#!/usr/bin/env python3.0

a = 1

while a < 10:
    a = a + 1
    print(a)
```

Burada Python'un arkada ne işler çevirdiğini daha net görebiliyoruz. Kodlarımız içine eklediğimiz `while` döngüsü sayesinde Python her defasında "a" değişkeninin değerini kontrol ediyor ve bu değer 10'dan küçük olduğu müddetçe "a" değişkeninin değerini "1" artırıp, yeni değeri ekrana basıyor. Bu değişkenin değeri 10'a ulaştığında ise, bu değerin artık 10'dan küçük olmadığını anlayıp bütün işlemleri durduruyor.

Gelin isterseniz bu `while` döngüsünü daha önce yazdığımız hesap makinemize uygulayalım:

```
#!/usr/bin/env python3.0

a = 1

while a < 10:
    a = a + 1

    giriş = """
    (1) topla
    (2) çıkar
    (3) çarp
    (4) böl
    (5) karesini hesapla"""

    print(giriş)

    soru = input("Yapmak istediğiniz işlemin numarasını girin: ")

    if soru == "1":
        sayı1 = int(input("Toplama işlemi için ilk sayıyı girin: "))
        sayı2 = int(input("Toplama işlemi için ikinci sayıyı girin: "))
        print(sayı1, "+", sayı2, "=", sayı1 + sayı2)

    elif soru == "2":
        sayı3 = int(input("Çıkarma işlemi için ilk sayıyı girin: "))
        sayı4 = int(input("Çıkarma işlemi için ikinci sayıyı girin: "))
        print(sayı3, "-", sayı4, "=", sayı3 - sayı4)

    elif soru == "3":
```

```
        sayı5 = int(input("Çarpma işlemi için ilk sayıyı girin: "))
        sayı6 = int(input("Çarpma işlemi için ikinci sayıyı girin: "))
        print(sayı5, "x", sayı6, "=", sayı5 * sayı6)

    elif soru == "4":
        sayı7 = int(input("Bölme işlemi için ilk sayıyı girin: "))
        sayı8 = int(input("Bölme işlemi için ikinci sayıyı girin: "))
        print(sayı7, "/", sayı8, "=", sayı7 / sayı8)

    elif soru == "5":
        sayı9 = int(input("Karesini hesaplamak istediğiniz sayıyı giriniz: "))
        print(sayı9, "sayısının karesi =", sayı9 ** 2)

    else:
        print("Yanlış giriş.")
        print("Aşağıdaki seçeneklerden birini giriniz:", giriş)
```

Burada girintilere özellikle dikkat ediyoruz. `while` döngüsünün kendisi zaten girintili bir kod yazmayı gerektirdiği için kendisinden sonra gelen bütün kodları da bir seviye sağa kaydırdık. Ayrıca burada yaptığımız şey aslında kullanıcıya 9 kez hesap yapma hakkı vermek oldu... "a" değişkeninin ilk değeri "1" olduğu ve ölçüt olarak belirlediğimiz sayı 10 olduğu için kullanıcı ancak 9 kez art arda hesap yapabilir bu kodlara göre...

Aslında burada bir problem var. Kullanıcıya 9 kez art arda hesap yapma hakkı veriyoruz, ama ona istediği zaman programdan çıkma hakkı tanımıyoruz. Bu kodları çalıştıran bir kullanıcı programdan çıkmak için 9 kez hesap yapmak zorunda kalacaktır! Bunun iyi bir yöntem olmadığı açık. O halde gelin bu soruna bir çare bulalım. Mesela kullanıcı klavyedeki "ç" harfine bastığında programdan çıkabilsin... Aşağıdaki kodları dikkatlice inceleyin:

```
#!/usr/bin/env python3.0

a = 1

while a == 1:

    giriş = """
    (1) topla
    (2) çıkar
    (3) çarp
    (4) böl
    (5) karesini hesapla

    Programdan çıkmak için "ç" harfine basınız...
    """

    print(giriş)

    soru = input("Yapmak istediğiniz işlemin numarasını girin: ")

    if soru == "1":
        sayı1 = int(input("Toplama işlemi için ilk sayıyı girin: "))
        sayı2 = int(input("Toplama işlemi için ikinci sayıyı girin: "))
        print(sayı1, "+", sayı2, "=", sayı1 + sayı2)

    elif soru == "2":
        sayı3 = int(input("Çıkarma işlemi için ilk sayıyı girin: "))
        sayı4 = int(input("Çıkarma işlemi için ikinci sayıyı girin: "))
        print(sayı3, "-", sayı4, "=", sayı3 - sayı4)
```

```
    elif soru == "3":
        sayı5 = int(input("Çarpma işlemi için ilk sayıyı girin: "))
        sayı6 = int(input("Çarpma işlemi için ikinci sayıyı girin: "))
        print(sayı5, "x", sayı6, "=", sayı5 * sayı6)

    elif soru == "4":
        sayı7 = int(input("Bölme işlemi için ilk sayıyı girin: "))
        sayı8 = int(input("Bölme işlemi için ikinci sayıyı girin: "))
        print(sayı7, "/", sayı8, "=", sayı7 / sayı8)

    elif soru == "5":
        sayı9 = int(input("Karesini hesaplamak istediğiniz sayıyı giriniz: "))
        print(sayı9, "sayısının karesi =", sayı9 ** 2)

    elif soru == "ç":
        print("Tekrar görüşmek üzere!")
        a = 0

    else:
        print("Yanlış giriş.")
        print("Aşağıdaki seçeneklerden birini giriniz:")
```

Bu kodlar bir öncekine göre çok daha sağlıklı oldu. Bir defa artık kullanıcıya istediği kadar hesap yapma imkanı sunuyoruz. Ayrıca kullanıcımız artık programdan istediği zaman çıkabilecek. Giriş paragrafımıza, programdan nasıl çıkılacağını anlatan bir cümle de ekledik. Buna göre eğer kullanıcı program çalışırken "ç" harfine basarsa programdan çıkılacak... Programımız içinde bunu sağlayan kodumuz şu:

```
elif soru == "ç":
    print("Tekrar görüşmek üzere!")
    a = 0
```

Programımızın en başındaki `while` döngüsünü nasıl kurduğumuza dikkat edin. Şöyle dedik:

```
a = 1

while a == 1:
```

Önce a'nın değerini "1" olarak belirledik ve şu emri verdik:

> *"a değişkeninin değeri "1" olduğu müddetçe aşağıdaki kodları çalıştır...*

Dikkat ederseniz burada "*a = a + 1*" gibi bir şey yazmadık. Çünkü buna gerek kalmadı. "a" değişkeninin değerini her döngüde "1" artırmak yerine, a değişkenin değerini, yeni eklediğimiz `elif` bloğu içinde değiştirdik. Farkettiyseniz, bu yeni `elif` bloğu içinde "*a = 0*" diye bir satır var. İşte bu satır sayesinde, kullanıcı "ç" harfine bastığında "a" değişkeninin değeri de "0" olarak belirleniyor. Bizim en başta belirlediğimiz, programın çalışmaya devam etme şartı "*a = 1*" olduğu için, Python artık "a" değişkeninin değerinin "1" değil "0" olduğunu görünce programdan çıkıyor... Ayrıca programımızın en sonundaki `else` bloğunda da ufak bir kozmetik değişiklik yaptık. Daha önce `else` bloğunu şöyle yazmıştık:

```
else:
    print("Yanlış giriş.")
    print("Aşağıdaki seçeneklerden birini giriniz:", giriş)
```

Ama artık bu sondaki "giriş" değişkenini kaldırıyoruz. Çünkü artık programımız sürekli olarak en başa döndüğü ve programın en başında da giriş değişkeni ekrana yazdırıldığı için, programın sonunda bir kez daha bu değişkeni ekrana yazdırmamıza gerek yok. İsterseniz pro-

gramın sonunda “giriş” değişkenini tekrar yazdırarak nasıl bir görüntü elde edildiğini kontrol edebilirsiniz...

Şimdilik `while` döngüsü ile ilgili olarak söyleyeceklerimiz bu kadar. Bundan sonra da bu döngüyü sık sık kullanıp farklı özelliklerini incelemeye devam edeceğiz. Ama şimdi başka bir döngüden bahsedeceğiz: Python’da `for` döngüsü.

3.2 for Döngüsü (for loop)

Python’da işleyeceğimiz ikinci döngü, `for` döngüsüdür. `for` döngüsü, Python’daki en sık kullanılan öğelerden biridir desek abartmış olmayız. Bu döngünün Python programlama dilinde oldukça geniş bir kullanım alanı vardır. Bu döngünün ne işe yaradığını ve nasıl kullanıldığını öğrendikten sonra, Python’da yapabildiklerimizin epey arttığını göreceksiniz. Ayrıca bu bölümde `for` döngüsünü işlerken bir yandan da çok önemli bazı başka yeni araçlar da öğreneceğiz. O halde hiç vakit kaybetmeden bu döngünün ne olup ne olmadığına bir bakalım:

İsterseniz `for` döngüsünü doğrudan tarif etmeye çalışmak yerine, basit bir örnek yardımıyla bu döngüyü kavramaya çalışalım:

```
isim = "istihza"

for harfler in isim:
    print(harfler)
```

Bu kodları incelemeye başlamadan önce, isterseniz kodlarımızı çalıştırıp ne iş yaptığına bir bakalım. Bu kodları çalıştırdığımız zaman şöyle bir çıktı elde edeceğiz:

```
i
s
t
i
h
z
a
```

Gördüğünüz gibi, yazdığımız kodlar, “isim” değişkeninin değerini alıp her bir harfi alt alta ekrana yazdırdı. Peki bu nasıl oldu? Hemen, yazdığımız kodları incelemeye koyulalım.

Öncelikle, her zaman yaptığımız gibi, bir değişken tanımlıyoruz:

```
isim = "istihza"
```

Burada herhangi bir gariplik yok. Devam edelim.

Değişkenimizi tanımladıktan sonra bir `for` döngüsü kuruyoruz:

```
for harfler in isim:
    print(harfler)
```

Burada, Python’a kabaca şöyle bir emir vermiş oluyoruz:

> *“isim” adlı değişkenin içindeki öğelerin her birini tek tek “harfler” olarak adlandır! Ardından da bu “harfler” değişkenini ekrana bas!*

Bu emrimizi duyan Python “istihza” adlı karakter dizisine şöyle bir bakacak, bu karakter dizisi içinde gördüğü bütün harfleri tek tek tarayacak, bu harflerin her birine “harfler” adını verecek ve “harfler” olarak adlandırdığı bu öğeleri tek tek ekrana basacaktır...

Burada neler olup bittiğini daha iyi anlamak için bir örnek daha verelim:

```
for harfler in "Python Programlama Dili":
    print(harfler)
```

Biraz önceki örnekte gördüğümüz gibi, yukarıdaki kodları çalıştırdığımızda da "Python Programlama Dili" adlı karakter dizisinin bütün harfleri tek tek ekrana basılacaktır. Ayrıca yine bu örnekte gördüğümüz gibi, karakter dizimizi, önce bir değişkene atamak zorunda kalmadan doğrudan `for` döngüsü içine de yerleştirebiliyoruz.

Hatırlarsanız, bir önceki bölümde `while` döngüsünü işlerken şuna benzer bir örnek vermiştik:

```
a = 0

while a < 10:
    a = a + 1
    print(a)
```

Bu kodları çalıştırdığımız zaman, 1'den 10'a kadar olan sayıların ekrana basıldığını görürüz. Böyle bir şeyi `for` döngüsünü kullanarak, daha kolay bir şekilde yapabiliriz:

```
for sayılar in range(10):
    print(sayılar)
```

Burada elde ettiğimiz çıktı tabii ki `while` döngüsü ile elde ettiğimiz çıktıdan biraz farklı. `while` döngüsünde, 1'den 10'a kadar olan sayıları elde etmişken, yukarıdaki `for` döngüsünde 0'dan 10'a kadar olan sayıları elde ettik (10 hariç). Bu durum yukarıdaki kodlar içinde kullandığımız `range()` fonksiyonunun çalışma biçiminden kaynaklanıyor. Bu `range()` fonksiyonunu bir sonraki bölümde ayrıntılı olarak işleyeceğiz. Şimdilik bu fonksiyonun sadece yukarıda nasıl kullanıldığına dikkat etmemiz yeterli olacaktır...

`for` döngüsüyle şöyle bir örnek de yapabiliriz:

```
for araya_nokta in "PYTHON":
    print(araya_nokta, end=".")
```

Burada `for` döngüsü "PYTHON" karakter dizisinin bütün öğelerini tek tek tarıyor ve "end" parçacığıyla belirttiğimiz "." işaretini her bir öğenin arasına yerleştiriyor...

Eğer yukarıdaki kodları çalıştırdığınızda komut ekranı yeni satıra geçmiyorsa, yani şöyle bir görüntü elde ediyorsanız:

```
P.Y.T.H.O.N.istihza@istihza:~/Desktop$
```

...yukarıdaki kodları şu şekilde yazabilirsiniz:

```
for araya_nokta in "PYTHON":
    print(araya_nokta, end=".")
print()
```

En son satıra yazdığımız boş `print()` fonksiyonu ekrana boş bir satır basarak, alt satıra geçilmesini sağlayacaktır...

Böylece temel olarak Python'daki döngüleri incelemiş olduk. Python'a ilişkin daha çok şey öğrendikçe bu döngüleri daha yetkin bir biçimde kullanabileceğiz. Şimdi isterseniz, biraz önce bahsettiğimiz `range()` fonksiyonuna bir göz atalım. Bu arada, Python'un 2.x sürümlerini kullanmış olanlar, bir sonraki bölümde işleyeceğimiz `range()` fonksiyonunun 2.x'e göre bazı farklılıklar içerdiğini göreceklerdir...

3.3 range() Fonksiyonu

Bir önceki bölümde `range()` fonksiyonuna kısaca değinmiş, konunun ayrıntısını bu bölüme bırakmıştık. İşte şimdi bu `range()` fonksiyonunun ne işe yaradığını, ne tür özellikleri olduğunu göreceğiz.

"range" kelimesi Türkçe'de "aralık" anlamına gelir. Adından da anlaşılacağı gibi, bu fonksiyon yardımıyla belli bir aralıktaki sayılar üzerinde işlem yapabileceğiz. Bu fonksiyonla birkaç örnek yaparak işe başlayalım. Mesela şu örneğe bakalım (Bu örneği etkileşimli kabukta yazabiliriz):

```
>>> range(10)
```

Bu komutu verdiğimizde şöyle bir çıktı alırız:

```
range(0, 10)
```

Buradan anladığımıza göre, "*range(10)*" komutu, içerisinde 0'dan 10'a kadar olan sayıları barındırıyor (10 sayısı hariç). Bu sayıları listelemek için şöyle bir komut vermemiz gerekir:

```
>>> list(range(10))
```

Burada, `list()` adlı yeni bir fonksiyon daha görüyoruz. Bu fonksiyonu bu derste incelemeyeceğiz. `list()` fonksiyonu daha sonraki bir dersimizin konusu olacak. Biz şimdilik bu fonksiyonun, "listeleme" görevi gördüğünü bilelim. `list()` fonksiyonunun buradaki görevi, `range()` fonksiyonu içindeki sayıları listelemektir. Yukarıdaki komutu işlettiğimizde şöyle bir çıktı alacağız:

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

Böylece `range()` fonksiyonu yardımıyla, 0'dan 10'a kadar olan sayıları listelemiş olduk. Burada 10 sayısının listede yer almadığına dikkat edin. Ayrıca, yukarıdaki komutu etkileşimli kabukta yazdığımız için başa `print()` komutunu eklemek zorunda kalmadık. Ama eğer bu kodları bir dosya içine yazıp kaydetmiş olsaydık, başa mutlaka `print()` fonksiyonunu eklememiz gerekecekti:

```
print(list(range(10)))
```

Burada, açtığımız bütün parantezleri satır sonunda kapatmayı unutmuyoruz... Eğer bütün parantezleri kapatmazsak programımız hata verecektir...

`range()` fonksiyonu bazı başka güzellikler de sunar bize. Mesela bu fonksiyonu kullanırken mutlaka 0'dan başlamak gibi bir mecburiyetimiz yoktur. Örneğin şöyle bir şey yazabiliriz:

```
>>> list(range(1, 11))

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

Gördüğünüz gibi, yukarıdaki komut yardımıyla 1'den 10'a kadar olan sayıları yazdırabildik. Burada, `range()` fonksiyonunun saymaya kaçtan başlayacağını nasıl belirlediğimize dikkat edin. Eğer 2'den 10'a kadar olan bütün sayıları listelemek isteseydik şöyle bir şey yazacaktık:

```
>>> list(range(2, 11))
```

`range()` fonksiyonunun yetenekleri bunlarla sınırlı değildir. Bu fonksiyon yardımıyla yazdırdığımız sayılar birer birer artmak zorunda da değildir. Yani illa "1, 2, 3, 4..." şeklinde ilerlemek zorunda değiliz. İstersek "0, 2, 4, 6..." şeklinde de ilerleyebiliriz:

```
>>> list(range(0, 10, 2))
```

Burada bir nokta dikkatinizi çekmiş olmalı. Yukarıdaki kodda başlangıç sayısı olan 0'ı da belirttik. Normalde, eğer saymaya sıfırdan başlayacaksak, "0" sayısını ayrıca belirtmemize gerek yoktur. Ama yukarıdaki örnekte, yapmak istediğimiz şeyi gerçekleştirebilmek için parantez içinde üç adet sayı belirtmemiz gerekiyor. Çünkü `range()` fonksiyonunun formülü şu şekildedir:

```
range(başlangıç_değeri, bitiş_değeri, atlama_değeri)
```

Burada "başlangıç_değeri" olarak belirttiğimiz yere, saymaya kaçtan başlayacağımızı yazıyoruz. "bitiş_değeri"ne sayma işleminin hangi sayıya kadar devam edeceğini yazacağız. "atlama_değeri" şeklinde gösterdiğimiz ifadeye ise, `range()` fonksiyonunun, başlangıç değerinden bitiş değerine kadar olan sayıları sıralarken kaçar kaçar atlaması gerektiğini yazmamız gerekiyor. `range()` fonksiyonunda "başlangıç_değeri"nin varsayılan bir değeri vardır. Bu değer 0'dır. Yani eğer `range()` fonksiyonunu kullanarak 100'e kadar sayacaksanız ve "*list(range(100))*" gibi bir kod yazmışsanız, Python sizin 0'dan başlamak istediğinizi varsayacaktır. "bitiş_değeri"nin varsayılan bir değeri yoktur. Bunu bizim belirlememiz gerekir. "atlama_değeri"nin varsayılan değeri ise "1"dir. Yani eğer herhangi bir atlama değeri belirtmezseniz, Python sizin birer birer saymak istediğinizi varsayacaktır. Dolayısıyla, "*list(range(100))*" gibi bir kod yazdığımızda Python bunu alttan alta şu şekilde görecektir:

```
>>> list(range(0, 100, 1))
```

Eğer 0'dan 100'e kadar olan sayıları ikişer ikişer atlayarak listelemek istiyorsanız ve bu isteğinizi şu şekilde belirtirseniz Python'un kafasının karışmasına yol açarsınız:

```
>>> list(range(100, 2))
```

Burada Python sizin ne yapmaya çalıştığınızı anlayamaz. Parantez içinde ilk değer olarak "100", ikinci değer olarak ise 2 yazdığınız için, Python bu 100 sayısını başlangıç değeri; 2 sayısını ise bitiş değeri olarak algılayacaktır. Dolayısıyla da Python bu durumda sizin 100'den 2'ye kadar olan sayıları listelemek istediğinizi zannedecek, `range()` fonksiyonuyla bu şekilde geriye doğru sayamayacağımız için de boş bir çıktı verecektir... Bu yüzden, Python'un şaşırmaması için yukarıdaki örneği şu şekilde yazmalıyız:

```
>>> list(range(0, 100, 2))
```

Kısacası, eğer `range()` fonksiyonunun kaçar kaçar sayacağını da belirtmek istiyorsak, parantez içine, gerekli bütün sayıları yazmalıyız...

Bu arada, "*range(100, 2)*" gibi bir ifade yazarak sayıları geriye doğru listeleyemeyecek olmamız, hiç bir şekilde geriye doğru sayamayacağımız anlamına gelmiyor. Elbette "atlama_değeri" parametresine eksi değerli bir sayı vererek `range()` fonksiyonunun geriye doğru saymasını da sağlayabiliriz:

```
>>> list(range(100, 2, -1))
```

Hatta istersek `range()` fonksiyonunun geriye doğru kaçar kaçar sayacağını da belirtebiliriz:

```
>>> list(range(100, 2, -2))
```

...gibi...

Sanırım tekrar hatırlatmaya gerek yok. Yukarıdaki kodları eğer bir dosyaya yazıp çalıştıracaksanız, başlarına `print()` fonksiyonunu da eklemeniz gerekir. Dolayısıyla yukarıda en son verdiğimiz örnek bir dosya içinde şöyle görünecektir:

```
print(list(range(0, 100, 2)))
```

Gördüğünüz gibi, bir sürü parantez işareti oldu kodlarımız içinde. Esasında yukarıda yaptığımız işlemleri yapabilmek için farklı bir yol da izleyebiliriz. `range()` fonksiyonunu yukarıdaki şekillerde kullanmak yerine, bunu bir `for` döngüsü içine de alabiliriz:

```
for i in range(0, 100, 2):
    print(i)
```

Ayrıca `range()` fonksiyonunu bir `for` döngüsü içine aldığımızda, `range()` fonksiyonunun ürettiği sayılarla bazı yararlı işler de yapabiliriz. Örneğin şuna bir bakın:

```
for çarp in range(10):
    print(çarp * 2)
```

Bu kodları çalıştırdığımızda şöyle bir çıktı elde ederiz:

```
0
2
4
6
8
10
12
14
16
18
```

Burada Python'un yaptığı şey, 0'dan 10'a kadar olan bütün sayıları tek tek 2 ile çarpmak oldu... Ayrıca `range()` fonksiyonunu "for" döngüsü içinde kullandığımızda `list()` fonksiyonunu kullanmamıza gerek kalmadığına da dikkat edin.

`for` döngüsünü ve `range()` fonksiyonunu `if` deyimi ile birlikte kullanarak şöyle bir şey da yapabiliriz:

```
for çift_sayılar in range(10):
    if çift_sayılar % 2 == 0:
        print(çift_sayılar)
```

Burada gerçekleşen süreci kabaca şöyle ifade edebiliriz:

0'dan 10'a kadar olan bütün sayıları tek tek tara ve bunların her birini "çift_sayılar" olarak adlandır. Eğer çift_sayılar olarak adlandırdığımız bu sayılar içinde 2'ye tam bölünen sayılar varsa bunları ekrana yazdır...

"%" işlecinin ne işe yaradığını daha önceki derslerimizden hatırlıyorsunuz....

BÖLÜM 4

Listeler

4.1 Giriş

Bu bölümde yine Python'un çok önemli bir öğesinden bahsedeceğiz. Bu dersimizin konusu "Python'da Listeler".

Aslında biz `range()` fonksiyonunu incelerken, ucundan da olsa listeleri görmüştük. Mesela şu örneği hatırlıyorsunuz:

```
list(range(10))
```

Bu komut şöyle bir çıktı veriyordu:

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

İşte yukarıda gördüğümüz bu çıktı aslında bir "liste"dir... Teknik bir dille ifade etmek gerekirse, "listeler" Python'daki "veri tipleri"nden (data types) yalnızca bir tanesidir. Şimdiye kadar gördüğümüz veri tipleri arasında "sayılar" ve "karakter dizileri" vardı... Dolayısıyla, "sayılar", "karakter dizileri" ve "listeler" Python'da "veri tipi" adı verilen bir grubun üyesidir. Bu teknik terimleri elbette ezberlemek zorunda değilsiniz. Ama bilmenin zararını değil faydasını görürsünüz.

Python'da "listeler" konusuna kısa bir giriş yaptığımıza göre artık yavaş yavaş ayrıntılara inmeye başlayabiliriz.

4.2 Listeleri Tanımlamak

Python'da listeleri kullanabilmek için öncelikle bunları tanımlamamız gerekir. Endişelenmeyin. "Liste tanımlamak" ifadesinin kendisi biraz fantastik duruyor olsa da, aslında liste tanımlamak çok kolaydır!

Listeleri şöyle tanımlıyoruz:

```
liste = []
```

Burada hiç bir öğe içermeyen boş bir liste tanımladık. Bunu ekrana basmak istersek şöyle yapmamız gerektiğini biliyorsunuz:

```
print(liste)
```

Tabii boş bir liste tanımladığımız için elde ettiğimiz çıktı da boş bir liste oldu... Burada şaşılacak bir şey yok. Şimdi isterseniz gelin boş bir liste yerine, içinde birkaç öğe barındıran daha işlevli bir liste yapalım:

```
diller = ["python", "php", "perl", "C++"]
```

Gördüğünüz gibi, programlama dillerinin bazılarını sıraladığımız, "diller" adlı bir liste oluşturduk. Dikkat ederseniz, liste tanımlamak, değişken veya karakter dizisi tanımlamaya çok benziyor. Listeleri başka veri tiplerinden ayıran en önemli özellik, "[]" işaretidir. Bu işaret, arka arkaya dizilmiş öğelerin bir "liste" olduğunu gösterir. Yani "[]" işareti Python'da listelerin ayırt edici özelliğidir. Gelin isterseniz elimiz alışsın diye bir başka liste daha tanımlayalım:

```
alışveriş = ["elma", "armut", "kabak", "ekmek", "çilek", "şeker", "şeftali"]
```

Burada da basit bir alışveriş listesi tanımladık. Dikkat ederseniz, listenin bütün öğeleri birer karakter dizisi. Bunu tırnak içine alınmış olmalarından anlıyoruz. Listelerin ayırt edici özelliği nasıl "[]" işareti ise, karakter dizilerinin ayırt edici özelliği de tırnak işaretleridir... Elbette sadece karakter dizilerinden oluşan listeler tanımlamak zorunda değiliz. İstersek, öğeleri sayılardan oluşmuş bir liste de tanımlayabiliriz:

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

Bu listemizin öğeleri ise tamsayılardan (integers) oluşmuş. Hatta istersek, farklı veri tiplerinden oluşmuş melez bir liste dahi tanımlayabiliriz:

```
liste1 = ["kedi", 4, 3.5]
```

Hatta inanmayacaksınız ama, tanımladığımız bir listenin öğesi başka bir liste dahi olabilir:

```
liste2 = ["kuzu", 12, 15.4, ["salam", "sucuk"], liste1]
```

Sonuncu listemizi dikkatlice inceleyin. Burada bildiğimiz bütün veri tiplerini tek bir listeye yerleştirdik. Bu listeyi ekrana bastığımızda şöyle bir çıktı elde ederiz:

```
>>> print(liste2)

['kuzu', 12, 15.4, ['salam', 'sucuk'], ['kedi', 4, 3.5]]
```

Bu listede bir önceki "liste1" adlı listenin kendisinin de yer aldığına dikkat edin...

Gördüğünüz gibi Python'da bir liste tanımlamak hem çok kolay, hem de öğelerin çeşitliliği açısından oldukça özgürüz. İsterseniz şimdi bu listelerle neler yapabileceğimize bakalım biraz.

4.3 Listelerin Öğelerine Erişmek

Bir önceki bölümde güzel güzel tanımladık listelerimizi. Peki bu listeleri tanımladık da ne oldu? Biz bu listeleri tanımladıktan sonra ne yapacağız? İşte bu bölümde bu sorulara cevap vermeye çalışacağız.

Öncelikle tanımladığımız listelerin öğelerine nasıl erişebileceğimize bakalım...

Hemen bir liste tanımlamakla başlayalım işe:

```
sistemler = ["GNU/Linux", "Mac Os X", "Windows"]
```

Böylece işletim sistemlerinin adlarını içeren bir liste tanımlamış olduk. Bir önceki derste öğrendiğimize göre bu listeyi şu şekilde ekrana basabiliriz:

```
print(sistemler)
```

Peki biz bu listenin bütün öğelerini değil de sadece istediğimiz bir öğesini yazdırmak istersek ne olacak?

Python'da liste öğelerinin her birinin bir sırası vardır. Liste öğelerine, öğenin sırasını kullanarak ulaşabiliriz. Formülümüz şöyle olacak:

```
liste[öğenin_sırası]
```

Buna hemen bir örnek verelim. Diyelim ki "sistemler" adlı listedeki "GNU/Linux" öğesini almak istiyoruz. Şöyle bir şey deneyelim:

```
>>> print(sistemler[1])

'Mac Os X'
```

Ne oldu? "GNU/Linux" yerine, listenin ikinci öğesi olan "Mac Os X"i verdi bu komut bize, değil mi? Ama bizim istediğimiz bu değildi... Gelin bir de şöyle bir şey deneyelim:

```
>>> print(sistemler[0])

'GNU/Linux'
```

Bu sefer oldu. Buradan anlıyoruz ki Python saymaya sıfırdan başlıyor. Yani listemizin ilk öğesine ulaşmak için "1" değil, "0" sayısını kullanmamız gerekiyor. Bu kuralı göz önüne alarak sırasıyla bütün öğelerimizi yazdıralım:

```
>>> print(sistemler[0])

'GNU/Linux'

>>> print(sistemler[1])

'Mac Os X'

>>> print(sistemler[2])

'Windows'
```

Eğer sıra numarası olarak pozitif bir sayı yerine negatif bir sayı kullanırsak, liste soldan sağa değil, sağdan sola, yani geriye doğru taranacaktır:

```
>>> print(sistemler[-1])

'Windows'

>>> print(sistemler[-2])

'Mac Os X'

>>> print(sistemler[-3])

'GNU/Linux'
```

Burada listenin en son öğesine, “-0” diye bir şey olamayacağı için “-1” ifadesiyle erişiyoruz... Eğer listede olmayan bir öğenin sırasını sorgularsak hata mesajı alırız. Yani “print(sistemler[5])” veya “print(sistemler[-4])” gibi komutlar hata verecektir. Çünkü “sistemler” adlı listede “beşinci” veya “eksi dördüncü” bir öğe yok...

Python’daki listelerin bir başka özelliği ise, yukarıdaki sıra numaralarının, bir aralık belirtecek şekilde kullanılabilmesidir. Bu ne demek? Yani bir listeden her defasında sadece tek bir öğe almak zorunda değiliz. İstersek bir kaç öğeyi birden de alabiliriz. Bununla ilgili hemen bir örnek verelim. Şöyle bir listemiz olsun:

```
>>> isimler = ["Udith Narayan", "Sonu Nigam", "Kajol Devgan",
... "Rani Mukherjee", "Preity Zinta", "Lata Mangeshkar"]
```

Şimdi mesela yukarıdaki listeden 1-3 arası öğeleri alalım:

```
>>> print(isimler[1:3])

['Sonu Nigam', 'Kajol Devgan']
```

Burada “0-3” arası öğeleri değil, “1-3” arası öğeleri aldığımıza dikkat edin. Bildiğiniz gibi, listelerin ilk öğesinin sırası “0”dır. Ayrıca “1-3” şeklinde bir aralık belirttiğimizde sadece birinci ve ikinci öğelerin alındığına dikkat edin. Yani isimler[1:3] dediğimizde, alacağımız sonuç 1. öğeyi içerirken 3. öğeyi içermez... Bu arada, bu işleme Python’cada “dilimleme” (slicing) adı verilir. Gerçekten de burada yaptığımız şey bir listeyi “dilimlemekten” ibarettir...

Aynı listeden devam edelim:

```
>>> print(isimler[2:5])

['Kajol Devgan', 'Rani Mukherjee', 'Preity Zinta']
```

Liste öğelerine tek tek erişirken yaptığımız gibi, burada da negatif sayılardan yararlanabiliriz:

```
>>> print(isimler[0:-1])
```

Gördüğünüz gibi, yukarıdaki komut bir listenin son öğesini kırpmak için gayet etkili bir yoldur. Burada farklı sayılarla kendi kendinize deneme yaparsanız Python’daki dilimleme mekanizmasını daha iyi kavrayabilirsiniz.

Listelerdeki bu sıra sayılarının bazı varsayılan değerleri vardır. Mesela, eğer ilk sayıyı belirtmezseniz, Python ilk sayıyı “0” varsayacaktır. Yani:

```
>>> print(isimler[:4])

['Udith Narayan', 'Sonu Nigam', 'Kajol Devgan', 'Rani Mukherjee']
```

Gördüğünüz gibi, ilk sayıyı belirtmediğimizde Python sanki “isimler[0:4]” yazmışız gibi davrandı. Peki ya ikinci sayıyı belirtmezsek ne olur?

```
>>> print(isimler[0:])

['Udith Narayan', 'Sonu Nigam', 'Kajol Devgan', 'Rani Mukherjee',
 'Preity Zinta', 'Lata Mangeshkar']
```

Burada da Python sanki “isimler[0:6]” yazmışız gibi davrandı. Yani listenin sonuna kadar gitti.

Listelerle yapabileceklerimiz bunlarla sınırlı değildir. Yukarıda gördüğümüz gibi, listelere iki farklı sıra sayısı vererek liste öğelerine teker teker veya topluca erişebiliyoruz. Eğer iki farklı sıra sayısıyla birlikte üçüncü bir sayı daha verirsek, liste öğelerini, bu verdiğimiz üçüncü sayı kadar atlayarak ekrana yazdırabiliriz. Bu tarif bir örnekle daha kolay anlaşılacaktır:

```
>>> print(isimler[0:6:2])

['Udith Narayan', 'Kajol Devgan', 'Preity Zinta']
```

Gördüğünüz gibi, üçüncü sayı olarak verdiğimiz "2" sayesinde liste öğelerini birer atlayarak ekrana yazdırdık. Burada farklı sayılar deneyerek sistemi daha iyi anlayabilirsiniz.

Aslında yukarıdaki son örneği, listelerin varsayılan değerlerinden yararlanarak daha basit bir şekilde yazabilirsiniz:

```
>>> print(isimler[::2])

['Udith Narayan', 'Kajol Devgan', 'Preity Zinta']
```

Bildiğiniz gibi, eğer ilk sayıyı belirtmezseniz Python "0" demek istediğinizi, ikinci sayıyı belirtmezseniz de listedeki öğe sayısını kullanmak istediğinizi varsayacaktır. Dolayısıyla bizim "0" ve "6"yı ayrıca belirtmemiz şart değil.

Şimdi vereceğimiz örneği dikkatlice inceleyin:

```
>>> print(isimler[::-1])

['Lata Mangeshkar', 'Preity Zinta', 'Rani Mukherjee', 'Kajol Devgan',
 'Sonu Nigam', 'Udith Narayan']
```

Gördüğünüz gibi, üçüncü sayı olarak negatif bir değer de verebiliyoruz. Üçüncü sayı olarak "-1" verdiğimiz zaman Python listeyi geriye doğru sıralayacaktır. Dolayısıyla yukarıdaki komutu kullanarak, herhangi bir listeyi ters çevirebiliriz. Bu özellik bir yerlerde mutlaka işinize yarayacaktır.

Hatırlarsanız bir önceki bölümde şöyle iki tane listemiz vardı:

```
>>> liste1 = ["kedi", 4, 3.5]

>>> liste2 = ["kuzu", 12, 15.4, ["salam", "sucuk"], liste1]
```

Burada "liste1" adlı ilk listemizi doğrudan "liste2" adlı ikinci listemizin içinde kullanmıştık. "liste2" adlı listeyi yazdırdığımızda şöyle bir çıktı alıyorduk:

```
>>> print(liste2)

['kuzu', 12, 15.4, ['salam', 'sucuk'], ['kedi', 4, 3.5]]
```

Bu liste biraz karışık görünüyor. Bakalım bu listelerin öğelerine nasıl ulaşabiliriz... Sırasıyla deneyelim:

```
>>> print(liste2[0])

'kuzu'

>>> print(liste1[1])

12

>>> print(liste1[2])

15.4

>>> print(liste1[3])
```

```
['salam', 'sucuk']
```

Gördüğünüz gibi, son örnekte tek bir öğe değil, listenin kendisini alıyoruz. Çünkü "liste2" adlı listenin üçüncü öğesi bir listedir. (["salam", "sucuk"]). Mesela sadece "salam" öğesini almak istersek şöyle bir yol izlememiz gerekir:

```
>>> print(liste2[3][0])

'salam'
```

Yani burada kabaca şöyle bir şey söylemiş olduk:

> "*Bana liste2 adlı listenin üçüncü öğesinin sıfırıncı öğesini ver!*"

Aynı listenin bir sonraki öğesi de bir listedir (["kedi", 4, 3.5]). Bu listedeki "3.5" öğesine ulaşmak için şöyle bir şey yazmamız gerekir:

```
>>> print(liste2[4][2])

3.5
```

Eğer liste2 listesinin dördüncü öğesini tersten sıralamak isterseniz şöyle bir şey yazmanız gerektiğini tahmin edebilirsiniz:

```
>>> print(liste2[4][::-1])

[3.5, 4, 'kedi']
```

Böylelikle listelere ilişkin önemli bir konuyu geride bırakmış olduk. Bu noktada sizin yapmanız gereken, yukarıdaki örneklere bağlı kalmadan kendi kendinize bazı örnek kodlar yazmaktır. Kendiniz olabildiğince büyük bir liste oluşturun ve başlayın bu listenin öğelerine erişmeye... Oluşturduğunuz bu listeye değişik sayılar verin. Bakın bakalım ne tür sonuçlar elde ediyorsunuz...

4.4 len() Fonksiyonu

Bir önceki bölümde gördüğünüz gibi, bir listenin kaç öğeden oluştuğunu bilmek önemli bir konu. Bir listedeki öğeleri elle saymaya çalışmak tabii ki hiç pratik bir yol değil. Neyse ki Python bize bu tür işlemler için işimize yarayacak bir fonksiyon sunuyor. Bu fonksiyonun adı `len()`. "len" ifadesi, İngilizce'deki "length" (uzunluk) kelimesinin kısaltmasıdır. Bu fonksiyon yardımıyla Python'da uzunluk ölçme işlemlerini yapacağız. Hemen bir örnek verelim:

```
>>> liste = ["kitap", "defter", "kalem", "silgi", "tebeşir",
... "tahta", "sıra", "öğrenci", "okul", "önlük"]

>>> print(len(liste))

10
```

Demek ki listemizde 10 tane öğe varmış. Peki bu listenin sonuncu öğesinin sırası kaçtır? Tabii ki 10 değildir. Hemen deneyip görelim:

```
>>> print(liste[10])

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Gördüğünüz gibi, listemizde 10. öğe diye bir şey yok. Çünkü bildiğiniz gibi listelerde saymaya 0'dan başlıyoruz. Yalnız farkettiyseniz `len()` fonksiyonu doğal olarak saymaya sıfırdan başlamıyor. Çünkü bu fonksiyon liste öğelerinin sırasıyla ilgilenmiyor. Bu fonksiyonun ilgilendiği şey listedeki öğelerin sayısı.. `len()` fonksiyonu bir listedeki öğenin sırasıyla ilgili herhangi bir bilgi vermez bize. Bu fonksiyonu kullanarak o listenin kaç adet öğesi olduğunu öğreniriz. Burada uyguladığımız `len()` fonksiyonuna göre bizim listemizin 10 adet öğesi varmış. Bu öğeleri sıralarken 0'dan başladığımız için sonuncu öğe doğal olarak 10-1 olacaktır. Yani sonuncu öğemizin sırası 9'dur:

```
>>> print(liste[9])

önlük
```

Herhangi bir listenin sonuncu öğesine kısa yoldan şöyle ulaşabileceğimizi bir önceki bölümde görmüştük:

```
>>> print(liste[-1])
```

Bu komut da bir öncekinde olduğu gibi "önlük" çıktısını verecektir. İsterseniz herhangi bir listenin son öğesine şu şekilde de ulaşabilirsiniz:

```
>>> print(liste[len(liste)-1])

önlük
```

Bu son komut gözünüze biraz karışık görünmüş olabilir, ama aslında hiç de karışık değildir. Şöyle düşünelim:

Listemizin 10 adet öğesi var. Python'da saymaya 0'dan başladığımız için, mantık olarak son öğe liste uzunluğunun bir eksiği olacaktır. Yani bizim örneğimizde son öğe "10-1" şeklinde gösterilebilir... Yani aslında yukarıdaki kodu şöyle de yazabiliriz:

```
>>> print(liste[10-1])

önlük
```

Ama böyle bir kod yazabilmek için listenin uzunluğunu tam olarak biliyor olmamız lazım. Listenin uzunluğunu önceden bilmediğimizi varsayarsak, yukarıdaki kodda "10" yerine bir formül yazmamız gerekir. Bize liste uzunluğunu tam olarak verebilecek kodun şu olduğunu biliyoruz:

```
>>> len(liste)
```

Dolayısıyla, bir listenin son öğesini bulmak için liste uzunluğunun bir eksiğini almamız, yani liste uzunluğundan "1" çıkarmamız yeterli olacaktır...

`len()` fonksiyonunu, daha önce öğrendiğimiz `range()` fonksiyonuyla birlikte de kullanabiliriz:

```
>>> print(list(range(len(liste))))
```

Yukarıdaki kod şununla eşdeğerdir:

```
>>> print(list(range(10))
```

Hatta istersek `for` döngüsünü kullanarak da benzer bir sonuç elde edebiliriz:

```
>>> for i in range(len(liste)):
...     print(i)
```

Gelin isterseniz şimdiye kadar öğrendiklerimizle az çok yararlı bir program yazalım. Mesela yukarıdaki listemizin öğelerini tek tek ekrana basalım ve her öğenin solunda birer numara gösterelim. Yani elde etmek istediğimiz çıktı şöyle bir şey olsun:

```
1. kitap
2. defter
3. kalem
4. silgi
5. tebeşir
6. tahta
7. sıra
8. öğrenci
9. okul
10.önlük
```

Öncelikle şöyle bir şeyler deneyelim:

```
>>> for eleman in liste:
...     print(eleman)

kitap
defter
kalem
silgi
tebeşir
tahta
sıra
öğrenci
okul
önlük
```

Liste öğelerini başarıyla ekrana yazdırdık. Ama bu çıktı bizim yapmaya çalıştığımız şeyden biraz uzak oldu. Demek ki kodlarımızda bazı değişiklikler yapmamız gerekiyor. Bir defa, bu öğelerin her birinin sırasını da ekrana basabilmeliyiz. Yani “1. defter”, “2. kalem”, gibi...

```
>>> for i in range(len(liste)):
...     print(i, liste[i])

0 kitap
1 defter
2 kalem
3 silgi
4 tebeşir
5 tahta
6 sıra
7 öğrenci
8 okul
9 önlük
```

Biraz karışık mı göründü gözünüze? Hiç endişelenmeyin. Burada ne yaptığımızı tane tane anlatacağız şimdi... Öncelikle şu satıra bakalım:

```
>>> for i in range(len(liste)):
...
```

Burada yaptığımız şey, listedeki öğe sayısını `range()` fonksiyonuna atamak. Yani aslında şöyle bir şey yazmış olduk:

```
>>> for i in range(10):
...
```

Elbette buraya her zaman “10” gibi sabit bir sayı yazamayabiliriz. Çünkü burada kullandığımız listeyi kendimiz oluşturduk ve kaç öğeden meydana geldiğini biliyoruz, ama gerçek hayatta bir listeyi her zaman kendiniz hazırlamamış olabilirsiniz. Mesela başkasının oluşturduğu bir listeyi alıp üzerinde işlem yapmanız gerekebilir. Öyle bir durumda o listenin kaç öğeden oluştuğunu kesin olarak bilemezsiniz. Hatta yazdığınız bir programın işleyişi sırasında sürekli olarak büyüyen bir liste de söz konusu olabilir. Bu tür durumlar için en uygun yol, sabit bir sayı yazmaya çalışmak yerine, bir formül kullanmak olacaktır. Bir listenin uzunluğunu bize verecek en uygun formül “len(liste)” formülüdür... Dolayısıyla şöyle bir şey yazmamız gerekir:

```
>>> for i in range(len(liste)):
...
```

Şimdi yazdığımız kodları incelemeye devam edelim. İlk satırla beraber, ikinci satır olarak şöyle bir şey yazdık:

```
>>> for i in range(len(liste)):
...     print(i, liste[i])
```

İlk satırı biraz önce açıklamıştık. Gelelim ikinci satıra... Burada öncelikle “i” değişkenini ekrana yazdırıyoruz. Şöyle bir şey yazarak resmi basitleştirelim:

```
>>> for i in range(len(liste)):
...     print(i)

0
1
2
3
4
5
6
7
8
9
```

Gördüğünüz gibi, bu kodlar, elimizdeki listenin uzunluğu kadar sayıyı ekrana basıyor. Yani sanki şöyle bir şey yazmışız gibi:

```
>>> for i in range(10):
...     print(i)
```

Şimdi de kodlarımızın geri kalanına bakalım. Önce tekrar topluca görelim kodlarımızı:

```
>>> for i in range(len(liste)):
...     print(i, liste[i])
```

Bir önceki bölümden hatırlayacağınız gibi, listelerin öğelerine tek tek ulaşmak istediğimizde şöyle bir yol izliyorduk:

```
>>> print(liste[0])
```

...veya...

```
>>> print(liste[2])
```

...gibi...

İşte yukarıda da aslında bu kalıba uygun bir kod yazdık... Birazcık sadeleştirelim:

```
>>> for i in range(len(liste)):
...     print(liste[i])

kitap
defter
kalem
silgi
tebeşir
tahta
sıra
öğrenci
okul
önlük
```

Sanırım ne yaptığımızı artık anladınız. Gördüğünüz gibi, *range(len(liste))* komutu, listenin uzunluğu kadar olan sayıları ekrana basıyor. İşte biz bu sayıları *liste[i]* şeklinde tek tek kullanıyoruz. Buradaki `for` döngüsü sayesinde *range(len(liste))* komutunun ürettiği her bir sayı tek tek *liste[i]* komutuna atanıyor. liste[0], liste[1], liste[2], liste[3], gibi...

Çıktıda görünen liste öğelerinin sol tarafında her birinin sırasının da görünmesi için kodlarımızı tabii ki şöyle yazdık:

```
>>> for i in range(len(liste)):
...     print(i, liste[i])

0 kitap
1 defter
2 kalem
3 silgi
4 tebeşir
5 tahta
6 sıra
7 öğrenci
8 okul
9 önlük
```

Yalnız, gördüğünüz gibi kodlarımız henüz elde etmek istediğimiz çıktıdan hâlâ biraz farklı... Bizim elde etmek istediğimiz çıktı şöyle bir şeydi:

```
1. kitap
2. defter
3. kalem
4. silgi
5. tebeşir
6. tahta
7. sıra
8. öğrenci
9. okul
10.önlük
```

Demek ki kodlarımıza hâlâ bazı eklemeler yapmamız gerekiyor. Gelin şimdi son darbeyi vuralım:

```
>>> for sondarbe in range(len(liste)):
...     print(sondarbe, liste[sondarbe], sep=". ")

0. kitap
```

```
1. defter
2. kalem
3. silgi
4. tebeşir
5. tahta
6. sıra
7. öğrenci
8. okul
9. önlük
```

"sep" parçacığını önceki derslerimizden hatırlıyorsunuz. Bu parçacık sayesinde öğeler arasına istediğimiz ifadeyi yerleştirebiliyorduk. Biz yukarıdaki örnekte her bir öğenin arasına "." işareti koymak için kullandık bu parçacığı...

Yalnız, her ne kadar yukarıdaki kodlarla son darbeyi vurduğumuzu söylesek de aslında son darbeyi henüz vuramadık... Neden mi? Çünkü çıktıda saymaya sıfırdan başlanıyor. Bizim istediğimiz şey saymaya "0"dan değil, "1"den başlamaktı. Saymaya 0'dan başlamak Python için doğal olsa da son kullanıcı açısından biraz tuhaf gelebilecek bir şeydir... İyi bir programcı, yazdığı programlarda mutlaka son kullanıcıyı da düşünmelidir. Sonuçta sizin yazdığınız programı sizden çok, programlamadan hiç bir şey anlamayan insanlar kullanacaktır. O yüzden son kullanıcıyı düşünmeniz gerekir... Bu sorunu da ufak bir kodla halledeceğiz:

```
>>> for sondarbe in range(len(liste)):
...     print(sondarbe+1, liste[sondarbe], sep=". ")

1. kitap
2. defter
3. kalem
4. silgi
5. tebeşir
6. tahta
7. sıra
8. öğrenci
9. okul
10.önlük
```

Gördüğünüz gibi, "sondarbe+1" şeklinde bir kod yazarak sorunumuzu çözebildik. Böylece Python her bir sayıya "1" ekleyecek, dolayısıyla her bir öğenin sırası "1" sayı artırılacaktır. Bu da bize yukarıdaki gibi, kullanıcı açısından daha doğal görünen bir çıktı verecektir.

Böylece `len()` fonksiyonunu enine boyuna incelemiş olduk. Yalnız bir noktayı özellikle vurgulamakta fayda var: `len()` fonksiyonu sadece listelerle kullanılabilecek bir fonksiyon değildir. `len()` fonksiyonunu, uzunluğu ölçülebilecek her şey için kullanabiliriz. Mesela karakter dizilerinin de bir uzunluğu vardır. Hemen bir örnek verelim:

```
>>> a = "Kahramanmaraşlı Abdullah"

>>> print(len(a))

24
```

Demek ki "Kahramanmaraşlı Abdullah" karakter dizisinde 24 adet karakter varmış. Bu 24 sayısına iki kelime arasındaki boşluk karakterinin de dahil olduğunu unutmayın. Çünkü boşluğun kendisi de bir karakterdir...

Hatta listelerin bir özelliği olarak gördüğümüz "dilimleme" tekniği, karakter dizilerine de uygulanabilir. Buna da bir örnek verelim:

```
>>> a = "Kahramanmaraşlı Abdullah"

>>> print(a[0])

K
```

Şimdilik burada bir nokta koyup konuyu kapatalım. Daha sonraki derslerimizde "Karakter Dizileri" konusunu yeniden ve oldukça ayrıntılı bir şekilde ele alacağız. O zaman karakter dizilerinin bütün özelliklerine tek tek bakacağız. Biz listeleri incelemeye devam edelim...

4.5 Liste Öğelerinde Değişiklik Yapmak

Şimdiye kadar listelerle ilgili pek çok şey öğrendik sayılır. Ama henüz bir listenin öğelerini nasıl değiştirebileceğimizi bilmiyoruz. İşte bu derste bu eksiğimizi kapatacağız. Yani bir listedeki tanımlanmış öğeleri nasıl değiştireceğimizi göreceğiz. İşe bir liste tanımlayarak başlayalım:

```
>>> met = ["Kwrite", "Kate", "Emacs", "Vi", "Notepad"]
```

Diyelim ki biz bu listenin ilk öğesini değiştirmek istiyoruz. Listemizin ilk öğesi olan "Kwrite" yerine "Vim" yazalım:

```
>>> met[0] = "Vim"
```

Şimdi listemizin son halini kontrol edelim:

```
>>> print(met)

['Vim', 'Kate', 'Emacs', 'Vi', 'Notepad']
```

Gördüğünüz gibi, listemizin ilk öğesi olan "Kwrite" gitti, onun yerine "Vim" geldi. Python'da liste öğelerini değiştirmenin ne kadar kolay olduğunu görüyorsunuz. Siz de bu listenin başka öğelerini değiştirerek pratik yapabilirsiniz...

Bu noktada listelerin ilginç bir özelliğinden söz edeceğiz. Şimdi "met" adlı listemizi başka bir değişkene atayalım:

```
>>> yeni_liste = met

>>> print(met)

['Vim', 'Kate', 'Emacs', 'Vi', 'Notepad']

>>> print(yeni_liste)

['Vim', 'Kate', 'Emacs', 'Vi', 'Notepad']
```

Gördüğünüz gibi "yeni_liste" ve "met" aynı nesneye atıfta bulunuyor. Dolayısıyla ikisi de aynı çıktıyı veriyor. Şimdi şöyle bir işlem yapalım:

```
>>> yeni_liste[0] = "IDLE"
```

Şimdi iki listeyi de yazdıralım:

```
>>> print(yeni_liste)

['IDLE', 'Kate', 'Emacs, 'Vi', 'Notepad']
```

```
>>> print(met)

['IDLE', 'Kate', 'Emacs', 'Vi', 'Notepad']
```

Gördüğünüz gibi, biz sadece "yeni_liste" adlı listede değişiklik yaptığımız halde, bu değişiklik "met" adlı listeyi de etkiledi... Buradan şu sonucu çıkarıyoruz: Bir listeyi başka bir değişkene atadığınız zaman, aslında o listeyi kopyalamış olmuyorsunuz. Yaptığınız şey, varolan bir nesneye, yani listeye başka bir ad daha vermekten ibaret... Dolayısıyla burada hem "yeni_liste" hem de "met" aynı nesneyi gösteriyor. Eğer amacınız bir listeyi kopyalamaksa en basit şekilde şöyle bir yol izleyebilirsiniz:

```
>>> yeni_liste = met[:]
```

Artık bu listelerden birinde yaptığınız değişiklik ötekini etkilemeyecektir. Çünkü bu şekilde "met" adlı listenin bütün öğelerini baştan sona kopyalayıp yeni_liste adlı değişkene atadınız...

Yalnız şu ayrıma dikkat etmek gerekir:

Bir listenin öğeleri üzerinde değişiklik yapmakla, yeni bir liste oluşturmak farklı şeylerdir. Bu ne demek? Yani şöyle:

```
>>> yeni_liste = met

>>> print(met)

['Vim', 'Kate', 'Emacs', 'Vi', 'Notepad']

>>> print(yeni_liste)

['Vim, 'Kate', 'Emacs','Vi', 'Notepad']
```

Tıpkı biraz önce yaptığımız gibi, met adlı listeyi, yeni_liste adlı değişkene atadık. Artık elimizde, aynı nesneye işaret eden iki farklı değişken var. Şimdi şöyle bir işlem yapıyoruz:

```
>>> yeni_liste = ["falanca", "filanca", "şu", "bu", "o"]
```

Burada yeni_liste adlı listenin öğeleri üzerinde değişiklik yapmıyoruz. Yaptığımız şey, "yeni_liste" adını taşıyan, tamamen yeni bir liste tanımlamak... Şimdi bu yeni tanımladığımız listeyi yazdıralım:

```
>>> print(yeni_liste)

['falanca', 'filanca', 'şu', 'bu', 'o']
```

Bir de "met" adlı listenin durumuna bakalım:

```
>>> print(met)

['Vim', 'Kate', 'Emacs', 'Vi', 'Notepad']
```

Gördüğünüz gibi, burada "met" adlı listeyi kopyalamadan yeni_liste adlı değişkene atadığımız halde, yeni_liste'de sonradan yaptığımız değişiklik metin_düzenleyici'yi etkilemedi. Çünkü dediğimiz gibi, yukarıdaki yeni_liste'yi değiştirirken aslında bir liste üzerinde değişiklik yapmadık. Biz burada bambaşka bir liste oluşturduk. O yüzden iki listeden birinde yapılan değişiklik ötekini etkilemedi... Ama eğer yeni_liste[1] = "falanfilan" şeklinde bir değişiklik yapsaydık, bu değişiklikten öteki liste de etkilenecekti. Çünkü burada zaten varolan bir liste üzerinde değişiklik yapmış oluyoruz. Bu ikisi arasındaki ayrıma çok dikkat etmek gerekir.

4.6 “in” Parçacığı ile Aitlik Kontrolü

Konu başlığımız biraz bulanık görünmüş olabilir gözünüze. Ama hiç endişelenmeyin. Zor bir konudan bahsetmeyeceğiz. Bu derste işleyeceğimiz konu hem zevkli hem de epey faydalıdır. Bu bölümde `in` parçacığını kullanarak bir öğenin listede bulunup bulunmadığını kontrol edeceğiz. Yani “aitlik kontrolü” yapacağız...

Diyelim ki şöyle bir listemiz var:

```
>>> liste = ["elma", "armut", "karpuz", "şeftali"]
```

Şimdi bir öğenin bu listede bulunup bulunmadığını sorgulayacağız:

```
>>> "elma" in liste

True
```

Çıktımız “True” oldu. Bu kelime İngilizce’de “doğru” anlamına gelir. Demek ki “elma” karakter dizisi listeye “ait”miş... Yani bu öğe listede varmış... Bir de şuna bakalım:

```
>>> "erik" in liste

False
```

Bu defa “False” çıktısı aldık. Bu kelime ise İngilizce’de “Yanlış” anlamına gelir. Demek ki “erik” listede yokmuş.

Bununla ilgili daha anlamlı bir örnek verelim:

```
üyeler = ["Ahmet", "Mehmet", "Selim", "Süleyman", "Ayşe",
          "Fatma", "Hale", "Jale", "Lale"]

ad = input("Lütfen adınızı giriniz:")

if ad in üyeler:
    print("Gizli derneğimizin üyelerindensiniz. Buyrun içeri!")

else:
    print("Üye değilsiniz!")
```

Bu `in` parçacığını `for` döngülerini işlerken de görmüştük. Bu parçacığın oradaki işleviyle buradaki işlevinin aslında birbirine çok benzer olduğunu görüyorsunuz... Bu parçacık her iki durumda da “içinde” anlamı katıyor cümleye...

Bu parçacığı elbette sadece listelerle birlikte kullanmak zorunda değiliz. Mesela bunu karakter dizileriyle de kullanabiliriz:

```
>>> a = "Mehmet"

>>> "e" in a

True
```

Bu `in` parçacığı ile birlikte kullanılan `not` parçacığını da burada anmamız oldukça faydalı olacaktır. `not` parçacığı kodlarımıza olumsuzluk anlamı katar. Mesela yukarıdaki örneği “not” kullanarak şöyle de yazabiliriz:

```
üyeler = ["Ahmet", "Mehmet", "Selim", "Süleyman", "Ayşe",
          "Fatma", "Hale", "Jale", "Lale"]
```

```
ad = input("Lütfen adınızı giriniz:")

if ad not in üyeler:
    print("Üye değilsiniz!")

else:
    print("Gizli derneğimize üyesiniz. Buyrun içeri!")
```

İlk örnekte “if ad in üyeler” ifadesiyle, “eğer ad üyeler listesi içinde varsa...” demiştik. Burada ise “eğer ad üyeler listesi içinde yoksa...” dedik... Yani cümlemize olumsuz bir anlam kattık.

4.7 Listelerin Metotları

Bu bölümde “metot” diye bir şeyden söz edeceğiz. Python’da bu “metot” denen şey bir hayli önem taşır ve epey de işe yarar. Metotlar yardımıyla pek çok karmaşık işlemi çok basit bir şekilde yerine getirebiliriz. Öncelikle elimizde kullanabileceğimiz hangi metotların olduğuna bir bakalım. Bu iş için Python’daki `dir()` fonksiyonundan yararlanabiliriz. Listelerin metotlarını sıralamak istersek, bu fonksiyonu şu şekilde kullanıyoruz:

```
>>> dir(list)
```

...veya şu da olur:

```
>>> a = []
>>> dir(a)
```

Yani önce herhangi bir liste tanımlayıp (burada biz boş bir liste tanımladık), daha sonra `dir()` fonksiyonunu bu liste üzerine de uygulayabiliriz.

Hangi yolu benimsersek benimseyelim, alacağımız çıktı şu olacaktır:

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

Burada sağında solunda “__” işareti olmayanlar bizim ilgi alanımıza giren metotlardır. Ötekiler ise bizim şu anda konumuz dışında kalan “özel metotlar”dır. Onlarla şimdilik ilgilenmeyeceğiz. Peki dir(list) komutu ile aldığımız listeyi Python programlama bilgimizi kullanarak ayıklayabilir miyiz? Yani sağında solunda “__” işareti olan metotları hiç göstermeden sadece ilgi alanımıza giren metotları alabilir miyiz? Elbette alabiliriz. Bu örnek bize iyi bir pratik yapma imkanı sağlayacak:

```
>>> for i in dir(list):
...     if "_" not in i:
...         print(i)

append
count
extend
```

```
index
insert
pop
remove
reverse
sort
```

Burada yaptığımız şey Python'a şu komutu vermek:

> *"Ey Python! dir(list) komutunun çıktısını baştan sona tara ve her bir öğeyi "i" olarak adlandır. Ardından bu "i"ler içinde "_" işaretini barındırmayanları bul ve ekrana bas!"*

Peki ilgilendiğimiz metotların kaç adet olduğunu nasıl bulursunuz? "Elle sayarım!" diyenleri şiddetle kınıyorum! Burada metot sayısı az olduğu için kod yazmak gereksiz görünebilir, ama ya yüzlerce öğe olsaydı saymamız gereken... Onları da elle mi sayacaktık? Elbette hayır! Gelin şimdi bu ilgilendiğimiz öğeleri numaralandıralım:

```
>>> a = dir(list)
>>> b = []
>>> for i in a:
...     if "_" not in i:
...         b.append(i)
...         print(len(b), i, sep=".")

1.append
2.count
3.extend
4.index
5.insert
6.pop
7.remove
8.reverse
9.sort
```

Bu kodlar içinde şimdiye kadar görmediğimiz tek şey `append()`. Öteki öğelerle daha önceki derslerimizde pek çok örnek yapmıştık.

Burada öncelikle *a = dir(list)* şeklinde bir değişken tanımladık. *dir(list)* komutunu "a" adlı bir değişkene atamak bize kullanım kolaylığı sağladı. Böylece *dir(list)* kodunu kullanmamız gereken yerlerde uzun uzun bu komutu yazmaktansa kısaca "a" değişkenini kullanabileceğiz.

Ardından *b = []* diyerek boş bir liste oluşturduk. Bu boş liste içine ilgilendiğimiz metotları yerleştireceğiz.

Bir sonraki satırda ise basit bir `for` döngüsü yazdık. Bu döngü yardımıyla *dir(list)* komutunun çıktısını baştan sona tarayıp, içinde "_" işareti geçmeyen öğeleri bulduk.

Bu satırın ardından *b.append(i)* gibi bir kod görüyoruz. `append()` listelerin metotlarından biridir. Bu metodu ve diğerlerini bir sonraki bölümde ayrıntılı olarak inceleyeceğiz. Şimdilik şu kadarını söyleyelim: Bu metot bir listeye yeni öğeler eklememizi sağlar. Zaten burada yaptığı iş de budur. *b.append(i)* komutu yardımıyla, ilk başta oluşturduğumuz "b" adlı boş listeye bütün "i"leri ekliyoruz. Bu "i"lerin ne olduğunu biliyorsunuz: Bir önceki satırda `for` döngüsü ile belirlediğimiz, "_" işareti içermeyen öğeler... Yani *dir(list)* çıktısı içinde "_" işaretini taşımayan öğeleri *b.append(i)* komutu ile "b" adlı listeye eklemiş oluyoruz.

En son satırda ise "b" adlı listenin uzunluğunu ekrana basıyoruz. Bununla birlikte, "i"leri de tek tek yazdırıyoruz.

Buradaki "sep" parçacığının işlevini biliyorsunuz. Her bir öğe arasına "." işareti koymak için bu parçacıktan yararlanıyoruz...

Kodlar arasındaki *len(b)*'nin nasıl kullanıldığına dikkat edin. *len(b)*'nin buradaki kullanımı Python'daki `for` döngülerinin işleyişi hakkında bize çok önemli bir bilgi veriyor. Dikkat ederseniz, burada *len(b)*, ekrana yazdırdığımız öğelerin sol tarafına denk gelen numaraları gösteriyor. Yani yukarıdaki çıktıda elde ettiğimiz "1, 2, 3, 4, 5, 6, 7, 8 ve 9" sayılarının ekrana basılmasından bu kod sorumlu... Peki *len(b)* kodu nasıl oluyor da bu sayıları bu şekilde ekrana basabiliyor? Bunu anlamak için `for` döngüsünün nasıl işlediğini incelememiz gerekiyor. Yukarıdaki `for` döngüsünün işleyişini adım adım görelim:

1. "a" adlı değişken içindeki bütün öğeleri tek tek tara ve bu öğelerin her birine "i" adını ver.
2. Eğer "i" adını verdiğimiz bu öğeler arasında, içinde "_" işareti geçmeyenler varsa...
3. Bu öğeleri "b" adını verdiğimiz boş listeye ekle.
4. *dir(list)* çıktısı içinde, "_" işaretini içermeyen ilk öğe "append". Dolayısıyla bunu hemen "b" adlı listeye ekle.
5. Şimdi "b" adlı listenin uzunluğunu ve öğenin kendisini ekrana bas. Bu ikisinin arasına da bir adet "." işareti koy.
6. "b" adlı listenin içinde "1" adet öğe var. Bu öğe "append" Dolayısıyla ekrana basacağın ifade şu: "1.append"
7. Şimdi tekrar dir(list) çıktısını kontrol et. Orada "_" işaretini içermeyen başka bir öğe var mı?
8. Evet var. Bu öğenin adı "count". Dolayısıyla hemen bunu da "b" adlı listeye ekle.
9. Listenin uzunluğu "2" oldu. Bu sayıyı ve yeni bulduğun öğeyi ekrana bas. Bunların arasına da bir adet "." işareti koy.
10. Dolayısıyla ekrana basacağın ifade şu: "2.count"

Python bu işlemleri *dir(list)* çıktısının hepsini kontrol edip bitirinceye kadar sürdürür ve her defasında bir işlem yaparak bize şu çıktıyı verir:

```
1.append
2.count
3.extend
4.index
5.insert
6.pop
7.remove
8.reverse
9.sort
```

Burada `append()` metodunun kullanımıyla ilgili bir örnek verdik. Yukarıdaki çıktıdan da gördüğünüz gibi, `append()`'den başka, listelerin daha pek çok metodu vardır. Bu metotlar bize yapacağımız işlerde büyük kolaylıklar sağlar. Bu dersimizde listelerin metotları konusuna sağlam bir giriş yaptığımıza göre, artık bu metotları tek tek incelemeye geçebiliriz...

4.7.1 append Metodu

Bir önceki bölümde `append()` metoduna değinmiştik. Orada söylediğimize göre bu metodun görevi bir listeye öğe eklemek idi. Zaten "append" kelimesi İngilizce'de "eklemek, iliştirmek" gibi anlamlara gelir. Hemen bu metodu nasıl kullanabileceğimizi gösteren bir örnek yapalım. Önce bir liste tanımlayalım. İsterseniz boş bir liste olsun bu:

```
>>> liste = []
```

Listemizi kontrol edelim:

```
>>> print(liste)

[]
```

Hakikaten listemiz boş!

Şimdi bu boş listeye bir öğe ekleyelim:

```
>>> liste.append("Ubuntu")
```

Şimdi listemizi tekrar kontrol edelim:

```
>>> print(liste)

['Ubuntu']
```

Bu listeye şimdi başka bir öğe daha ekleyelim:

```
>>> liste.append("Debian")
```

Listemize bakalım tekrar:

```
>>> print(liste)

['Ubuntu', 'Debian']
```

Listemizin öğelerine tek tek erişebileceğimizi biliyorsunuz:

```
>>> print(liste[0])

Ubuntu
```

...veya..

```
>>> print(liste[1])

Debian
```

Hatta liste öğelerini ters de çevirebiliriz:

```
>>> print(liste[::-1])

['Debian', 'Ubuntu']
```

Peki ya bir öğeyi değiştirmek istersek...

```
>>> liste[1] = "Gentoo"

>>> print(liste)

['Debian', 'Gentoo']
```

append() metodunu kullanarak bir listeye sadece tek bir öğe ekleyebiliriz. Yani şöyle bir şey mümkün değildir:

```
>>> liste.append("SuSe", "RedHat")

Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
TypeError: append() takes exactly one argument (2 given)
```

Bu hata çıktısına göre append() metodu yalnızca tek bir öğe alabiliyor. Biz ise 2 tane vermişiz...

Ama şöyle bir şey yapmak mümkündür:

```
>>> liste.append(["SuSe", "RedHat"])
```

Gördüğünüz gibi, “SuSe” ve “RedHat” öğelerini bir liste olarak ekledik. Yalnız, öğeleri bu şekilde eklediğimizde listenin durumu pek beklediğiniz gibi olmayabilir:

```
>>> print(liste)

['Debian', 'Gentoo', ['SuSe', 'RedHat']]
```

Gördüğünüz gibi, “SuSe” ve “RedHat” öğeleri listeye ayrı bir liste olarak eklendi. Yani aslında yine listeye tek bir öğe eklenmiş oldu:

```
>>> print(liste[2])

['SuSe', 'RedHat']
```

Demek ki listemizin üçüncü öğesi buymuş... Burada sadece “SuSe” öğesini almak isterseniz şöyle bir şey yazmanız gerektiğini biliyorsunuz:

```
>>> print(liste[2][0])

'SuSe'
```

“RedHat”i almak için ise:

```
>>> print(liste[2][1])

'RedHat'
```

Eğer append() metodunu kullanarak bir listeye birden fazla öğe eklemek isterseniz elbette çaresiz değilsiniz. for döngüsü ne güne duruyor?

```
>>> for s in ["Fedora", "Knoppix"]:
...     liste.append(s)

['Debian', 'Gentoo', ['SuSe', 'RedHat'], 'Fedora', 'Knoppix']
```

Dediğimiz gibi, for döngüsünü kullanarak bu şekilde birden fazla öğeyi listeye ekleyebilirsiniz. Ama isterseniz, bu işlem için üretilmiş ayrı bir metottan da yararlanabilirsiniz. Bu yeni metodun adı extend(). Bunu bir sonraki bölümde göreceğiz... Ama append() metodu ile ilgili olarak söylememiz gerekenler henüz bitmedi.

append() metodu liste metotları içinde en çok kullanılan metottur. Dolayısıyla listelerin başka hiçbir metodunu bilmeseniz de append() metodunu bilmeniz gerekir. Bu metodu kullanarak bazı örnekler vermeye devam edelim. Mesela şu programa bakalım:

```
tek_sayılar = []
çift_sayılar = []

başla = 1

bilgi = """Rastgele sayılar giriniz.
Her sayı girişte enter tuşuna basınız.
```

```
İşlemi bitirmek için "ç" tuşuna basınız:"""

print(bilgi)

while başla == 1:
    soru = input()

    if soru == "ç":
        başla = 0

    elif int(soru) % 2 == 0:
        çift_sayılar.append(soru)

    else:
        tek_sayılar.append(soru)

print("Girdiğiniz şu sayılar çifttir: ", çift_sayılar)
print("Girdiğiniz şu sayılar ise tektir: ", tek_sayılar)
```

Burada öncelikle “tek_sayılar” ve “çift_sayılar” adlı iki farklı liste oluşturduk. Kullanıcıdan gelecek tek ve çift sayıları ayrı ayrı listelerde depolayacağız.

Ardından “başla = 1” şeklinde bir kod yazarak “başla” değişkenini “1” olarak ayarlıyoruz. Bu değişken, programın bitişini kontrol etmemizi sağlayacak.

Daha sonra “bilgi” adlı başka bir değişken daha tanımlıyoruz. Burada kullanıcıyı programın nasıl çalıştırılacağı hakkında bilgilendiriyoruz. Buna göre kullanıcıdan rastgele sayılar girmesini, her sayıdan sonra da enter tuşuna basmasını istiyoruz. Eğer kullanıcı programdan çıkmak isterse “ç” tuşuna basacak.

Bir sonraki satırda bilgi adlı değişkeni ekrana yazdırıyoruz.

Daha sonra, programımızın sürekli olarak çalışmasını temin etmek için bir `while` döngüsü kuruyoruz. Programımızın sürekli olarak çalışması “başla” adlı değişkenin değerinin “1” olmasına bağlı. En başta, bu “başla” adlı değişkenin değerini “1” olarak ayarladığımız için programımız sürekli olarak çalışmaya devam edecektir.

Bir alttaki satırda kullanıcıdan veri almamızı sağlayacak olan `input()` fonksiyonunu yazıyoruz. Dikkat ederseniz `input()` fonksiyonunu boş bıraktık. Neden? Çünkü zaten kullanıcıya göstereceğimiz bilgiyi en başta “bilgi” adlı bir değişken tanımlayarak gösterdik. Elbette isteseydik bu `input()` fonksiyonuyla da kullanıcıya bilgi verebilirdik. Ama burada `input()` fonksiyonunu boş bırakmamızın nedeni biraz da estetik kaygılar.. Çünkü programımız içinde bir `while` döngüsü olduğu için, eğer burada `input()` fonksiyonunu boş bırakmazsak, kullanıcı her sayı girişinde bu `input()` fonksiyonu içinde tanımladığımız karakter dizisini de görecektir. İsterseniz, `input()` fonksiyonunu boş bırakmadan bir deneme yapın. Göreceğiniz gibi, kullanıcının her sayı girişinde tekrar tekrar aynı karakter dizisini görmesi pek hoş olmuyor...

Bir sonraki adımda programdan çıkış koşulumuzu belirliyoruz. Eğer kullanıcı “ç” harfine basarsa “başla” değişkeninin değeri “0” olarak ayarlanacaktır. Daha önce tanımladığımız `while` döngüsüne göre, programımızın sürekli olarak çalışması “başla” değişkeninin değerinin “1” olmasına bağlıydı. Kullanıcı “ç” tuşuna bastığında “başla” değişkeninin değeri “0” olacağı için programımız duracaktır.

Hemen ardından kullanıcıdan gelen sayıların çift olup olmadığına bakıyoruz. Burada “soru” değişkenini `int()` fonksiyonu yardımıyla sayıya çevirdiğimize dikkat edin. Bildiğiniz gibi, `input()` fonksiyonunun verdiği değerin tipi karakter dizisidir. Kullanıcıdan gelen bilgiyle aritmetik bir işlem yapabilmek için bu bilgiyi öncelikle sayıya dönüştürmemiz gerekir. Bir önceki satırda programdan çıkış koşulunu tanımlarken “soru” değişkenini sayıya dönüştürmediğimize

dikkat edin. Aksi halde kullanıcı “ç” tuşuna bastığında Python “ç” harfini sayıya dönüştürmeye çalışacak, bunu başaramayınca da hata verecektir. O yüzden orada `input()` fonksiyonuna herhangi bir dönüştürme işlemi uygulamadık... Çift sayıları belirledikten sonra “*çift_sayılar.append(soru)*” komutu yardımıyla kullanıcıdan aldığımız çift sayıları “çift_sayılar” adlı listeye ekliyoruz.

Şimdi sıra geldi tek sayıları belirlemeye. Bunu da bir `else:` bloğu yardımıyla yapıyoruz. Eğer kullanıcıdan gelen bilgi bir çift sayı değilse veya kullanıcı “ç” harfine basmamışsa kullanıcının girdiği sayı bir tek sayıdır. Bunu da “tek_sayılar” adlı listeye ekliyoruz. Tabii kullanıcı “ç” harfi dışında başka bir harfe basarsa programımız hata verecektir. Henüz bu tür durumlara karşı önlem almayı bilmiyoruz. Ama ilerde bunu nasıl yapacağımızı da öğreneceğiz. Fakat şimdilik şöyle bir kod yazarak en azından bu tür bir problemi büyük ölçüde çözmüş olursunuz:

```
tek_sayılar = []
çift_sayılar = []

kontrol = "0123456789"

başla = 1

bilgi = """Rastgele sayılar giriniz.
Her sayı girişte enter tuşuna basınız.
İşlemi bitirmek için "ç" tuşuna basınız:"""

print(bilgi)

while başla == 1:
    soru = input()

    if soru == "ç":
        başla = 0

    elif soru[0] not in kontrol:
        print("lütfen bir sayı giriniz")

    elif "." in soru:
        print("yoksa ondalık bir sayı mı girdiniz?")

    elif int(soru) % 2 == 0:
        çift_sayılar.append(soru)

    else:
        tek_sayılar.append(soru)

print("Girdiğiniz şu sayılar çifttir: ", çift_sayılar)
print("Girdiğiniz şu sayılar ise tektir: ", tek_sayılar)
```

Burada bir önceki kodlara ilave olarak bir “kontrol” değişkeni tanımlıyoruz. Bu değişken 0’dan 9’a kadar olan bütün sayıları içeriyor. Daha sonra program içinde iki adet `elif...` bloğu oluşturarak bu kontrol değişkeninin içindeki sayılara göre bazı denetlemeler yapıyoruz. Burada daha önce öğrendiğimiz `in` ve `not` parçacıklarını kullandığımıza dikkat edin. Bu parçacıklar yardımıyla, kullanıcının girdiği verilerin kontrol değişkeni içinde olup olmadığını denetliyoruz. Yani “aitlik” kontrolü yapıyoruz... Burada ayrıca kontrol değişkenimizin bir karakter dizisi olduğuna dikkat edin. Daha önce de söylediğimiz gibi, `in` ve `not` parçacıkları sadece listelere özgü bir özellik değildir. Bunları Python’daki başka öğelerle birlikte de kullanabiliriz. Dediğimiz gibi, bu kodlar da karşı karşıya olduğumuz sorunu tam anlamıyla çözmeye yetmez. Bu sorunun kökten çözümünü birkaç ders sonra göreceğiz.

Şimdi artık bir sonraki metodumuzu incelemeye geçebiliriz.

4.7.2 extend Metodu

Hatırlarsanız bir önceki bölümde append() metodunu işlerken, bu metotla bir listeye birden fazla öğe eklenemeyeceğini söylemiştik. append() metodu yardımıyla, eklemek istediğimiz birden fazla öğeyi liste haline getirip o şekilde ekleyebiliyorduk. Ama eklenen bu liste de aslında tek bir öğe olarak ekleniyordu. Mutlaka birden fazla öğe eklemek istediğimizde ise for döngülerinden yararlanıyorduk. Ama aslında for döngüsüne hiç gerek olmadan, listelerin extend() adlı metodu yardımıyla da bu isteğimizi yerine getirebiliriz. Nasıl mı? Hemen bir örnek verelim. Şöyle bir listemiz vardı:

```
>>> liste = ["Debian", "Gentoo", "Fedora", "Knoppix"]
```

Şimdi extend() metodunu kullanarak, bu listeye yeni öğeler ekleyeceğiz:

```
>>> liste.extend(["Arch", "Ubuntu", "Kubuntu", "PCLinuxOs"])

['Debian', 'Gentoo', 'Fedora', 'Knoppix', 'Arch',
 'Ubuntu', 'Kubuntu', 'PCLinuxOs']
```

Gördüğünüz gibi, append metodunun aksine, extend metodu eklediğimiz öğeleri teker teker yerleştirdi listeye. Üstelik bizi herhangi bir for döngüsü kullanmak zorunda da bırakmadı.

Aslında extend() metodunun yaptığı işi sadece "+" işaretini kullanarak da yapabilirsiniz. Mesela şu örneğe bakalım:

```
>>> liste = ["Debian", "Gentoo", "Fedora", "Knoppix",
... "Arch", "Ubuntu", "Kubuntu", "PCLinuxOs"]

>>> yeni_liste = ["Mandriva", "Sabayon", "OpenSuSe"]

>>> liste = liste + yeni_liste

>>> print(liste)

['Debian', 'Gentoo', 'Fedora', 'Knoppix', 'Arch', 'Ubuntu',
 'Kubuntu', 'PCLinuxOs', 'Mandriva', 'Sabayon', 'OpenSuSe']
```

Hangi yöntem kolayınıza geliyorsa onu seçmekte özgürsünüz...

4.7.3 insert Metodu

Python'un bize sunduğu bir başka metot da insert() adlı metottur. Bu metot yardımıyla listenin herhangi bir noktasına öğe ekleyebiliyoruz. "insert" kelime anlamı olarak "yerleştirmek" demek... İşte bu metotla istediğimiz bir öğeyi bir listenin istediğiniz noktasına yerleştirebileceğiz. Mesela bu metot yardımıyla örnek bir listenin 1. sırasına (Dikkat edin, 0'ıncı sıraya demiyoruz) istediğimiz bir öğeyi yerleştirebiliriz:

```
>>> bir_liste = ["Debian", "Gentoo", "Fedora", "Knoppix",
... "Arch", "Ubuntu", "Kubuntu", "PCLinuxOs",
... "Mandriva", "Sabayon", "OpenSuSe"]

>>> bir_liste.insert(1, "Mint")

>>> print(bir_liste)
```

```
['Debian', 'Mint', 'Gentoo', 'Fedora', 'Knoppix', 'Arch', 'Ubuntu',
 'Kubuntu', 'PCLinuxOs', 'Mandriva', 'Sabayon', 'OpenSuSe']
```

Gördüğünüz gibi listemizin birinci sırasına "Mint" adlı öğeyi yerleştirdik. Bu metot da, tıpkı append() metodunda olduğu gibi tek bir öğe yerleştirmemize izin verir. Ama tabii ki for döngülerinden yararlanarak bu kısıtlamayı aşabileceğinizi biliyorsunuz:

```
>>> liste = ["elma", "armut", "kebap"]

>>> ekle = ["lahmacun", "pide", "şeftali"]

>>> for i in ekle:
...     liste.insert(0, i)
```

Burada liste'nin 0. sırasına üç adet yeni öğe yerleştiriyoruz. Yalnız burada yeni yerleştirilen öğelerin ters çevrildiğine dikkat edin...

4.7.4 remove Metodu

Python listelere öğe eklemenin yanısıra, bize listelerden öğe silme imkanı da sağlar. Bunun için listelerin remove() metodunu kullanacağız:

```
>>> bir_liste = ["Debian", "Gentoo", "Fedora", "Knoppix",
... "Arch", "Ubuntu", "Kubuntu", "PCLinuxOs",
... "Mandriva", "Sabayon", "OpenSuSe"]

>>> bir_liste.remove("Fedora")
```

Eğer listede birden fazla "Fedora" varsa, Python soldan sağa doğru, listede bulduğu ilk "Fedora"yı listeden çıkaracaktır.

Eğer listedeki bir öğeyi adına göre değil de sırasına göre çıkarmak isterseniz şöyle bir yol izleyebilirsiniz:

```
>>> bir_liste.remove(bir_liste[3])
```

Bu kod, "bir_liste" adlı listenin 3. sırasındaki öğeyi çıkaracaktır.

4.7.5 index Metodu

Bu metot, bir listedeki öğenin sırasını bulmamızı sağlar. Hemen bir örnek verelim:

```
>>> bir_liste = ["Debian", "Gentoo", "Fedora", "Knoppix",
... "Arch", "Ubuntu", "Kubuntu", "PCLinuxOs",
... "Mandriva", "Sabayon", "OpenSuSe"]

>>> bir_liste.index("Sabayon")

9
```

Eğer listede olmayan bir öğeyi sorgularsanız Python bir hata mesajı gösterecektir.

4.7.6 sort Metodu

`sort()` metodu ile liste öğelerini alfabe sırasına dizeceğiz. Yalnız bu metot Python listelerinin en "gıcık" metotlarından biridir. Neden böyle olduğunu biraz sonra göreceğiz. Ama önce `sort()` ile ilgili bir örnek yapalım:

```
>>> liste = ["Mehmet", "Ahmet", "Cemal", "Seval", "Kezban"]

>>> liste.sort()

>>> print(liste)

['Ahmet', 'Cemal', 'Kezban', 'Mehmet', 'Seval']
```

Gördüğünüz gibi, Python bu listemizi güzel güzel alfabe sırasına koydu. Yalnız bu listede bir şey dikkatinizi çekmiş olmalı. Listedeki hiçbir isim Türkçe karakter içermiyor! Bunu bilerek böyle yaptık. Çünkü içinde Türkçe karakterler geçen kelimeleri sıralamak o kadar kolay değildir. İsterseniz deneyelim:

```
>>> liste = ["Mehmet", "Ahmet", "Şevket", "Çetin",
... "Cemal", "Seval", "Kezban"]

>>> liste.sort()

>>> print(liste)

['Ahmet', 'Cemal', 'Kezban', 'Mehmet', 'Seval',
 'Çetin', 'Şevket']
```

Gördüğünüz gibi, Python bu defa liste öğelerini alfabe sırasına dizmekte başarısız oldu. Burada "Çetin" ve "Şevket"in yerleri yanlış... Peki bunu nasıl düzelteceğiz?

Bu sorunu düzeltmek için henüz öğrenmediğimiz bazı öğeler kullanmamız gerekecek:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, "tr_TR.UTF8")

>>> liste = ["Mehmet", "Ahmet", "Şevket", "Çetin",
... "Cemal", "Seval", "Kezban"]

>>> liste.sort(key=locale.strxfrm)

>>> print(liste)
```

Buradaki bilmediğiniz öğelere takılmayın. Yeri geldiğinde bunları ayrıntılarıyla inceleyeceğiz. Burada bu kodu vermekteki amacım `sort()` metoduyla Türkçe karakter içeren sözcükleri de sıralayabileceğinizi göstermek. Yalnız yukarıda verdiğim yöntem GNU/Linux için geçerlidir. Windows'ta çalışmayabilir...

4.7.7 reverse Metodu

Bu bölümde göreceğimiz `reverse()` metodu, liste öğelerinin sırasını ters çevirmemizi sağlar. Yani bu metot yardımıyla liste öğelerini tersyüz edebiliriz:

```
>>> liste = list(range(11))

>>> print(liste)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Elbette 0'dan 10'a kadar olan sayıları içeren bir liste oluşturmak için sayıları tek tek elle yazmaya gerek yok. Bu iş için `range()` metodundan yararlanabiliyoruz... Şimdi `range()` fonksiyonundan da yardım alarak oluşturduğumuz bu listeyi ters çevirelim:

```
>>> liste.reverse()

[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Gördüğünüz gibi, `reverse()` metodu listedeki öğelerin sırasını tersine çeviriyor. Aslında biz bu işi `reverse()` metodunu kullanmadan da yapmayı öğrenmiştik:

```
>>> liste[::-1]
```

Yalnız bu yöntem ile `reverse()` metodunu kullanmak arasında önemli bir fark var. reverse() metodu, uygulandığı liste üzerinde derhal değişiklik yapıyor. Yani bir listeye `reverse()` metodunu uygulayıp liste öğelerinin sırasını değiştirdiğinizde eski listeyi kaybetmiş oluyorsunuz. Artık eski listeye tekrar ulaşmanızın imkanı yoktur. Tabii yeni listeye tekrar `reverse()` metodunu uygularsanız eski listeye benzeyen başka bir liste oluşturabilirsiniz. Bu durumun aksine, liste[::-1] komutu liste üzerinde herhangi bir değişiklik yapmaz. İsterseniz bu yöntemi uyguladıktan sonra "*print(liste)*" komutunu vererek bunu doğrulayabilirsiniz. Bu komutu verdiğinizde karşınıza öğeleri ters çevrilmemiş olan liste gelecektir. Eğer yukarıdaki yöntem yardımıyla listeyi değiştirmek isterseniz şöyle bir şey yapmanız gerekir:

```
>>> liste = liste[::-1]
```

Bu şekilde "liste" adlı listemizi değiştirebiliyoruz. Bu iki yöntem arasındaki fark önemli bir farktır. Bunu unutmamak gerekir.

4.7.8 count Metodu

Bu metot bir öğenin bir listede kaç kez geçtiğini bulmamızı sağlar. Kullanımı çok basittir:

```
>>> liste = ["elma", "armut", "elma", "kebap", "şeker",
... "armut", "çilek", "ağaç", "şeker", "kebap", "şeker"]

>>> liste.count("elma")

2
```

Peki yukarıdaki listede gördüğümüz her bir öğenin listede kaç kez geçtiğini nasıl buluruz? Tabii ki bir `for` döngüsü yardımıyla:

```
>>> for i in liste:
...     print(i, "öğesi listede", liste.count(i), "kez geçiyor...")
```

Fena değil... Ama bu kodların bir sorunu var. Her cümleyi, öğenin listede geçtiği sayı kadar tekrar ediyor. Her öğeyle ilgili cümleyi bir kez bassa ne iyi olurdu, değil mi?

Elbette bunu da yapmanın yolları vardır. Bunun en kolay yolu Python'daki `set()` fonksiyonunu kullanmaktır. Bu fonksiyonu daha sonraki bir bölümde detaylı olarak inceleyeceğiz. Şimdilik sadece nasıl kullanıldığına dikkat edelim yeter:

```
for i in set(liste):
    print(i, "öğesi listede", liste.count(i), "kez geçiyor...")
```

4.7.9 pop Metodu

pop() metodu listelerin ilginç ve sevimli bir metodudur! Bu metot bazı yönlerden remove() metoduna benzer. Tıpkı remove() metodunda olduğu gibi, pop() metoduyla da listeden öğe silebiliriz. Şu örneğe bakalım:

```
>>> liste = ["FVWM", "FVWM95", "TWM/VTWM", "MWM", "CTWM",
... "OLWM/OLVWM", "wm2/wmx", "AfterStep", "AmiWM",
... "Enlightenment", "WindowMaker", "SCWM", "IceWM",
... "Sawfish", "Blackbox", "Fluxbox", "Metacity"]

>>> liste.pop()

'Metacity'
```

Şimdi listeyi tekrar ekrana yazdırırsanız en sondaki "Metacity" öğesinin olmadığını göreceksiniz. pop() metodu bu öğeyi listeden sildi. Silerken de ekrana çıktı olarak verdi... Demek ki pop() metodu bir listedeki en son öğeyi silip, sildiği öğeyi ekranda gösteriyor. Elbette bu metodu kullanarak illa son öğeyi sileceğiz diye bir kaide yok. Eğer istersek, pop() metodu içinde, silmek istediğimiz liste öğesinin sırasını belirterek, belirli bir öğeyi de silebiliriz:

```
>>> liste.pop(0)

'FVWM'
```

Bu metodun remove() metodundan farkı, remove() metodunun liste öğelerinin adına göre, pop() metodunun ise liste öğelerinin sırasına göre öğe silmesidir... Yani şöyle:

```
>>> liste.remove("AmiWM")

>>> liste.pop(7)
```

Gördüğünüz gibi pop() metodunu kullanabilmek için sileceğimiz öğenin listede kaçıncı sırada bulunduğunu bilmemiz gerekiyor. Eğer sileceğiniz öğenin adını biliyor, ancak sırasını bilmiyorsanız şöyle bir şey yapabilirsiniz. ("IceWM" adlı öğeyi silmek istiyoruz...):

```
>>> liste.pop(liste.index("IceWM"))
```

Burada kullandığımız index() metodu "IceWM" adlı öğenin listedeki sırasını verecek, pop() metodu ise index metodundan gelen bu sayıyı kullanarak "IceWM" adlı öğeyi listeden çıkaracak ve ekrana basacaktır...

Demetler (Tuples)

Bu bölümde “tuple” diye bir şeyden söz edeceğiz. Biz buna Türkçe’de genellikle “demet” adını veriyoruz. Demetler de tıpkı listeler, sayılar ve karakter dizileri gibi, Python’daki veri tiplerinden biridir. Demetler bazı yönleriyle listelere benzerler, ama listelerle aralarında çok önemli bir fark vardır. Listeler “değiştirilebilir” (mutable) bir veri tipiyken, demetler ise “değiştirilemez” (immutable) bir veri tipidir. Peki bu ne anlama geliyor? Hatırlarsanız listeler konusunu incelerken listelerin bazı metotları olduğundan söz etmiştik. Örneğin `remove()` metodu bunlardan biriydi. Mesela bu metodu kullanarak şöyle bir şey yapabiliyorduk:

```
>>> geliştirme_araçları = ["Drpython", "Spe", "IDLE", "Komodo", "Eclipse"]

>>> geliştirme_araçları.remove("IDLE")

>>> print(geliştirme_araçları)

['Drpython', 'Spe', 'Komodo', 'Eclipse']
```

Gördüğünüz gibi `remove()` metodunu kullanarak, “geliştirme_araçları” adlı liste üzerinde değişiklik yapabildik. Mesela şu yöntemi kullanarak, liste öğelerinden birini silip yerine başka bir öğe de koyabiliriz:

```
>>> geliştirme_araçları[1] = "Kwrite"

>>> print(geliştirme_araçları)

['Drpython', 'Kwrite', 'Komodo', 'Eclipse']
```

Burada en son komutun çıktısına baktığımızda, listenin birinci sırasındaki “Spe” adlı öğenin yerine “Kwrite” öğesinin geldiğini görüyoruz. Bu durum, yani listeler üzerinde değişiklik yapabilme olanağı, listelerin “değiştirilebilir” bir veri tipi olmasından kaynaklanıyor. Mesela aynı şeyi karakter dizileri ile yapamayız:

```
>>> a = "Adana"
>>> a[3] = "m"

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Gördüğünüz gibi, böyle bir şey yapmaya çalıştığımızda Python bize bir hata mesajı gösteriyor.

Çünkü karakter dizileri, listelerin aksine değiştirilebilme kabiliyetine sahip değildir. Yani bunlar "değiştirilemeyen" (immutable) veri tipleridir. Elbette karakter dizileri üzerinde de değişiklik yapmanın yolları var. Biz "Karakter Dizileri" konusunu ayrıntılı olarak incelediğimiz zaman bütün bu özellikleri tek tek öğreneceğiz. Şimdilik konumuzdan sapmayalım...

Tıpkı karakter dizileri gibi, demetler de değiştirilemeyen veri tipleridir. Demetlere kısaca değindiğimize göre, gelin isterseniz bu "giriş" bölümünü burada sonlandırıp asıl konumuza geçelim...

5.1 Demetleri Tanımlamak

Bir önceki bölümde demetler konusuna kısaca bir giriş yapmıştık. O bölümde, demetlerin listelere benzediğini ifade etmiştik. Hatırlarsanız listeleri şöyle tanımlıyorduk:

```
>>> liste = []
```

Burada boş bir liste tanımladık. Demetler de buna benzer bir şekilde tanımlanır:

```
>>> demet = ()
```

Gördüğünüz gibi, listelerdeki köşeli parantezlerin yerini demetlerde normal parantezler alıyor... İsterseniz yukarıda tanımladığımız veri tiplerinin sağlamasını yapalım:

```
>>> type(liste)

<class 'list'>
```

Demek ki tanımladığımız şey gerçekten de bir listeymiş... Bir de şuna bakalım:

```
>>> type(demet)

<class 'tuple'>
```

Burada gördüğümüz "tuple" ifadesi Türkçe'de "demet" anlamına geliyor. Demek ki burada oluşturduğumuz veri tipi gerçekten de bir demetmiş...

Daha önce "dir(list)" komutunu kullanarak, listelerin metotlarını sıralamayı öğrenmiştik. Şimdi de demetlerin metotlarına bakalım:

```
>>> dir(tuple)

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

Daha önce listelerin metotlarını almak için kullandığımız "list" yerine burada "tuple" kelimesini yazdığımıza dikkat edin. Elbette `dir()` metodunu şu şekilde kullanabileceğimizi de tahmin ediyorsunuzdur:

```
>>> dir(demet)
```

Burada daha önce tanımladığımız "demet" adlı öğeden yararlandık. Hatta istersek şöyle bir şey bile yazabiliriz:

```
>>> dir(())
```

Burada ise, demetlerin ayırt edici özelliği olan parantezleri kullandık. Benzer bir şeyi listeler için yapmak isteseydik şöyle bir yazım tarzı benimseyecektik:

```
>>> dir([])
```

Burada da listelerin ayırt edici özelliği olan köşeli parantezlerden faydalandık...

dir(tuple) komutundan elde ettiğimiz çıktılar arasında, içinde "__" işareti olmayanlar bizim ilgi alanımıza giren metotlar oluyor. Tıpkı daha önce listeler için yaptığımız gibi burada da ufak bir-iki satırlık kod yardımıyla dir(tuple) çıktısı içinde bizim ilgimizi çeken metotları alabiliriz:

```
>>> for i in dir(tuple):
...     if "_" not in i:
...         print(i)
...
count
index
```

Demek ki demetlerin yalnızca iki adet metodu varmış. Demetlerin metot sayısı listelere kıyasla epey az. Bunun nedeni, daha önce bahsettiğimiz "değiştirilebilme/değiştirilememe" meselesidir... Demetler, listelerin aksine değiştirilemeyen veri tipleri olduğu için, doğal olarak, değişiklik yapmaya yönelik metotlara sahip değildir. Listeler ise değiştirilebilen veritipleri olduğu için, `append()`, `extend()` ve `remove()` gibi, liste üzerinde değişiklik yapmayı sağlayan metotlar barındırır. Zaten dikkat ederseniz, demetlerin sahip olduğu bu iki metot da değişiklik yapmaya değil, bir demetin öğeleri hakkında bilgi vermeye yöneliktir. dir(tuple) komutundan elde ettiğimiz çıktıda görünen `index()` ve `count()` metotlarının ne işe yaradığını az çok tahmin edersiniz. Bunları zaten listeleri işlerken de görmüştük. Burada da işlevleri aynıdır.

Peki demetler ne işe yarar? Yani listeler gibi oldukça güçlü bir veri tipi önümüzde dururken demetleri kim takar!.. Elbette Python geliştiricileri çeşit olsun diye yerleştirmemişler demet diye bir şeyi dilin içine... Demetlerin de gerekli olduğu yerler vardır.

Bir defa demetler listelere kıyasla daha hızlı çalışırlar. Dolayısıyla üzerinde değişiklik yapmanız gerekmeyecek öğeleri gruplarken listeler yerine demetleri kullanmayı tercih edebilirsiniz. Ayrıca demetler değiştirilemeyen bir veri tipi olduğu için, bu veri tipini, program boyunca değişmesini istemediğimiz öğeleri gruplamak için de kullanabiliriz. Yani bir bakıma "salt-okunur" öğeleri demetler içinde saklamayı tercih edebiliriz...

Örnek olması açısından bir demet tanımlayalım:

```
>>> personel = ("Mehmet", "Ahmet", "Zeliha")
```

Böylelikle üç öğeli bir demet tanımlamış olduk. Listelerle yapabildiğimiz bazı şeyleri demetlerle de yapabiliriz. Mesela:

```
>>> personel[0]

'Mehmet'

>>> personel[:-1]

('Mehmet', 'Ahmet')

>>> personel[::-1]

('Zeliha', 'Ahmet', 'Mehmet')
```

Ama tabii ki şöyle bir şey mümkün değildir:

```
>>> personel[1] = "İlkay"
```

Neden? Çünkü, daha önce de birkaç kez söylediğimiz gibi, demetler üzerinde değişiklik yapamayız. Demetler “değiştirilemeyen” (immutable) veri tipleridir.

Demetlerin ilginç bir özelliği daha vardır. Her ne kadar demetlerin ayırt edici özelliği “()” işareti olsa da demetleri tanımlamak için parantez kullanmak zorunda değiliz:

```
>>> a = "Ahmet", "Mehmet", "Zarife"
```

Hemen bu “a” değişkeninin tipini kontrol edelim:

```
>>> type(a)

<class 'tuple'>
```

Gördüğünüz gibi, parantezleri kullanmadan da demet tanımlayabiliyoruz. Bu demek oluyor ki, aslında bir değişkene birden fazla değer atadığımızda yaptığımız şey bir demet tanımlamak. Tek öğeli bir değişken, içinde sakladığı verinin tipiyle anılır. Yani:

```
>>> hava = "yağmurlu"

>>> type(hava)

<class 'str'>
```

Bir de şuna bakalım:

```
>>> hava = 10

>>> type(hava)

<class 'int'>
```

...ama birden fazla öğe barındıran değişkenler tip olarak birer demettir...

Mesela:

```
>>> hava = "yağmurlu", "bulutlu", "rüzgârlı"

>>> type(hava)

<class 'tuple'>
```

Ya da:

```
>>> hava = 10, 15, 17

>>> type(hava)

<class 'tuple'>
```

Elbette, eğer istersek yukarıdaki demetlerin öğelerini parantez içinde de belirtiriz. Bu şekilde hangi tür bir veri tipi tanımladığımızı daha açık ve net bir şekilde belli etmiş oluruz.

5.2 Tek Öğeli bir Demet Tanımlamak

Bir önceki bölümde bir demetin nasıl tanımlanacağını gördük. Yalnız tanımladığımız demetler hep birden fazla öğe içeriyordu. Önceki bölümde tek öğeli bir demetin nasıl tanımlanacağını öğrenmedik. "Tek bir öğe yazarsın, olur sana tek öğeli bir demet!" dediğinizi duyar gibiyim... Ama ne yazık ki durum bu kadar basit değil. Hemen durumun karmaşıklığını gösteren bir örnek yaparak işe başlayalım:

```
>>> a = ("tek")

>>> type(a)

<class 'str'>
```

Gördüğünüz gibi hüsrana uğradık!... Basitçe tek bir öğe içeren bir demet tanımlamaya çalışırken, elde ettiğimiz şey bir "str", yani "karakter dizisi" oldu... Demek ki tek bir öğeyi parantez içine almak yeterli olmuyormuş... Peki ne yapacağız? Elbette elimiz kolumuz bağlı oturmayacağız. Bu tür durumlar için Python'un özel bir çözümü vardır. Görelim:

```
>>> a = ("tek",)

>>> type(a)

<class 'tuple'>
```

Gördüğünüz gibi, "tek" adlı öğenin yanına minik bir virgül işareti koyarak amacımıza ulaşabildik... İşte Python'da tek öğeli bir demet tanımlamanın yolu budur. Gözünüze tuhaf görünmüş olabilir, ama zamanla alışırsınız...

"a" adlı demeti parantezsiz olarak kullanmak istersek de buna benzer bir yol izlememiz gerekir:

```
>>> a = "tek",

>>> type(a)

<class 'tuple'>
```

Çünkü eğer oraya o virgülü koymazsak elde edeceğimiz şey bir karakter dizisi olacaktır...

Önceki bölümlerde demetler hakkında şöyle bir şey demiştik: "Demetler değiştirilemeyen bir veri tipidir." Gerçekten de öyledir... Ama değiştirilemeyen bir veri tipi olan demetler, öğe olarak, değiştirilebilen veri tiplerini barındırabilir. Yani mesela şöyle bir şey yapabiliriz:

```
>>> a = (["elma", "armut", "erik"], 1, 2, 3)
```

Gördüğünüz gibi, "a" adlı demetimizin sıfırıncı öğesi bir liste. Buna şu şekilde erişebiliriz:

```
>>> a[0]
```

Eğer bu listenin ilk öğesine erişmek istersek ne yapmamız gerektiğini biliyorsunuz:

```
>>> a[0][0]
```

Demetimizin sıfırıncı öğesi bir liste olduğu için, listelerin sahip olduğu bütün özellikleri taşır. Yani listelerin metotlarını burada kullanabiliriz:

```
>>> a[0].append("kestane")
```

Eğer tek bir öğe içeren ve bu tek öğesi de bir liste olan bir demet tanımlamak istersek, her zamanki gibi virgülden yararlanacağız:

```
>>> a = (["elma", "armut", "erik"],)
```

5.3 Demetlerin Metotları

Daha önce de söylediğimiz gibi, demetler değiştirilemeyen veri tipleri oldukları için metot yönünden fakirdirler. Demetlerin hangi metotlara sahip olduğunu nasıl bulacağımızı biliyoruz:

```
>>> dir(tuple)
```

Bu komutun çıktısı içinde bizim ilgilendiğimiz metotlar `count()` ve `index()` metotlarıdır. Bu metotları listelerden de hatırlıyoruz. Bu metotlar listelerde ne işe yarıyorsa demetlerde de aynı işe yarar. Kısaca bakalım.

5.3.1 "count" metodu

Bu metot bir öğenin demet içinde kaç kez geçtiğini söyler bize:

```
>>> a = ("elma", "armut", "erik", "şeftali", "elma", "erik", "armut")

>>> a.count("elma")
```

5.3.2 "index" metodu

Bu metot ise bir demet öğesinin demet içindeki yerini bildirir. Yani o öğenin demet içinde kaçıncı sırada yer aldığını söyler:

```
>>> a = ("elma", "armut", "erik", "şeftali", "elma", "erik", "armut")

>>> a.index("elma")

0
```

Demek ki "elma" öğesi demet içinde "sıfırıncı" sırada bulunuyormuş. Bu arada gördüğünüz gibi, `index()` metodu bir öğeyi ilk geçtiği noktada tespit eder ve ilk bulduğu sıra numarasını verir. Dolayısıyla aynı demet içinde geçen öteki "elma" öğesinin kaçıncı sırada olduğuna bakmaz.

`index()` metodu, öğe adı dışında ikinci bir argüman daha alır. Yani parantez içinde belirttiğimiz öğe adının dışında bir sayı da belirterek, o öğenin kaçıncı tekrarının sırasını elde etmek istediğimizi belirleyebiliriz. Bu ne demek? Hemen bakalım:

```
>>> a = ("elma", "armut", "erik", "şeftali", "elma", "erik", "armut")

>>> a.index("elma", 2)

4
```

Bu defa "0" çıktısı yerine "4" çıktısını elde ettik. Parantez içinde belirttiğimiz "2" sayısı aracılığıyla Python'a şöyle dedik:

> "Ey Python! Bana demet içinde geçen ikinci "elma" öğesinin sırasını söyle!"

Bu emri alan Python da demet içindeki ilk “elma” öğesini atlayıp ikinci “elma” öğesinin sırasına baktı. Böylece bize “4” çıktısını verdi. Demek ki demet içindeki ikinci “elma” öğesinin sıra numarası 4 imiş...

Demetlerin bize sunduğu metotlar bunlardan ibarettir. Bu bölümde demetlerin `index()` ve `count()` metotlarını incelediğimize göre, artık yolumuza devam edebiliriz...

5.4 Demetleme ve Demet Çözme

Bu bölümde demetlerle ilgili olarak daha önce öğrendiğimiz bazı şeylere farklı bir gözle bakacağız. Tabii bu arada yeni şeyler de öğreneceğiz. Bu bölümün amacı, demetlerle ilgili olarak şimdiye kadar öğrendiklerimize bir nevi “ad koymak”tan ibaret olacaktır.

Bu bölümün konusu “demetleme” (tuple packing) ve “demet çözme” (tuple unpacking). Öncelikle demetlemeden bahsedelim. Aslında şimdiye kadar birkaç kez “demetleme” örneği gördük. Şimdi bu gördüğümüz şeyi iyi bir tanımlayacağız.

Demetleri nasıl tanımlayacağımızı biliyoruz:

```
>>> demet = ("elma", 10, ["su", "ekmek", "zeytin"])
```

Burada, içinde farklı veri tipleri barındıran bir demet oluşturduk. Baştan beri tekrar tekrar söylediğimiz gibi demetler değiştirilemeyen veri tipleridir. Ama yukarıdaki örnekten de göreceğiniz gibi, demetlerin içine, değiştirilebilen veri tiplerini de yerleştirebiliyoruz. Örneğin yukarıda, değiştirilebilen bir veri tipi olan listeleri demetimizin içine yerleştirdik. Demetin kendisi değiştirilemese de, demetin barındırdığı liste öğelerinin değiştirilebileceğini biliyoruz.

Yukarıda verdiğimiz örnek basit bir demet tanımlama işlemidir. Şimdi vereceğimiz örnek ise bir “demetleme” (tuple packing) işlemidir:

```
>>> demet = "elma", (1, 2, 3, 4), ["kestane", "armut", "çilek"]
```

Burada yaptığımız şey, birbirinden farklı veri tiplerini alıp tek bir demet haline getirmekten ibaret... İşte bu işleme kısaca “demetleme” adı veriliyor. Çok özel bir yanı yok yani... Bu işlemin pek özel bir yanı olmasa da bu işlemin tersi olan “demet çözme” (tuple unpacking) işleminin güzel yanları vardır:

```
>>> demet = ("elma", (1, 2, 3, 4), ["kestane", "armut", "çilek"])

>>> a, b, c = demet
```

İşte burada yaptığımız şey de “demet çözme” işlemine bir örnektir. “a, b, c, = demet” şeklinde demetimizi çözdükten sonra artık şu şekilde öğelere erişebiliriz:

```
>>> a

'elma'

>>> b

(1, 2, 3, 4)

>>> c

['kestane', 'armut', 'çilek']
```

Bu işlemi normal bir değişken tanımlayarak da yapabilirsiniz:

```
>>> demet = 1, 2, 3, 4

>>> a, b, c, d = demet
```

Tabii daha önceki bölümlerde gördüğümüz gibi, birden fazla öğeye sahip bir değişken tanımladığımızda aslında daha ziyade bir demet tanımlamış oluyoruz. İsterseniz "type(demet)" komutuyla yukarıdaki "demet" değişkeninin tipini sorgulayarak bunu kendi gözlerinizle görebilirsiniz...

5.5 Döngülenebilir Nesneleri Çözme

Bu bölümde demetlere özgü olmayan bir konuya değineceğiz. Bu konuyu buraya almamın nedeni, konu demetlere özgü olmasa da demetlerle de bağlantısının bulunması, yapısı itibariyle demetleri de ilgilendirmesidir... Ayrıca bu bölüm bir bakıma bir önceki bölümün devamı olarak da düşünülebilir. Çünkü burada da temel olarak aynı kavramlardan bahsedeceğiz.

Bölümümüzün konusu "Döngülenebilir nesneleri çözme". İngilizce'de buna "Iterable Unpacking" diyorlar. Peki ne demek bu "döngülenebilir nesneleri çözme" denen şey?

Öncelikle burada "döngülenebilir nesne" kavramından bahsetmemiz gerekiyor. İngilizce'de "iterable" olarak adlandırılan döngülenebilir nesneler kabaca "üzerinde döngü kurulabilen herhangi bir şey" olarak tarif edilebilir. Mesela karakter dizileri döngülenebilen nesnelerdir. Ama sayılar döngülenebilen nesneler değildir. Peki bir nesnenin döngülenebilip döngülenemeyeceğini nasıl anlayacağız? Çok basit. Bunun için kullanabileceğimiz ayrı bir fonksiyon bulunur Python'da... Bu fonksiyonun adı `iter()`:

```
>>> iter("elma")

<str_iterator object at 0x00E4B8F0>
```

Bu komuttan elde ettiğimiz çıktı "elma" karakter dizisinin bir "döngülenebilir nesne" olduğunu söylüyor bize. Elde ettiğimiz çıktıya göre, "elma" karakter dizisi, "bellekte bulunduğu adres 0x00E4B8F0 olan bir döngülenebilir nesne" imiş... Bir de şuna bakalım:

```
>>> iter(10)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

Burada en son satırda gördüğümüz "int object is not iterable" ifadesinden anlıyoruz ki sayılar döngülenebilir nesneler değilmiş... Dolayısıyla buradan şu sonucu çıkarıyoruz: Python'da karakter dizileri döngülenebilirken, sayılar döngülenemez. Yani karakter dizileri üzerinde mesela bir `for` döngüsü kurabiliriz, ama sayıların üzerinde bir `for` döngüsü kuramayız...

Konumuz demetler olduğuna göre, bir de demetlerin durumuna bakalım:

```
>>> iter(())

<tuple_iterator object at 0x00E4B910>
```

Burada demetlerin ayırt edici özelliği olan "()" işaretlerini kullandığımıza dikkat edin. Listeler için ise şöyle bir şey yapabiliriz:

```
>>> iter([])

<list_iterator object at 0x00E4B8F0>
```

Demek ki tıpkı demetler gibi, listeler de döngülenebilir nesnelermiş...

"Döngülenebilir Nesneleri Çözme" ifadesinin "Döngülenebilir Nesne" kısmını anladığımıza göre, gelelim bu nesneyi "çözme" meselesine... Döngülenebilir nesneleri çözmeyi aslında biliyoruz. Yukarıda dediğimiz gibi, demetler bir döngülenebilir nesnedir. Geçen bölümde demetlerin nasıl çözüleceğini (tuple unpacking) gördük. Dolayısıyla döngülenebilir nesneleri nasıl çözeceğimizi de tahmin edebiliriz... Zira demetler de bir döngülenebilir nesne olduğu için, "demet çözme" işleminin kendisi de aslında bir "döngülenebilir nesne çözme" işlemidir...

Hatırlarsanız demet çözme işlemini şu şekilde yapıyorduk:

```
>>> demet = (0, 1, 2)

>>> a, b, c = demet
```

Elbette konumuz demetler olduğu için örneğimizi önce bir demet tanımlayarak yaptık. Ama tabii ki illa demet tanımlamak zorunda değiliz. Yukarıdaki kodları şöyle de yazabiliriz:

```
>>> aralık = range(3)

>>> a, b, c = aralık
```

Öncelikle burada tanımladığımız şeyin bir demet olmadığına dikkatinizi çekmek isterim. "aralık" değişkenine `type()` fonksiyonunu uygularsanız bu verinin tipinin "tuple" olmadığını göreceksiniz...

Ayrıca burada tek tek "0, 1, 2" yazmak yerine, `range()` fonksiyonunun bize sunduğu kolaylıktan yararlandığımıza da dikkat edin...

Demetimizi veya aralığımızı tanımladıktan sonra a, b ve c değişkenlerini kullanarak demetin veya aralığın öğelerini tek tek yazdırabiliriz. Burada önemli nokta, eşitliğin sol tarafındaki nesne sayısı ile sağ tarafındaki nesne sayısının aynı olmasıdır. Yani sol tarafta "a", "b" ve "c" gibi üç adet nesneye karşılık, sağ tarafta da "0", "1" ve "2" gibi üç adet öğe var. Bu arada yukarıdaki kodları istersek şöyle de yazabiliriz elbette:

```
>>> a, b, c = range(3)
```

Bunu böyle yazdıktan sonra normal bir şekilde "a", "b" ve "c" değişkenlerini kullanarak 0'dan 3'e kadar olan sayıları ekrana dökebiliriz.

Tanımı gereği, döngülenebilir nesneleri çözebilmek için, eşitliğin sağ tarafında takdir edersiniz ki döngülenebilir bir nesne olması gerekiyor... Mesela bu bölümün başında sayıların döngülenemediğini öğrenmiştik. Dolayısıyla şöyle bir şey yazamayız:

```
>>> a, b, c = 5

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

Bu hata bize "int" nesnesinin döngülenemeyeceğini söylüyor. Buradaki sorun eşitliğin sol ve sağ tarafındaki öğe sayısının aynı olmamasından kaynaklanmıyor. Sebep "int", yani sayı veri tipinin döngülenemez bir nesne olması... Bu durumu daha net olarak şu örnekte görebiliriz:

```
>>> a, b, c = "su"

ValueError: need more than 2 values to unpack
```

Gördüğünüz gibi burada aldığımız hata farklı. Buradaki sorun eşitliğin sol ve sağ tarafında aynı sayıda öğe olmamasından kaynaklanıyor. Yoksa "su" karakter dizisi döngülenebilir bir nesne olduğu için öğe sayısını tutturmamız yeterli olacaktır. Zaten aldığımız hata, çözme işlemi için 2'den fazla değer gerektiğini söylüyor bize...

```
>>> a, b, c = "Can"
```

"Peki ben "a", "b" ve "c" değişkenlerinin üçüne birden tek değer vermek istiyorsam ne yapacağım?" diye soruyor olabilirsiniz... Yani mesela hem a, hem b, hem de c değişkeninin değerinin "Can" olmasını istersek bunu nasıl yapacağız? Şöyle yapacağız:

```
>>> a = b = c = "Can"
```

Ancak şöyle bir deneme sizi hüsrana uğratabilir:

```
>>> a = b = c = 1, 2, 3
```

Burada her bir değişken 1, 2 ve 3 sayılarının üçünü birden içine alacaktır. Yani mesela "a" komutunu verdiğimizde şöyle bir çıktı elde edeceğiz:

```
>>> a

(1, 2, 3)
```

Aldığımız bu çıktı, neden istediğimiz gibi bir çıktı elde edemediğimizi aslında açıklıyor. Dikkat ederseniz aldığımız çıktının veri tipi bir demettir. İsterseniz a değişkenine `type()` fonksiyonunu uygulayarak bunu kendiniz de doğrulayabilirsiniz. Zira yukarıda yaptığımız şey aslında bir önceki bölümde öğrendiğimiz "demetleme" (tuple packing) işleminin ta kendisidir... Demetleme işlemini yaptıktan sonra isterseniz bu demeti çözebilirsiniz de...

```
>>> bir, iki, üç = a

>>> bir

1

>>> iki

2

>>> üç

3
```

Şimdiye kadar verdiğimiz örneklerde eşitliğin sol ve sağ tarafının aynı sayıda öğe içermesi gerekiyordu. Peki ya diyelim ki elimizde şöyle bir demet var:

```
>>> fedailer = ("Ahmet", "Mehmet", "Özcan", "Kezban", "Süreyya", "Cevat")
```

Yapmak istediğimiz şey bu demeti "ilk", "orta" ve "son" olarak gruplandırmak olsun... Bu işlemi acaba nasıl yapmalıyız?

İsteğimizi gerçekleştirmek için iki yöntem var önümüzde. Birinci yöntemde Python'daki "dilimleme" (slicing) özelliğinden yararlanabiliriz:

```
>>> ilk, orta, son = fedailer[:1], fedailer[1:-1], fedailer[-1]
```

Öğeleri bu şekilde tanımladıktan sonra "ilk", "orta" ve "son" değişkenlerini kullanarak her bir grubu ayrı ayrı yazdırabiliriz.

Gelelim ikinci yönteme... Bu ikinci yöntem Python 3.0 ile birlikte gelen bir özelliktir. Python'un 3.0'dan önceki sürümlerinde bu özellik bulunmaz:

```
>>> ilk, *orta, son = fedailer

>>> ilk

'Ahmet'

>>> orta

['Mehmet', 'Özcan', 'Kezban', 'Süreyya']

>>> son

'Cevat'
```

Gördüğünüz gibi, Python 3.0 ile birlikte yukarıdaki işlevi çok daha basit ve temiz bir şekilde yerine getirebiliyoruz. Burada "*" işaretinden yararlandığımıza dikkat edin. Bu tür ifadelere Python'da "starred expressions" (yıldızlı ifadeler) adı verilir. Python 3.0 ile birlikte bu yıldızlı ifadeler daha geniş bir kullanım alanına kavuştu. Hatta `print()` fonksiyonu içinde bile bu yıldızlı ifadelerden yararlanabiliriz. Yıldızlı ifadelerin `print()` fonksiyonunda kullanımı ile ilgili örnekleri birkaç bölüm sonra göreceğiz...

Yukarıdaki örnekte kullandığımız yıldızlı ifade, "ilk" ve "son" adlı değişkenler ile belirlenen öğelerin dışında kalan öteki bütün öğeleri tanımlayabilmemizi sağladı. Ayrıca burada yıldızlı ifade ile elde ettiğimiz kısmın bir liste olduğuna da dikkat edin. Burada normal olarak liste yerine demet verilmesi beklenebilirdi, ama çıktı üzerinde daha kolay işlem yapılabilmesi için Python geliştiricileri buradaki yıldızlı ifadenin çıktısının liste veri tipinde olmasını uygun görmüşlerdir... Gerçekten de değiştirilebilir (mutable) bir veri tipi olan listeler üzerinde işlem yapmak, değiştirilemeyen (immutable) bir veri tipi olan demetler üzerinde işlem yapmaktan çok daha kolaydır.

Döngülenebilir nesneleri çözme işleminin bu genişletilmiş haline ilişkin bütün özellikler ve bu yeni işlevin gerekçeleri, http://www.python.org/dev/peps/pep-3132/ adresinde bulunan **PEP 3132**'de açıklanmıştır... Eğer bu konuyla ilgili olarak İngilizce kaynaklarda arama yapmak isterseniz, kullanmanız gereken ifade "*Extended Iterable Unpacking*" olacaktır...

Böylelikle Python'da "demetler" konusunu da bitirmiş olduk. Bu bölümde pek çok şey öğrendik. Bu öğrendiklerimizi birkaç kez gözden geçirerek bilgilerimizi pekiştirme yoluna gitmemiz, ilerde göreceğimiz konuları daha iyi kavrayabilmek bakımından oldukça faydalı olacaktır.

BÖLÜM 6

Sözlükler

Bu bölümde Python'daki en verimli ve en kullanışlı veri tiplerinden birini inceleyeceğiz. Bölümümüzün konusu sözlükler. Sözlükler Python'un bir hayli güçlü araçlarından bir tanesidir. O kadar ki, istersek sözlükleri basit bir veritabanı niyetine bile kullanabiliriz.

Şimdiye kadar şu veri tiplerini görmüştük:

1. Karakter Dizileri
2. Sayılar
3. Listeler
4. Demetler

Şimdi ise bunlara bir de sözlükleri ekleyeceğiz. Sözlükler şekil olarak öteki veri tiplerinden birazcık farklı bir görünüme sahiptir. Ayrıca mesela listeler "sıralı diziler" (ordered sequence) şeklinde tanımlanan bir gruba girerken, sözlükler "sırasız diziler" (unordered sequence) grubuna dahildir. Burada sıralı-sırasız kavramlarıyla kastettiğimiz şeyin ne olduğunu asıl konumuza geçtiğimizde öğreneceğiz.

İsterseniz lafı daha fazla dolandırmadan asıl mevzuya geçelim. Ben sizin, sözlükleri öğrendikten sonra bu veri tipini çok seveceğinize eminim...

6.1 Sözlükleri Tanımlamak

Öteki veri tiplerinde yaptığımız gibi, sözlüklerle de ilk işimiz bunları tanımlamak olacaktır. Hatırlarsanız listelerin ve demetlerin birtakım ayırt edici işaretleri vardı. Mesela listelerin ayırt edici işareti "[]" iken, demetlerin ayırt edici işareti ise "()" idi... Tıpkı listeler ve demetlerde olduğu gibi, sözlüklerin de bir ayırt edici işareti vardır:

```
>>> ilk_sözlük = {}
```

Gördüğünüz gibi, sözlükleri küme parantezleri yardımıyla öteki veri tiplerinden ayırt ediyoruz. Yukarıda boş bir sözlük tanımladık. Pythonca'da sözlükler "dict" parçacığıyla gösterilir. Gelin isterseniz basit bir tip denetlemesi yapalım:

```
>>> type({})

<class 'dict'>
```

Buradaki “dict” sözcüğü denetlediğimiz veri tipinin bir sözlük olduğuna işaret ediyor.

Şimdi boş bir sözlük yerine, içinde öğe barındıran bir sözlük tanımlamaya çalışalım:

```
>>> telefonlar = {"Ahmet": "0555 555 55 55", "Ayşe": "0212 212 12 12",
... "Veli": "0312 312 13 13"}
```

Burada sözlüğü nasıl tanımladığımıza çok dikkat edin. Dediğimiz gibi, sözlüklerin yapısı öteki veri tiplerinden biraz farklıdır. İsterseniz yukarıda tanımladığımız sözlüğü biraz sadeleştirerek incelemeyi kolaylaştıralım:

```
>>> telefonlar = {"Ahmet": "0555 555 55 55"}
```

Burada şöyle bir yapı gözümüze çarpıyor:

```
{anahtar: değer}
```

Bu yapıda “anahtar” ve “değer” kelimelerini kullanmamız tesadüfi değildir. Python programlama dilinde sözlükler bir “anahtar-değer” (key-value) çifti olarak tanımlanır. Yani her sözlükte en az bir anahtar, bir de değer bulunur. Sözlükleri oluştururken bu anahtar ve değerleri iki nokta üst üste işareti (”:”) ile ayırırız. Bizim örneğimizde anahtara karşılık gelen ifade “Ahmet”; değere karşılık gelen ifade ise “0555 555 55 55”tir. Sözlüklerin öğelerine ulaşmak için anahtarları kullanacağız. Bu açıklamaların biraz soyut gelebileceğini düşünerek, durumu somutlaştırmak için bir örnek verelim. Yukarıdaki sözlük üzerinden gidelim. Sözlüğümüzdeki “Ahmet” adlı anahtarın değerini alalım:

```
>>> telefonlar["Ahmet"]

'0555 555 55 55'
```

Gördüğünüz gibi, ortada listeler ve demetlere göre farklı bir durum var. Bir sözlüğün öğelerine ulaşma biçimimiz, demetler ve listelerdeki yapıdan farklı. Hatırlarsanız demetler ve listelerin öğelerine şöyle ulaşıyorduk:

```
liste[öğenin_sıra_numarası]
```

...veya...

```
demet[öğenin_sıra_numarası]
```

Listeler ve demetlerde, öğeye ulaşmak için o öğenin sıra numarasını verirken, sözlüklerde ise anahtarını kullanıyoruz. Çıktıda elde ettiğimiz şey ise o anahtarın değeri oluyor... Yani “Ahmet” anahtarını vererek, “0555 555 55 55” değerine ulaşıyoruz.

Hatırlarsanız giriş bölümünde sözlüklerin sırasız bir veri tipi olduğundan söz etmiştik. Sözlüklerin sırasız olmasından dolayı, öğelere erişmek için sıra numarası kullanamıyoruz. Sözlüklerin aksine listeler ve demetler sıralı birer veri tipi olduğu için, öğelere sıra numarasıyla ulaşabiliyoruz.

Sözlüklerin sırasız bir veri tipi olması kendini çıktıda da gösterecektir. Yani bir sözlüğü tanımladıktan sonra, o sözlüğü ekrana yazdırdığımızda elde ettiğimiz çıktı öğelerin sırası bakımından karışık olacaktır... Mesela şu sözlüğü ele alalım:

```
>>> harfler = {"a": 1, "b": 2, "c": 3}
```

Bu sözlüğü ekrana yazdırdığımızda şöyle bir çıktı alırız:

```
>>> harfler

{'a': 1, 'c': 3, 'b': 2}
```

Gördüğünüz gibi elde ettiğimiz çıktıda öğelerin sıralaması sözlüğü tanımlarken kullandığımız öğe sıralamasından farklı... Eğer tanımladığınız bir sözlükteki öğelerin sıralaması çıktıdakiyle aynıysa bu durum tamamen tesadüften ibarettir. Sözlükler sırasız veri tipleri olduğu için, sözlüklerle güvenli bir şekilde sıralamaya dayalı herhangi bir işlem yapamayız...

Sözlükleri tanımlamanın başka yolları da vardır. Yukarıdaki tanımlama yönteminin dışında bir de şöyle bir şey yapabiliriz:

```
>>> giyim = {}

>>> giyim["ayakkabı"] = 2
>>> giyim["elbise"] = 1
>>> giyim["gömlek"] = 4

>>> giyim

{'elbise': 1, 'ayakkabı': 2, 'gömlek': 4}
```

Gördüğünüz gibi, önce boş bir sözlük tanımladıktan sonra bu boş sözlüğe daha sonradan teker teker öğe ekleyebiliyoruz.

Eğer istersek üçüncü bir yol olarak sözlükleri "dict" parçacığını kullanarak da tanımlayabiliriz:

```
>>> dict({"a": 1, "b": 2})
```

Bu "dict" parçacığını farklı şekillerde de kullanabiliyoruz:

```
>>> dict(ahmet = "0533 344 44 44")
```

Burada da sanki bir değişken tanımlar gibi sözlük tanımladığımıza dikkat edin...

Bunun dışında, demetler ve listeleri kullanarak da bir sözlük tanımlayabiliriz:

```
>>> dict([("ahmet", "0533 444 44 44")])
```

Burada iki öğeli bir demetten oluşan bir listeyi "dict" parçacığı yardımıyla sözlüğe dönüştürüyoruz...

Gördüğünüz gibi, sözlük tanımlamak için pek çok farklı yöntem bulunuyor. Siz bunlar içinde kolayınıza geleni veya o an karşı karşıya olduğunuz duruma en uygun yöntemi benimseyeceksiniz...

6.2 Sözlüklerin Metotları

Sözlükler değiştirilebilir veri tipleridir. Dolayısıyla sözlükler metot sayısı bakımından zengin sayılır. Sözlüklerin hangi metotlara sahip olduğunu nasıl öğreneceğimizi az çok tahmin ediyorsunuzdur. Daha önce liste ve demet veri tiplerini incelerken birkaç farklı şekilde, bu veri tiplerinin sahip olduğu metotları listelemeyi öğrenmiştik. Sözlükler de benzer yöntemlerle bize metotlarını listeleme imkanı sunar. Mesela şöyle bir şey yapabiliriz:

```
>>> dir(dict)

['__class__', '__contains__', '__delattr__', '__delitem__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
 '__gt__', '__hash__', '__init__', '__iter__', '__le__', '__len__',
 '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
```

```
 '__subclasshook__', 'clear', 'copy', 'fromkeys', 'get', 'items',
 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']
```

Her zaman olduğu gibi, bu liste içinde bizim ilgileneceğimiz metotlar şunlardır:

```
>>> for i in dir(dict):
...     if "_" not in i:
...         print(i)
...
clear
copy
fromkeys
get
items
keys
pop
popitem
setdefault
update
values
```

Gördüğünüz gibi sözlüklerin 11 adet metodu var bizim ilgilendiğimiz... Sonraki sayfalarda bu metotları tek tek inceleyeceğiz. Yalnız, dikkat ederseniz yukarıdaki listeyi elde etmek için çok fazla uğraşıyoruz. İlerde kodlarımızı belli ölçütlere göre süzmek istediğimiz zaman bu isteğimizi tek satırla yerine getirmeyi de öğreneceğiz:

```
>>> [i for i in dir(dict) if "_" not in i]

['clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem',
'setdefault', 'update', 'values']
```

Python'da bu yapıya "liste üreteçleri" (list comprehensions) adı verilir. Yeri geldiğinde bu konuyu da ayrıntısıyla inceleyeceğiz. Şimdilik sadece böyle bir imkanımızın olduğunu da bilelim yeter...

Neyse... Tekrar konumuza dönelim. Sözlüklerin 11 adet metodu varmış. İsterseniz lafı hiç uzatmadan ilk metodumuzu incelemeye başlayalım...

6.2.1 clear metodu

Sözlüklerin, inceleyeceğimiz ilk metodu `clear()`. Bu kelime İngilizce'de "temizlemek" anlamına gelir. Görevi sözlükteki öğeleri temizlemektir. Yani içi dolu bir sözlüğü bu metot yardımıyla tamamen boşaltabiliriz:

```
>>> lig = {"şampiyon": "Adana Demirspor", "ikinci": "Mersin İdman Yurdu",
... "üçüncü": "Adana Gençlerbirliği"}
```

İsterseniz sözlüğümüzü boşaltmadan önce bu sözlükle biraz çalışalım:

Sözlüğümüzün öğelerine şöyle ulaşıyoruz:

```
>>> lig

{'şampiyon': 'Adana Demirspor', 'ikinci': 'Mersin İdman Yurdu',
 'üçüncü': 'Adana Gençlerbirliği'}
```

Eğer bu sözlüğün öğelerine tek tek erişmek istersek şöyle yapıyoruz:

```
>>> lig["şampiyon"]

'Adana Demirspor'

>>> lig["üçüncü"]

'Adana Gençlerbirliği'
```

Şimdi geldi bu sözlüğün bütün öğelerini silmeye:

```
>>> lig.clear()
```

Şimdi sözlüğümüzün durumunu tekrar kontrol edelim:

```
>>> lig

{}
```

Gördüğünüz gibi artık "lig" adlı sözlüğümüz bomboş. `clear()` metodunu kullanarak bu sözlüğün bütün öğelerini sildik. Ama tabii ki bu şekilde sözlüğü silmiş olmadık. Boş da olsa bellekte hâlâ "lig" adlı bir sözlük duruyor. Eğer siz "lig"i ortadan kaldırmak isterseniz "del" adlı bir parçacıktan yararlanmanız gerekir:

```
>>> del lig
```

Kontrol edelim:

```
>>> lig

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

Gördüğünüz gibi artık "lig" diye bir şey yok... Bu sözlüğü bellekten tamamen kaldırdık.

`clear()` adlı metodun ne olduğunu ve ne işe yaradığını gördüğümüze göre başka bir metoda geçebiliriz.

6.2.2 copy metodu

Diyelim ki elimizde şöyle bir sözlük var:

```
>>> hava_durumu = {"İstanbul": "yağmurlu", "Adana": "güneşli",
... "İzmir": "bulutlu"}
```

Biz bu sözlüğü kopyalamak istiyoruz. Hemen şöyle bir şey deneyelim:

```
>>> yedek_hava_durumu = hava_durumu
```

Artık elimizde aynı sözlükten iki tane var:

```
>>> hava_durumu

{'İstanbul': 'yağmurlu', 'Adana': 'güneşli', 'İzmir': 'bulutlu'}

>>> yedek_hava_durumu

{'İstanbul': 'yağmurlu', 'Adana': 'güneşli', 'İzmir': 'bulutlu'}
```

Şimdi hava_durumu adlı sözlüğe bir öğe ekleyelim:

```
>>> hava_durumu["Mersin"] = "sisli"

>>> hava_durumu

{'İstanbul': 'yağmurlu', 'Adana': 'güneşli', 'Mersin': 'sisli',
 'İzmir': 'bulutlu'}
```

Şimdi bir de yedek_hava_durumu adlı sözlüğün durumuna bakalım:

```
>>> yedek_hava_durumu

{'İstanbul': 'yağmurlu', 'Adana': 'güneşli', 'Mersin': 'sisli',
 'İzmir': 'bulutlu'}
```

Gördüğünüz gibi, hava_durumu adlı sözlüğe yaptığımız ekleme yedek_hava_durumu adlı sözlüğü de etkiledi. Hatırlarsanız buna benzer bir durumla daha önce listeleri anlatırken de karşılaşmıştık. Çünkü varolan bir sözlüğü veya listeyi başka bir değişkene atadığımızda aslında yaptığımız şey bir kopyalama işleminden ziyade bellekteki aynı nesneye gönderme yapan iki farklı isim belirlemekten ibaret. Yani sözlüğümüzü bellekteki bir nesne olarak düşünürsek, bu nesneye atıfta bulunan, "hava_durumu" ve "yedek_hava_durumu" adlı iki farklı isim belirlemiş oluyoruz. Eğer istediğimiz şey bellekteki nesneden iki adet oluşturmak ve bu iki farklı nesneyi iki farklı isimle adlandırmak ise yukarıdaki yöntemi kullanmak istemediğiniz sonuçlar doğurabilir. Yani amacınız bir sözlüğü yedekleyip orijinal sözlüğü korumaksa ve yukarıdaki yöntemi kullandıysanız, hiç farkında olmadan orijinal sözlüğü de değiştirebilirsiniz. İşte böyle durumlarda imdadımıza sözlüklerin "copy" metodu yetişecek. Bu metodu kullanarak varolan bir sözlüğü gerçek anlamda kopyalayabilir, yani yedekleyebiliriz...

```
>>> hava_durumu = {"İstanbul": "yağmurlu", "Adana": "güneşli",
... "İzmir": "bulutlu"}
```

Şimdi bu sözlüğü yedekliyoruz. Yani kopyalıyoruz:

```
>>> yedek_hava_durumu = hava_durumu.copy()
```

Bakalım hava_durumu adlı sözlüğe ekleme yapınca yedek_hava_durumu adlı sözlüğün durumu ne oluyor?

```
>>> hava_durumu["Mersin"] = "sisli"

>>> hava_durumu

{'İstanbul': 'yağmurlu', 'Adana': 'güneşli', 'Mersin': 'sisli',
 'İzmir': 'bulutlu'}
```

yedek_hava_durumu adlı sözlüğe bakalım:

```
>>> yedek_hava_durumu

{'İstanbul': 'yağmurlu', 'Adana': 'güneşli', 'İzmir': 'bulutlu'}
```

Gördüğünüz gibi bu defa sözlüklerin birinde yapılan değişiklik öbürünü etkilemedi... copy metodu sağolsun!...

6.2.3 fromkeys metodu

`fromkeys()` metodu öteki metotlardan biraz farklıdır. Bu metot mevcut sözlük üzerinde işlem yapmaz. `fromkeys()`'in görevi yeni bir sözlük oluşturmaktır. Bu metot yeni bir sözlük oluştu-

rurken listeler veya demetlerden yararlanır. Şöyle ki:

```
>>> elemanlar = "Ahmet", "Mehmet", "Can"

>>> adresler = dict.fromkeys(elemanlar, "Kadıköy")

>>> adresler

{'Ahmet': 'Kadıköy', 'Mehmet': 'Kadıköy', 'Can': 'Kadıköy'}
```

Gördüğünüz gibi öncelikle "elemanlar" adlı bir demet tanımladık. Daha sonra da "adresler" adlı bir sözlük tanımlayarak, `fromkeys()` metodu yardımıyla anahtar olarak "elemanlar" demetindeki öğelerden oluşan, değer olarak ise "Kadıköy"ü içeren bir sözlük meydana getirdik.

En başta tanımladığımız "elemanlar" demeti liste de olabilirdi. Hatta tek başına bir karakter dizisi dahi yazabilirdik oraya...

6.2.4 get metodu

Sözlük metotları içinde en faydalı metotlardan birisi bu get()'' adlı metottur. Bu metot pek çok durumda işinizi bir hayli kolaylaştırır.

Diyelim ki şöyle bir program yazdık:

```
#!/usr/bin/env python3.0

ing_sözlük = {"dil": "language", "bilgisayar": "computer", "masa": "table"}

sorgu = input("Lütfen anlamını öğrenmek istediğiniz kelimeyi yazınız:")

print(ing_sözlük[sorgu])
```

Bu programı çalıştırdığımızda eğer kullanıcı "ing_sözlük" adıyla belirttiğimiz sözlük içinde bulunan kelimelerden birini yazarsa, o kelimenin karşılığını alacaktır. Diyelim ki kullanıcımız soruya "dil" diye cevap verdi. Bu durumda ekrana "dil" kelimesinin sözlükteki karşılığı olan "language" yazdırılacaktır. Peki ya kullanıcı sözlükte tanımlı olmayan bir kelime yazarsa ne olacak? Öyle bir durumda programımız hata verecektir. Programımız için doğru yol, hata vermektense, kullanıcıyı kelimenin sözlükte olmadığı konusunda bilgilendirmektir. Bunu klasik bir yaklaşımla şu şekilde yapabiliriz:

```
ing_sözlük = {"dil": "language", "bilgisayar": "computer", "masa": "table"}

sorgu = input("Lütfen anlamını öğrenmek istediğiniz kelimeyi yazınız:")

if sorgu not in ing_sözlük:
    print("Bu kelime veritabanımızda yoktur!")

else:
    print(ing_sözlük[sorgu])
```

Ama açıkçası bu pek verimli bir yaklaşım sayılmaz. Yukarıdaki yöntem yerine sözlüklerin `get()` metodundan faydalanabiliriz. Bakalım bunu nasıl yapıyoruz:

```
ing_sözlük = {"dil": "language", "bilgisayar": "computer", "masa": "table"}

sorgu = input("Lütfen anlamını öğrenmek istediğiniz kelimeyi yazınız:")
```

```
print(ing_sözlük.get(sorgu, "Bu kelime veritabanımızda yoktur!"))
```

Gördüğünüz gibi, burada çok basit bir metot yardımıyla bütün dertlerimizi hallettik. Sözlüklerin `get()` adlı metodu, parantez içinde iki adet argüman alır. Birinci argüman sorgulamak istediğimiz sözlük öğesidir. İkinci argüman ise bu öğenin sözlükte bulunmadığı durumda kullanıcıya hangi mesajın gösterileceğini belirtir. Buna göre, yukarıda yaptığımız şey, önce "sorgu" değişkenini sözlükte aramak, eğer bu öğe sözlükte bulunamıyorsa da kullanıcıya, "Bu kelime veritabanımızda yoktur!" cümlesini göstermekten ibarettir...

Gelin isterseniz bununla ilgili bir örnek daha yapalım.

Diyelim ki bir havadurumu programı yazmak istiyoruz. Bu programda kullanıcı bir şehir adı girecek. Program da girilen şehre ait havadurumu bilgilerini ekrana yazdıracak. Bu programı klasik yöntemle şu şekilde yazabiliriz:

```
#!/usr/bin/env python3.0

soru = input("Şehrinizin adını tamamı küçük harf olacak şekilde yazın:")

if soru == "istanbul":
    print("gök gürültülü ve sağanak yağışlı")

elif soru == "ankara":
    print("açık ve güneşli")

elif soru == "izmir":
    print("bulutlu")

else:
    print("Bu şehre ilişkin havadurumu bilgisi bulunmamaktadır.")
```

Yukarıdaki, gayet geçerli bir yöntemdir. Ama biz istersek bu kodları "get" metodu yardımıyla çok daha verimli ve sade bir hale getirebiliriz:

```
#!/usr/bin/env python3.0

soru = input("Şehrinizin adını tamamı küçük harf olacak şekilde yazın:")

cevap = {"istanbul": "gök gürültülü ve sağanak yağışlı",
         "ankara": "açık ve güneşli", "izmir": "bulutlu"}

print(cevap.get(soru,"Bu şehre ilişkin havadurumu bilgisi bulunmamaktadır."))
```

Böylelikle bir metodu daha geride bıraktık... Gelelim sıradaki metodumuza...

6.2.5 items metodu

Bu metot yardımıyla bir sözlüğün bütün öğelerine ulaşabiliriz. Yalnız bu metot bir sözlüğün öğelerini doğrudan vermez. `items()` metodunu yalın bir şekilde kullandığımızda elde ettiğimiz şeye Python dilinde "görünüm nesnesi" (view object) adı verilir. Örnek verelim:

```
>>> sepet = {"meyveler": ("elma", "armut"), "sebzeler": ("pırasa", "fasulye")}

>>> sepet.items()

<dict_items object at 0xb7cee95c>
```

İşte burada elde ettiğimiz çıktıya "görünüm nesnesi" adı veriyorlar. Bu nesneler dinamik bir yapıdadır. Peki bu ne anlama geliyor? Hemen bakalım:

```
>>> a = sepet.items()

>>> len(a)

2
```

Buradan, sepet sözlüğündeki öğe sayısının iki olduğunu görüyoruz. Şimdi sepet sözlüğüne ekleme yapalım:

```
>>> sepet["içecekler"] = ("su", "ayran", "şalgam")
```

Şimdi "a" adlı değişkenin uzunluğunu tekrar kontrol edelim:

```
>>> len(a)

3
```

Gördüğünüz gibi, sepet adlı sözlüğün içeriği değişince "sepet.items()" metodunun içeriği de otomatik olarak güncellendi. Bu, Python 3.0'la gelen bir özelliktir. Python'un 2.x sürümlerinde `items()` metodu daha farklı bir yapıya sahipti ve yukarıdaki gibi otomatik güncellenme gibi bir özellik de taşımıyordu...

Peki biz bu `items()` metoduyla elde ettiğimiz öğeleri nasıl göreceğiz? Bunun birkaç yolu vardır. En basitinden bu metot üzerinde bir döngü kurabiliriz:

```
>>> for i in sepet.items():
...     print(i)
...
('içecekler', ('su', 'ayran', 'şalgam'))
('meyveler', ('elma', 'armut'))
('sebzeler', ('pırasa', 'fasulye'))
```

Veya `list()` ya da `tuple()` metotlarını kullanarak da sepet.items() metodunun içeriğini görebiliriz:

```
>>> list(sepet.items())

>>> tuple(sepet.items())
```

Hatırlarsanız bu yöntemi daha önce `range()` fonksiyonuyla birlikte de kullanmıştık...

6.2.6 keys metodu

Bir önceki bölümde incelediğimiz `items()` metodu bir sözlüğün bütün öğelerini almamızı sağlıyordu. Bu bölümde göreceğimiz `keys()` metodu ise bir sözlüğün yalnızca anahtarlarını almamımızı sağlayacak. Bu metot da tıpkı `items()` metodu gibi yalın olarak kullanıldığında bir görünüm nesnesi verir. Elde edilen bu görünüm nesnesi de tıpkı `items()` metoduyla elde ettiğimiz görünüm nesnesi gibi dinamik yapıdadır. Yani sözlüğün anahtarlarında bir değişiklik olduğunda bu görünüm nesnesi de otomatik olarak güncellenir. Şimdi bu `keys()` metodunu nasıl kullanacağımıza bakalım. Daha önce tanımlamış olduğumuz sepet sözlüğü üzerinden gidelim isterseniz:

```
>>> sepet = {"meyveler": ("elma", "armut"), "sebzeler": ("pırasa", "fasülye")}
```

Şimdi bu sözlükteki anahtarları alalım:

```
>>> anahtarlar = sepet.keys()
```

Şimdi de bu öğeleri ekrana yazdıralım:

```
>>> list(anahtarlar)

['meyveler', 'sebzeler']
```

Eğer anahtarları liste olarak değil de demet olarak almak istersek şu komutu kullanacağız:

```
>>> tuple(anahtarlar)

('meyveler', 'sebzeler')
```

Alternatif olarak, sepet.keys() metodu üzerinde bir `for` döngüsü kurmayı da tercih edebiliriz. Bu şekilde elde ettiğimiz çıktı görünüş olarak biraz daha temiz olacaktır:

```
>>> for i in sepet.keys():
...     print(i)
...
meyveler
sebzeler
```

Gördüğünüz gibi `keys()` metodunu kullanmak da oldukça kolaydır.

6.2.7 values metodu

Bu metot, bir önceki bölümde gördüğümüz `keys()` metodunun yaptığı işin tam tersini yapar. `keys()` metoduyla bir sözlüğün anahtarlarını alıyorduk. `values()` metoduyla ise sözlüğün değerlerini alacağız:

```
>>> sepet = {"meyveler": ("elma", "armut"), "sebzeler": ("pırasa", "fasulye")}

>>> degerler = sepet.values()

>>> for i in degerler:
...     print(i)
...
('elma', 'armut')
('pirasa', 'fasulye')
```

6.2.8 pop metodu

Bu metodu listelerden hatırlıyoruz. Bu metot listelerle birlikte kullanıldığında, listenin en son öğesini silip, silinen öğeyi de ekrana basıyordu. Eğer bu metodu bir sıra numarası ile birlikte kullanırsak, listede o sıra numarasına karşılık gelen öğe siliniyor ve silinen bu öğe ekrana basılıyordu. Bu metodun sözlüklerdeki kullanımı da az çok buna benzer. Ama burada farkı olarak, pop metodunu argümansız bir şekilde kullanamıyoruz. Yani pop metodunun parantezi içinde mutlaka bir sözlük öğesi belirtmeliyiz:

```
>>> sepet = {"meyveler": ("elma", "armut"), "sebzeler": ("pırasa", "fasulye"),
... "içecekler": ("su", "kola", "ayran")}

>>> sepet.pop("meyveler")
```

Bu komut, sözlükteki “meyveler” anahtarını silecek ve sildiği bu öğenin değerini ekrana basacaktır. Eğer silmeye çalıştığımız anahtar sözlükte yoksa Python bize bir hata mesajı gösterecektir:

```
>>> sepet.pop("tatlılar")

KeyError: 'tatlilar'
```

Bir program yazarken böyle bir durumla karşılaşmak istemeyiz çoğu zaman. Yani bir sözlük içinde arama yaparken, aranan öğenin sözlükte bulunmadığı bir durumda kullanıcıya mekanik ve anlamsız bir hata göstermek yerine, daha anlaşılır bir mesaj iletmeyi tercih edebiliriz. Hatırlarsanız sözlüklerin `get()` metodunu kullanarak benzer bir şey yapabiliyorduk. Şu anda incelemekte olduğumuz `pop()` metodu da bize böyle bir imkan verir. Bakalım:

```
>>> sepet.pop("tatlılar", "Silinecek öğe yok!")
```

Böylelikle sözlükte bulunmayan bir öğeyi silmeye çalıştığımızda Python bize bir hata mesajı göstermek yerine, “Silinecek öğe yok!” şeklinde daha anlamlı bir mesaj verecektir...

6.2.9 popitem metodu

`popitem()` metodu da bir önceki bölümde öğrendiğimiz `pop()` metoduna benzer. Bu iki metodun görevleri hemen hemen aynıdır. Ancak `pop()` metodu parantez içinde bir parametre alırken, `popitem()` metodunun parantezi boş, yani parametresiz olarak kullanılır. Bu metot bir sözlükten rastgele öğeler silmek için kullanılır. Daha önce de pek çok kez söylediğimiz gibi, sözlükler sırasız veri tipleridir. Dolayısıyla `popitem()` metodunun öğeleri silerken kullanabileceği bir sıra kavramı yoktur. Bu yüzden bu metot öğeleri rastgele silmeyi tercih eder...

```
>>> sepet = {"meyveler": ("elma", "armut"), "sebzeler": ("pırasa", "fasulye")}

>>> sepet.popitem()
```

Bu komut sözlükten rastgele bir anahtarı, değerleriyle birlikte sözlükten silecektir. Eğer sözlük boşsa bu metot bize bir hata mesajı gösterir.

6.2.10 setdefault metodu

Bu metot epey enteresan, ama bir o kadar da yararlı bir mahluktur... Bu metodun ne işe yaradığını doğrudan bir örnek üzerinde görelim:

```
>>> sepet = {"meyveler": ("elma", "armut"), "sebzeler": ("pırasa", "fasulye")}

>>> sepet.setdefault("içecekler", ("su", "kola"))
```

Bu komut yardımıyla sözlüğümüz içinde “içecekler” adlı bir anahtar oluşturduk. Bu anahtarın değeri ise (“su”, “kola”) oldu... Bir de şuna bakalım:

```
>>> sepet.setdefault("meyveler", ("erik", "çilek"))

('elma', 'armut')
```

Gördüğünüz gibi, sözlükte zaten “meyveler” adlı bir anahtar bulunduğu için, Python aynı adı taşıyan ama değerleri farklı olan yeni bir “meyveler” anahtarı oluşturmadı... Demek ki bu metot yardımıyla bir sözlük içinde arama yapabiliyor, eğer aradığımız anahtar sözlükte yoksa, setdefault() metodu içinde belirttiğimiz özellikleri taşıyan yeni bir anahtar-değer çifti oluşturabiliyoruz.

6.2.11 update metodu

İnceleyeceğimiz son metot update() metodu... Bu metot yardımıyla oluşturduğumuz sözlükleri yeni verilerle güncelleyeceğiz. Diyelim ki elimizde şöyle bir sözlük var:

```
>>> stok = {"elma": 5, "armut": 10, "peynir": 6, "sosis": 15}
```

Stoğumuzda 5 adet elma, 10 adet armut, 6 kutu peynir, 15 adet de sosis var. Diyelim ki daha sonraki zamanlarda bu stoğa mal giriş-çıkışı oldu ve stoğun son hali şöyle:

```
>>> yeni_stok = {"elma": 3, "armut": 20, "peynir": 8, "sosis": 4, "sucuk": 6}
```

Yapmamız gereken şey, stoğumuzu yeni bilgilere göre güncellemek olacaktır. İşte bu işlemi update() metodu ile yapabiliriz:

```
>>>updatestok.update(yeni_stok)

>>> print(stok)

{'peynir': 8, 'elma': 3, 'sucuk': 6, 'sosis': 4, 'armut': 20}
```

Böylelikle malların son miktarlarına göre stok bilgilerimizi güncellemiş olduk...

6.2.12 Sözlük Öğelerini Alfabe Sırasına Dizmek

Daha önce de dediğimiz gibi sözlükler sıralı olmayan veri tipleridir. Dolayısıyla bir sözlüğü ekrana yazdırırken elde edeceğiniz çıktıdaki öğe sıralaması, sözlüğü tanımlarken kullandığınız öğe sıralamasından farklı olacaktır. Mesela:

```
>>> harfler = {"a": 1, "b": 5, "c": 8, "d": 10, "e": "30"}

>>> harfler

{'a': 1, 'c': 8, 'b': 5, 'e': '30', 'd': 10}
```

Gördüğünüz gibi, öğelerin sırası çıktıda karışık görünüyor. Peki biz bu sözlüğün öğelerini sıralı olarak çıktı almak istersek ne yapmamız gerekir? Python böyle bir şeyi yapmak için özel bir araç sunmaz. Dolayısıyla bizim bu işlem için başka yollar aramamız gerekecek.

En basit şekilde yukarıdaki sözlüğü şöyle sıralayabiliriz:

```
>>> for i in sorted(harfler):
...     print(i, harfler[i])
...
a 1
b 5
c 8
d 10
e 30
```

Burada dikkat ederseniz yeni bir fonksiyonla karşılaştık. Bu yeni fonksiyonun adı `sorted()`. Bu fonksiyon, işlev olarak listelerin `sort()` metoduna çok benzer. Ancak `sort()` metodu yalnızca listeler için geçerlidir. Yani mesela `sort()` metodunu demetlerle veya sözlüklerle birlikte kullanamayız. Ama `sorted()` fonksiyonu sıralanabilen bütün veritipleri için kullanılabilir. Bu bakımdan `sorted()` fonksiyonu `sort()` metoduna göre çok daha genel bir araçtır ve çok daha kullanışlıdır. Mesela `sorted()` fonksiyonuyla bir örnek daha yapalım:

```
>>> demet = ("mehmet", "ahmet", "kezban", "celal")

>>> sorted(demet)

['ahmet', 'celal', 'kezban', 'mehmet']
```

Gördüğünüz gibi, `sorted()` fonksiyonu yardımıyla demetin öğelerini alfabe sırasına dizebildik. Bu fonksiyonu listelerle birlikte de kullanabilirsiniz...

Eğer sıralamak istediğiniz demet, liste veya sözlükte Türkçe karakterler varsa, bu metot o demet, liste veya sözlüğü düzgün bir şekilde sıralayamayacaktır. Hatırlarsanız listelerin `sort()` metodunu incelerken de böyle bir durumla karşılaşmıştık. Orada kullandığımız yöntemi burada da kullanabiliriz:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, "tr_TR.UTF8")

>>> demet = ("mehmet", "ahmet", "kezban", "celal", "çiğdem")

>>> print(sorted(demet, key=locale.strxfrm))

['ahmet', 'celal', 'çiğdem', 'kezban', 'mehmet']
```

Neyse... Biz konumuza dönelim. Yukarıda `sorted()` fonksiyonunu kullanarak sözlük öğelerini sıraladık ve şöyle bir çıktı elde ettik:

```
a 1
b 5
c 8
d 10
e 30
```

Bu çıktının görüntüsünü hoşunuza gitmediyse, bu çıktıyı biraz biçimlendirmek de elbette mümkündür. Mesela her öğenin arasına, bu öğelerin birbiriyle bağlantılı olduğunu gösterecek bir işaret yerleştirebiliriz:

```
>>> for i in sorted(harfler):
...     print(i, harfler[i], sep = " --> ")
...
a --> 1
b --> 5
c --> 8
d --> 10
e --> 30
```

Gördüğünüz gibi, sözlük öğelerini alfabe sırasına dizmek o kadar da zor değil. İsterseniz yukarıdaki döngüyü nasıl kurduğumuzu biraz inceleyelim:

İlk satırda bildiğimiz şekilde bir for döngüsü oluşturuyoruz. Burada bir yenilik olarak `sorted()` fonksiyonunu öğrendik. Bu fonksiyon bize öğeleri sıralama imkanı tanıyor:

```
>>> for i in sorted(harfler):
...
```

İkinci satırda gördüğümüz kodlar biraz kafanızı karıştırmış olabilir. Belki ilk bakışta bu satırın ne işe yaradığını anlamak çok kolay olmayabilir. Ama aslında hem çok kolay, hem de çok mantıklı bir koddur bu. İsterseniz kodlarımızı şu şekilde sadeleştirerek olayların arka yüzünü görmeye çalışabiliriz:

```
>>> for i in sorted(harfler):
...     print(i)
...
a
b
c
d
e
```

Burada "sorted(sözlük)" ifadesinin nasıl bir çıktı verdiğine baktık. Gördüğünüz gibi, bu ifade, sözlüğün anahtarlarını alfabe sırasına diziyor. Ama sözlükteki değerlerle ilgili herhangi bir işlem yapmıyor. Yani bu ifade, "anahtar-değer" çiftlerinden oluşan sözlüğümüz içinde sadece "anahtar" kısmıyla ilgileniyor, değer kısmını es geçiyor. Bizim bir yolunu bulup, `sorted()` ile elde ettiğimiz sıralı anahtarları, bunlara karşılık gelen değerlerle eşleştirmemiz gerekiyor. Hatırlarsanız bir sözlüğün öğelerine şu şekilde ulaşıyorduk:

```
>>> harfler[anahtar]
```

Bunu kendi sözlüğümüze uyarlayalım:

```
>>> harfler["a"]

1

>>> harfler["b"]

5
```

İşte yukarıdaki örnekte de sözlüklerin bu özelliğinden faydalandık. Şu satırı tekrar gözümüzün önüne getirelim:

```
>>> print(i, harfler[i])
```

Burada dikkat ederseniz, `for` döngüsü içinde "i" adını vererek aldığımız bütün sözlük anahtarlarını tek tek sözlük[anahtar] şeklinde uyguluyoruz. Kodlarımıza tekrar bütün olarak bakalım:

```
>>> for i in sorted(harfler):
...     print(i, harfler[i], sep = " --> ")
```

Burada önce sözlükteki anahtarları alfabe sırasına dizip her birine "i" adını veriyoruz. Daha sonra `print()` fonksiyonu içinde öncelikle bu "i"leri ekrana basıyoruz. Bu "i"lerle birlikte ayrıca "sözlük[i]" kodunu kullanarak, "i" adını verdiğimiz her bir anahtarın değerini de çağırıyoruz. Satırın en sonundaki "sep" parçacığının görevini biliyorsunuz zaten. Bu parçacık burada bize "estetik" bir fayda sunuyor... Ve nihayet şu çıktıyı elde ediyoruz:

```
a --> 1
b --> 5
c --> 8
d --> 10
e --> 30
```

6.2.13 Sözlükten Öğe Silmek

Sözlüklerin metotlarını işlerken `pop()` ve `popitem()` adlı metotları görmüştük. Bu metotlar yardımıyla sözlükteki öğeleri silebiliyorduk. Yalnız bu metotlar silinen öğeleri de otomatik olarak ekrana basıyorlardı. Bizim ihtiyacımız olan şey listelerdeki `remove()` metodu gibi bir şey olabilir. Yani silinen öğeyi dünya aleme duyurmadan silmek istiyor olabiliriz, değil mi?

Eğer böyle bir isteğimiz varsa bunu daha önce bir kez gördüğümüz `del` deyimi yardımıyla yapabiliriz. Hatırlarsanız `del` deyimini bütün bir listeyi ortadan kaldırmak için kullanmıştık. İstersek bu `del` deyimini sözlükteki tek bir öğeyi silmek için de kullanabiliriz:

```
>>> stok = {'peynir': 8, 'elma': 3, 'sucuk': 6, 'sosis': 4, 'armut': 20}

>>> del stok["peynir"]
```

Bu şekilde stoktaki "peynir" anahtarını değeriyle birlikte silmiş olduk...

Daha önceki konulardan hatırladığımız bir de `clear()` metodu var. Bu metot ise bütün bir listeyi boşaltmak için kullanılıyor. `clear()` metodunun görevi elimize boş bir liste vermektir.

BÖLÜM 7

Kümeler

Yeni konumuz Python'da kümeler. İngilizce'de buna "set" diyorlar... Kümeler Python'da nispeten yeni sayılabilecek bir araçtır ve hem çok hızlıdır, hem de epey işe yarar...

Kümeleri öğrendiğimizde, bunların kimi zaman hiç tahmin bile edemeyeceğimiz yerlerde işimize yaradığını göreceğiz. Normalde uzun uzun kod yazmayı gerektiren durumlarda kümeleri kullanmak, bir-iki satırla işimizi halletmemizi sağlayabilir.

Kümeler, matematikten bildiğimiz "küme" kavramının sahip olduğu bütün özellikleri taşır. Yani kesişim, birleşim ve fark gibi özellikler Python'daki "set"ler için de geçerlidir.

Bu arada hatırlarsanız listelerin `count()` metodunu anlatırken de kümelerden faydalanmıştık. O bölümde şöyle bir örnek vermiştik:

```
>>> for i in set(liste):
...     print(i, "öğesi listede", liste.count(i), "kez geçiyor...")
```

Bu örneği verdiğimizde, `set()` fonksiyonunu henüz öğrenmediğimizi, ama hiç endişe etmememiz gerektiğini, zira bunu sonraki bölümlerde ayrıntılı olarak işleyeceğimizi söylemiştik. İşte artık bu kümeler konusunu ayrıntılı olarak öğrenme zamanı geldi. İsterseniz şimdi hiç vakit kaybetmeden işimize bakalım...

7.1 Küme Oluşturmak

Kümelerin bize sunduklarından faydalanabilmek için elbette öncelikle bir küme oluşturmamız gerekiyor. Küme oluşturmak çok kolay bir işlemdir:

```
>>> kume = set(["elma", "armut", "kebap"])
```

Böylelikle ilk kümemizi başarıyla oluşturduk. Dikkat ederseniz, küme oluştururken listelerden faydalandık. Gördüğünüz gibi `set()` fonksiyonu içindeki öğeler bir liste içinde yer alıyor. Dolayısıyla yukarıdaki tanımlamayı şöyle de yapabiliriz:

```
>>> liste = ["elma", "armut", "kebap"]

>>> kume = set(liste)
```

Bu daha temiz bir görüntü oldu. Elbette küme tanımlamak için mutlaka liste kullanmak zorunda değiliz. İstersek demetleri de küme haline getirebiliriz:

```
>>> demet = ("elma", "armut", "kebap")

>>> kume = set(demet)
```

Hatta ve hatta karakter dizilerinden dahi küme yapabiliriz:

```
>>> kardiz = "Python Programlama Dili için Türkçe Kaynak"

>>> kume = set(kardiz)
```

Kullandığımız karakter dizisinin böyle uzun olmasına da gerek yok. Tek karakterlik dizilerden bile küme oluşturabiliriz:

```
>>> kardiz = "a"

>>> kume = set(kardiz)
```

Ama sayılardan küme oluşturamayız:

```
>>> n = 10

>>> kume = set(n)

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

Burada kümelerle ilgili çok önemli bir bilgi ediniyoruz. Dikkat ederseniz yukarıda aldığımız hata mesajı, "Döngülenebilir Nesneleri Çözme" konusundan bahsederken, sayıları kullandığımızda aldığımız hatayla aynı. Buradan anlıyoruz ki, `set()` fonksiyonu parametre olarak bir döngülenebilir nesne istiyor... Sayılar döngülenebilir nesneler olmadığı için, bunları kullanarak küme oluşturamıyoruz. Bu da aklımızın bir kenarında bulunsun...

Peki sözlükleri kullanarak küme oluşturabilir miyiz? Evet, neden olmasın? Sözlükler de birer döngülenebilir nesne olduğuna göre, bunlardan küme oluşturmamızın önünde hiçbir engel yok:

```
>>> bilgi = {"işletim sistemi": "GNU", "sistem çekirdeği": "Linux",
... "dağıtım": "Ubuntu GNU/Linux"}

>>> kume = set(bilgi)
```

Böylece kümeleri nasıl oluşturacağımızı öğrendik. Eğer oluşturduğunuz kümeyi ekrana yazdırmak isterseniz, ne yapacağınızı biliyorsunuz. Burada tanımladığınız "küme" değişkenini kullanmanız yeterli olacaktır:

```
>>> kume

{'işletim sistemi', 'sistem çekirdeği', 'dağıtım'}
```

Tabii burada ben sizin etkileşimli kabukta çalıştığınızı varsayıyorum. Eğer yukarıdaki örneği bir metin düzenleyici kullanarak kaydedecekseniz "küme" yerine "print(küme)" ifadesini kullanmanız gerektiğini hatırlatmaya gerek yok sanırım...

Bir de şuna bakalım:

```
>>> kardiz = "Python Programlama Dili"

>>> kume = set(kardiz)

>>> print(kume)
```

```
{'a', ' ', 'D', 'g', 'i', 'h', 'm', 'l', 'o', 'n', 'P', 'r', 't', 'y'}
```

Burada bir şey dikkatinizi çekmiş olmalı. Bir karakter dizisini küme olarak tanımlayıp ekrana yazdırdığımızda elde ettiğimiz çıktı, o karakter dizisi içindeki her bir karakteri tek bir kez içeriyor. Yani mesela "Python Programlama Dili" içinde iki adet "P" karakteri var, ama çıktıda bu iki "P" karakterinin yalnızca biri görünüyor. Buradan anlıyoruz ki, kümeler aynı öğeyi birden fazla tekrar etmez. Bu çok önemli bir özelliktir ve pek çok yerde işimize yarar. Aynı durum karakter dizisi dışında kalan öteki veri tipleri için de geçerlidir. Yani mesela eğer bir listeyi küme haline getiriyorsak, o listedeki öğeler küme içinde yalnızca bir kez geçecektir. Listede aynı öğeden iki-üç tane bulunsa bile, kümemiz bu öğeleri teke indirecektir... Öğrendiğimiz bu bilgi sayesinde listelerin count metodunu anlatırken verdiğimiz örneği çok daha iyi kavrayabilirsiniz. İsterseniz o örneğe bir kez daha bakalım:

```
>>> liste = ["elma", "armut", "elma", "kebap", "şeker", "armut",
... "çilek", "ağaç", "şeker", "kebap", "şeker"]

>>> for i in set(liste):
...     print(i, "öğesi listede", liste.count(i), "kez geçiyor...")

elma öğesi listede 2 kez geçiyor...
şeker öğesi listede 3 kez geçiyor...
kebap öğesi listede 2 kez geçiyor...
ağaç öğesi listede 1 kez geçiyor...
çilek öğesi listede 1 kez geçiyor...
armut öğesi listede 2 kez geçiyor...
```

Burada "set(liste)" ifadesini kullanarak, liste öğelerini "eşşiz ve benzersiz" hale getiriyoruz... Dolayısıyla özgün listede "şeker" öğesi üç kez geçtiği halde, elde ettiğimiz çıktıda "şeker" öğesi bir kez geçiyor... Bu da "şeker öğesi listede 3 kez geçiyor..." cümlesinin çıktıda üç kez tekrarlanmasını engelliyor... Dolayısıyla, aynı öğeden birden fazla sayıda içeren listeler, sözlükler veya başka veri tiplerini, her bir öğeyi tek sayıda içerecek hale getirmenin en kestirme yolu kümeleri kullanmaktır. Neyse... Biz konumuza devam edelim...

Esasında tek bir küme pek bir işe yaramaz. Kümeler ancak birden fazla olduğunda bunlarla yararlı işler yapabiliriz. Çünkü kümelerin en önemli özelliği, başka kümelerle karşılaştırılabilme kabiliyetidir. Yani mesela kümelerin kesişimini, birleşimini veya farkını bulabilmek için öncelikle elimizde birden fazla küme olması gerekiyor. İşte biz de şimdi bu tür işlemleri nasıl yapacağımızı öğreneceğiz. O halde hiç vakit kaybetmeden yolumuza devam edelim.

7.2 Kümelerin Metotları

Daha önceki veri tiplerinde olduğu gibi, kümelerin de metotları vardır. Artık biz bir veri tipinin metotlarını nasıl listeleyeceğimizi çok iyi biliyoruz. Nasıl liste için `list()`; demet için `tuple()`; sözlük için de `dict()` gibi parçacıklar kullanıyorsak, kümeler için de `set()` adlı parçacıktan yararlanacağız:

```
>>> dir(set)

['__and__', '__class__', '__contains__', '__delattr__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__iand__', '__init__', '__ior__', '__isub__', '__iter__',
'__ixor__', '__le__', '__len__', '__lt__', '__ne__', '__new__',
'__or__', '__rand__', '__reduce__', '__reduce_ex__', '__repr__',
'__ror__', '__rsub__', '__rxor__', '__setattr__', '__sizeof__',
```

```
'__str__', '__sub__', '__subclasshook__', '__xor__', 'add',
'clear','copy', 'difference', 'difference_update', 'discard',
'intersection', 'intersection_update', 'isdisjoint', 'issubset',
'issuperset', 'pop', 'remove', 'symmetric_difference',
'symmetric_difference_update', 'union', 'update']
```

Hemen işimize yarayacak metotları alalım:

```
>>> for i in dir(set):
...     if "__" not in i:
...         print(i)
...
add
clear
copy
difference
difference_update
discard
intersection
intersection_update
isdisjoint
issubset
issuperset
pop
remove
symmetric_difference
symmetric_difference_update
union
update
```

Gördüğünüz gibi kümelerin epey metodu var. Bu arada *if "__" not in i* satırında "_" yerine "__" kullandığımıza dikkat edin... Neden? Çünkü eğer sadece "_" kullanırsak "symmetric_difference" ve "symmetric_difference_update" metotları çıktımızda yer almayacaktır...

Unutmadan söyleyelim: Kümeler de, tıpkı listeler gibi, değiştirilebilir (mutable) bir veri tipidir...

7.2.1 clear metodu

Kümelerle ilgili olarak inceleyeceğimiz ilk metot `clear()`. Bu metodu daha önce sözlükleri çalışırken de görmüştük. Sözlüklerde bu metodun görevi sözlüğün içini boşaltmak idi... Burada da aynı vazifeyi görür:

```
>>> tr = set("şçöğüı")

>>> tr

{'ç', 'ı', 'ş', 'ö', 'ü', 'ğ'}

>>> tr.clear()

set()
```

Burada önce "tr" adlı bir küme oluşturduk. Bu küme, Türkçe'ye özgü karakterleri barındırıyor. Daha sonra da `clear()` metodunu kullanarak bu kümenin bütün öğelerini sildik. Artık elimizde boş bir liste var...

7.2.2 copy metodu

Değiştirilebilir veri tipleri olan listeler ve sözlüklerden edindiğimiz tecrübeye göre, eğer değiştirilebilir bir veri tipini kopyalamak istersek yapmamız gereken şey şu değildir:

```
>>> liste = range(100)
>>> yedek = liste
```

Bu şekilde aslında listeyi kopyalamadığımızı, yalnızca aynı nesneye (*range(100)*) gönderme yapan iki farklı isim oluşturduğumuzu öğrenmiştik. Dolayısıyla yukarıdaki gibi bir işlem yaptığımızda, listedeki değişiklikler yedek adlı nesneyi de etkileyecektir. Eğer istediğimiz buysa ne ala!... Ama eğer yapmak istediğimiz şey bu değilse, izlememiz gereken yöntem farklı olacaktır.

Listeler ve sözlükleri incelerken `copy()` adlı bir metot öğrenmiştik. Bu metot aynı zamanda kümelerle birlikte de kullanılabilir. Üstelik işlevi de aynıdır:

```
>>> tr = set("şçöğüı")

>>> yedek = tr.copy()

>>> tr

{'ç', 'ı', 'ş', 'ö', 'ü', 'ğ'}

>>> yedek

{'ı', 'ğ', 'ç', 'ö', 'ü', 'ş'}
```

Burada bir şey dikkatinizi çekmiş olmalı. "tr" adlı kümeyi "yedek" adıyla kopyaladık, ama bu iki kümenin çıktılarına baktığımız zaman öğe sıralamasının birbirinden farklı olduğunu görüyoruz. Bu da bize kümelerle ilgili çok önemli bir bilgi daha veriyor. Demek ki, tıpkı sözlüklerde olduğu gibi, kümeler de sırasız veri tipleriymiş. Elde ettiğimiz çıktıda öğeler rastgele diziliyor... Dolayısıyla öğelere sıralarına göre erişemiyoruz. Aynen sözlüklerde olduğu gibi...

7.2.3 add metodu

Kümelerden bahsederken, bunların değiştirilebilir bir veri tipi olduğunu söylemiştik. Dolayısıyla kümeler, üzerlerinde değişiklik yapmamıza müsaade eden metotlar da içerir. Örneğin `add()` bu tür metotlardan biridir. "Add" kelimesi Türkçe'de "eklemek" anlamına gelir. Adından da anlaşılacağı gibi, bu metot yardımıyla kümelerimize yeni öğeler ilave edebileceğiz. Hemen bunun nasıl kullanıldığına bakalım:

```
>>> kume = set(["elma", "armut", "kebap"])

>>> kume.add("çilek")

>>> print(kume)

{'elma', 'armut', 'kebap', 'çilek'}
```

Gördüğünüz gibi, `add()` metodunu kullanarak, kümemize "çilek" adlı yeni bir öğe ekledik. Eğer kümede zaten varolan bir öğe eklemeye çalışırsak kümede herhangi bir değişiklik olmayacaktır. Çünkü, daha önce de söylediğimiz gibi, kümeler her bir öğeyi tek bir sayıda barındırır...

Eğer bir kümeye birden fazla öğeyi aynı anda eklemek isterseniz `for` döngüsünden yararlanabilirsiniz:

```
>>> yeni = [1, 2, 3]

>>> for i in yeni:
...     kume.add(i)
...
{1, 2, 3, 'elma', 'kebap', 'çilek', 'armut'}
```

Burada yeni adlı listeyi kümeye `for` döngüsü ile ekledik. Ama bu işlemi yapmanın başka bir yolu daha vardır. Bu işlem için Python'da ayrı bir metot bulunur. Bu metodun adı `update()` metodudur. Sırası gelince bu metodu da göreceğiz...

7.2.4 difference metodu

Bu metot iki kümenin farkını almamızı sağlar. Örneğin:

```
>>> k1 = set([1, 2, 3, 5])
>>> k2 = set([3, 4, 2, 10])

>>> k1.difference(k2)

{1, 5}
```

Demek ki k1'in k2'den farkı buymuş... Peki k2'nin k1'den farkını bulmak istersek ne yapacağız?

```
>>> k2.difference(k1)

{10, 4}
```

Gördüğünüz gibi, birinci kullanımda, k1'de bulunup k2'de bulunmayan öğeleri elde ediyoruz. İkinci kullanımda ise bunun tam tersi. Yani ikinci kullanımda k2'de bulunup k1'de bulunmayan öğeleri alıyoruz...

İsterseniz uzun uzun `difference()` metodunu kullanmak yerine sadece "eksi" (-) işaretini kullanarak da aynı sonucu elde edebilirsiniz:

```
>>> k1 - k2
```

...veya...

```
>>> k2 - k1
```

Hayır, "madem eksi işaretini kullanabiliyoruz, o halde artı işaretini de kullanabiliriz!" gibi bir fikir doğru değildir...

7.2.5 difference_update metodu

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

Hemen bir örnek verelim:

```
>>> k1 = set([1, 2, 3])
>>> k2 = set([1, 3, 5])

>>> k1.difference_update(k2)

{2}
```

```
>>> print(k1)

{2}

>>> print(k2)

{1, 3, 5}
```

Gördüğünüz gibi, bu metot k1'in k2'den farkını aldı ve bu farkı kullanarak k1'i yeniden oluşturdu... k1 ile k2 arasındaki tek fark "2" adlı öğe idi. Dolayısıyla `difference_update()` metodunu uyguladığımızda k1'in öğelerinin silinip yerlerine "2" adlı öğenin geldiğini görüyoruz...

7.2.6 discard metodu

Bir önceki bölümde öğrendiğimiz `add()` metodu yardımıyla, önceden oluşturduğumuz bir kümeye yeni öğeler ekleyebiliyorduk. Bu bölümde öğreneceğimiz `discard()` metodu ise kümeden öğe silmemizi sağlayacak:

```
>>> hayvanlar = set(["kedi", "köpek", "at", "kuş", "inek", "deve"])

>>> hayvanlar.discard("kedi")

>>> print(hayvanlar)

{'köpek', 'kuş', 'deve', 'inek', 'at'}
```

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

```
>>> hayvanlar.discard("yılan")
```

Burada etkileşimli kabuk sessizce bir alt satıra geçecektir. Bu metodun en önemli özelliği budur. Yani olmayan bir öğeyi silmeye çalıştığımızda hata vermemesi...

7.2.7 remove metodu

Bu metot da bir önceki bölümde gördüğümüz `discard()` metoduyla aynı işlevi yerine getirir. Eğer bir kümeden öğe silmek istersek `remove()` metodunu da kullanabiliriz:

```
>>> hayvanlar.remove("köpek")
```

Peki `discard()` varken `remove()` metoduna ne gerek var? Ya da tersi... Bu iki metot aynı işlevi yerine getirse de aralarında önemli bir fark vardır. Hatırlarsanız `discard()` metoduyla, kümede olmayan bir öğeyi silmeye çalışırsak herhangi bir hata mesajı almayacağımızı söylemiştik. Eğer `remove()` metodunu kullanarak, kümede olmayan bir öğeyi silmeye çalışırsak, `discard()` metodunun aksine, hata mesajı alırız:

```
>>> hayvanlar.remove("fare")

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'fare'
```

Bu iki metot arasındaki bu fark önemli bir farktır. Bazen yazdığımız programlarda, duruma göre her iki özelliğe de ihtiyacımız olabilir...

7.2.8 intersection metodu

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

```
>>> k1 = set([1, 2, 3, 4])
>>> k2 = set([1, 3, 5, 7])

>>> k1.intersection(k2)

{1, 3}
```

Gördüğünüz gibi, bu metot bize k1 ve k2'nin kesişim kümesini veriyor. Dolayısıyla bu iki küme arasındaki ortak elemanları bulmuş oluyoruz...

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

```
>>> k1 & k2

{1, 3}
```

Python programcıları genellikle uzun uzun "intersection" yazmak yerine "&" işaretini kullanırlar...

İsterseniz bu metot için örnek bir program verelim. Böylece gerçek hayatta bu metodu nasıl kullanabileceğimizi görmüş oluruz:

```
parola = input("Lütfen sisteme giriş için bir parola belirleyin: ")

tr = "şçöğüıŞÇÖĞÜİ"

if set(tr) & set(parola):
    print("Lütfen parolanızda Türkçe harfler kullanmayın!")
else:
    print("Parolanız kabul edildi!")
```

Burada eğer kullanıcı, parola belirlerken içinde Türkçe bir harf geçen bir kelime yazarsa programımız kendisini Türkçe harf kullanmaması konusunda uyaracaktır. Bu kodlarda kümeleri nasıl kullandığımıza dikkat edin. Programda asıl işi yapan kısım şu satırdır:

```
if set(tr) & set(parola):
    print("Lütfen parolanızda Türkçe harfler kullanmayın!")
```

Burada aslında şöyle bir şey demiş oluyoruz:

> *Eğer set(tr) ve set(parola) kümelerinin kesişim kümesi boş değilse, kullanıcıya "Lütfen parolanızda Türkçe harfler kullanmayın!" uyarısını göster!*

set(tr) ve set(parola) kümelerinin kesişim kümesinin boş olmaması, kullanıcının girdiği kelime içindeki harflerden en az birinin "tr" adlı değişken içinde geçtiği anlamına gelir... Burada basitçe, "tr" değişkeni ile "parola" değişkeni arasındaki ortak öğeleri sorguluyoruz. Eğer kullanıcı herhangi bir Türkçe harf içermeyen bir kelime girerse set(tr) ve set(parola) kümelerinin kesişim kümesi boş olacaktır. İsterseniz küçük bir deneme yapalım:

```
>>> tr = "şçöğüıŞÇÖĞÜİ"

>>> parola = "çilek"

>>> set(tr) & set(parola)

{'ç'}
```

Burada kullanıcının "çilek" adlı kelimeyi girdiğini varsayıyoruz... Böyle bir durumda set(tr) ve set(parola) kümelerinin kesişim kümesi "ç" harfini içerecek, dolayısıyla da programımız kullanıcıya uyarı mesajı gösterecektir. Eğer kullanıcımız "kalem" gibi Türkçe harf içermeyen bir kelime girerse:

```
>>> tr = "şçöğüıŞÇÖĞÜİ"

>>> parola = "kalem"

>>> set(tr) & set(parola)

set()
```

Gördüğünüz gibi, elde ettiğimiz küme boş... Dolayısıyla böyle bir durumda programımız kullanıcıya herhangi bir uyarı mesajı göstermeyecektir...

`intersection()` metodunu pek çok yerde kullanabilirsiniz. Hatta iki dosya arasındaki benzerlikleri bulmak için dahi bu metottan yararlanabilirsiniz... İlerde dosya işlemleri konusunu işlerken bu metottan nasıl yararlanabileceğimizi de anlatacağız...

7.2.9 intersection_update metodu

Hatırlarsanız `difference_update()` metodunu işlerken şöyle bir şey demiştik:

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

İşte `intersection_update` metodu da buna çok benzer bir işlevi yerine getirir. Bu metodun görevi, `intersection()` metodundan elde edilen sonuca göre bir kümenin güncellenmesini sağlamaktır:

```
>>> k1 = set([1, 2, 3])
>>> k2 = set([1, 3, 5])

>>> k1.intersection_update(k2)

{1, 3}

>>> print(k1)

{1, 3}

>>> print(k2)

{1, 3, 5}
```

Gördüğünüz gibi, `intersection_update()` metodu k1'in bütün öğelerini sildi ve yerlerine k1 ve k2'nin kesişim kümesinin elemanlarını koydu...

7.2.10 isdisjoint metodu

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

```
>>> a = set([1, 2, 3])
>>> b = set([2, 4, 6])

>>> a.isdisjoint(b)

False
```

Gördüğünüz gibi, a ve b kümesinin kesişimi boş olmadığı için, yani bu iki küme ortak en az bir öğe barındırdığı için, `isdisjoint()` metodu "False" (Yanlış) çıktısı veriyor. Burada temel olarak şu soruyu sormuş oluyoruz:

a ve b ayrık kümeler mi?

Python da bize cevap olarak, "Hayır değil," anlamına gelen "False" çıktısını veriyor... Çünkü a ve b kümelerinin ortak bir elemanı var (2).

Bir de şuna bakalım:

```
>>> a = set([1, 3, 5])
>>> b = set([2, 4, 6])

>>> a.isdisjoint(b)

True
```

Burada a ve b kümeleri ortak hiç bir elemana sahip olmadığı için "Doğru" anlamına gelen "True" çıktısını elde ediyoruz...

7.2.11 issubset metodu

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

```
>>> a = set([1, 2, 3])
>>> b = set([0, 1, 2, 3, 4, 5])

>>> a.issubset(b)

True
```

Bu örnekte True çıktısını aldık, çünkü a kümesinin bütün öğeleri b kümesi içinde yer alıyor. Yani a, b'nin alt kümesidir...

7.2.12 issuperset metodu

Bu metot, bir önceki bölümde gördüğümüz `issubset()` metoduna benzer. Matematik derslerinde gördüğümüz "kümeler" konusunda hatırladığınız "b kümesi a kümesini kapsar"

ifadesini bu metotla gösteriyoruz... Önce bir önceki derste gördüğümüz örneğe bakalım:

```
>>> a = set([1, 2, 3])
>>> b = set([0, 1, 2, 3, 4, 5])

>>> a.issubset(b)

True
```

Buradan, "a kümesi b kümesinin alt kümesidir," sonucuna ulaşıyoruz... Bir de şuna bakalım:

```
>>> a = set([1, 2, 3])
>>> b = set([0, 1, 2, 3, 4, 5])

>>> b.issuperset(a)

True
```

Burada ise, "b kümesi a kümesini kapsar," sonucunu elde ediyoruz... Yani b kümesi a kümesinin bütün elemanlarını içinde barındırıyor...

7.2.13 union metodu

`union()` metodu iki kümenin birleşimini almamızı sağlar. Hemen bir örnek verelim:

```
>>> a = set([2, 4, 6, 8])
>>> b = set([1, 3, 5, 7])

>>> a.union(b)

{1, 2, 3, 4, 5, 6, 7, 8}
```

Önceki bölümlerde gördüğümüz bazı metotlarda olduğu gibi, `union()` metodu da bir kısayola sahiptir. `union()` metodu yerine "|" işaretini de kullanabiliriz:

```
>>> a | b
```

`union()` metodu yerine, bu metodun kısayolu olan "|" işareti Python programcıları tarafından daha sık kullanılır...

7.2.14 update metodu

Hatırlarsanız `add()` metodunu anlatırken şöyle bir örnek vermiştik:

```
>>> kume = set(["elma", "armut", "kebap"])

>>> yeni = [1, 2, 3]

>>> for i in yeni:
...     kume.add(i)
...
{1, 2, 3, 'elma', 'kebap', 'çilek', 'armut'}
```

Bu örneği verdikten sonra da şöyle bir şey demiştik:

"Burada yeni adlı listeyi kümeye `for` döngüsü ile ekledik. Ama bu işlemi yapmanın başka bir yolu daha vardır. Bu işlem için Python'da ayrı bir metot bulunur."

İşte bu metodu öğrenmenin vakti geldi. Metodumuzun adı `update()`. Bu metot, bir kümeyi güncellememizi sağlar. İsterseniz yukarıdaki örneği, bu metodu kullanarak tekrar yazalım:

```
>>> kume = set(["elma", "armut", "kebap"])

>>> yeni = [1, 2, 3]

>>> kume.update(yeni)

>>> print(küme)

{1, 2, 3, 'elma', 'kebap', 'armut'}
```

Gördüğünüz gibi, `for` döngüsünü kullanmaya gerek kalmadan aynı sonucu elde edebildik...

7.2.15 symmetric_difference metodu

Daha önceki bölümlerde `difference()` metodunu kullanarak iki küme arasındaki farklı öğeleri bulmayı öğrenmiştik. Örneğin elimizde şöyle iki küme var diyelim:

```
>>> a = set([1, 2, 5])
>>> b = set([1, 4, 5])
```

Eğer a kümesinin b kümesinden farkını bulmak istersek şöyle yapıyoruz:

```
>>> a.difference(b)

{2}
```

Demek ki a kümesinde bulunup b kümesinde bulunmayan öğe "2" imiş...

Bir de b kümesinde bulunup a kümesinde bulunmayan öğelere bakalım:

```
>>> b.difference(a)

{4}
```

Bu da bize "4" çıktısını verdi. Demek ki bu öğe b kümesinde bulunuyor, ama a kümesinde bulunmuyormuş. Peki ya kümelerin ikisinde de bulunmayan öğeleri aynı anda nasıl alacağız? işte bu noktada yardımımıza `symmetric_difference()` adlı metot yetişecek:

```
>>> a.symmetric_difference(b)

{2, 4}
```

Böylece iki kümede de bulunmayan öğeleri aynı anda almış olduk...

7.2.16 symmetric_difference_update metodu

Daha önce `difference_update` ve `intersection_update` gibi metotları öğrenmiştik. `symmetric_difference_update()` metodu da bunlara benzer bir işlevi yerine getirir:

```
>>> a = set([1,2, 5])

>>> b = set([1,4, 5])

>>> a.symmetric_difference_update(b)
```

```
>>> print(a)

set([2, 4])
```

Gördüğünüz gibi, a kümesinin eski öğeleri gitti, yerlerine `symmetric_difference()` metoduyla elde edilen çıktı geldi... Yani a kümesi, `symmetric_difference()` metodunun sonucuna göre güncellenmiş oldu...

7.2.17 pop metodu

İnceleyeceğimiz son metot `pop()` metodu olacak. Gerçi bu metot bize hiç yabancı değil. Bu metodu listeler konusundan hatırlıyoruz. Orada öğrendiğimize göre, bu metot listenin bir öğesini silip ekrana basıyordu... Aslında buradaki fonksiyonu da farklı değil. Burada da kümelerin öğelerini silip ekrana basıyor:

```
>>> a = set(["elma", "armut", "kebap"])

>>> a.pop()

'elma'
```

Peki bu metot hangi ölçüte göre kümeden öğe siliyor? Herhangi bir ölçüt yok. Bu metot, küme öğelerini tamamen rastgele siliyor...

BÖLÜM 8

Hata Yakalama

Bu bölümde yine Python Programlama Dili ile ilgili çok önemli bir konudan söz edeceğiz. Yeni konumuz Python'da Hata Yakalama. Bu bölümde Python'da hatalarla başetmeyi öğreneceğiz...

Hatırlarsanız önceki bölümlerde yazdığımız programlarda, örneğin kullanıcı bizim beklentimizin aksine bir sayı yerine harf girerse programımız hata verip çöküyordu. Gelin isterseniz buna basit bir örnek verelim:

Amacımız iki sayıyı toplayan bir program yazmak olsun:

```
topla1 = input("Lütfen toplama işlemine girecek ilk sayıyı giriniz: ")
topla2 = input("Lütfen toplama işlemine girecek ikinci sayıyı giriniz: ")

sonuç = int(topla1) + int(topla2)
print(sonuç)
```

Bu programı çalıştırdığımızda, programımız kullanıcı tarafından girilen iki sayıyı toplayıp ekrana yazdıracaktır. Burada işlerin doğru gitmesi kullanıcının ekrana iki adet sayı yazmasına bağlıdır. Eğer kullanıcı sayı yerine harf girerse Python'un bize vereceği çıktı şuna benzer bir şey olacaktır:

```
Traceback (most recent call last):
  File "deneme.py", line 5, in <module>
    sonuç = int(topla1) + int(topla2)
ValueError: invalid literal for int() with base 10: 'f'
```

Esasında daha önceki bir bölümde örnek olarak basit bir hesap makinesi yazarken de böyle bir sorunumuzun olduğunu söylemiştik. Orada aynı zamanda sonraki bir derste bu sorunun kesin çözümünü öğreneceğimizi de söylemiştik. İşte bu tür problemlerin çözümünü şimdi inceleyeceğimiz bölümde öğreneceğiz...

8.1 try... except...

Python'da hatalarla başetmek için `try... except...` adlı bloklardan yararlanacağız. Bu blokları Türkçe olarak söylememiz gerekirse şöyle diyebiliriz: "dene... kabul_et...". Buradan anlayabileceğimiz gibi, öncelikle Python'a bir şey deneteceğiz. Bu deneme esnasında bir hata meydana gelirse, Python bu hatayı kabul edecek, yani sineye çekecek ve hiçbir şey olmamış gibi yoluna devam edecektir. İsterseniz bu sözlerimizi bir örnekle süsleyelim.

Geçen bölümde şöyle bir örnek vermiştik:

```
topla1 = input("Lütfen toplama işlemine girecek ilk sayıyı giriniz: ")
topla2 = input("Lütfen toplama işlemine girecek ikinci sayıyı giriniz: ")

sonuç = int(topla1) + int(topla2)
print(sonuç)
```

Bu programı çalıştırdığımızda eğer kullanıcı sırasıyla iki tane sayı girerse ne ala!... Programımız hiç bir sorun çıkartmadan bu sayıları toplayacaktır. Ama eğer kullanıcı sayı girmek yerine, mesela "a" harfine basarsa programımızın şöyle bir hata vereceğini biliyoruz:

```
Traceback (most recent call last):
  File "deneme.py", line 5, in <module>
    sonuç = int(topla1) + int(topla2)
ValueError: invalid literal for int() with base 10: 'a'
```

Bu hata mesajı bizim için önemli bazı bilgiler içeriyor. Burada bizim dikkat etmemiz gereken, bu hatanın son satırı... Görüyoruz ki hata mesajının son satırında "ValueError" ifadesi var. İşte hata mesajının bizim işimize yarayacak kısmı bu. Demek ki kullanıcı sayı yerine harf girerse Python'un verdiği hata türü "ValueError" imiş... Şimdi şöyle bir şey yazalım:

```
try:
    topla1 = input("Lütfen toplama işlemine girecek ilk sayıyı giriniz: ")
    topla2 = input("Lütfen toplama işlemine girecek ikinci sayıyı giriniz: ")
    sonuç = int(topla1) + int(topla2)
    print(sonuç)

except ValueError:
    print("Yanlış değer!")
```

Artık kullanıcımız sayı dışında bir değer girerse, programımız hata vermek yerine daha anlamlı bir çıktı üretecek ve kullanıcıya "Yanlış değer!" uyarısı gösterecektir. Burada yaptığımız şey, programın bütün aşamalarını bir `try...` bloğu içine almak.. Kullanıcının sayı dışı bir veri girmesi durumunda "ValueError" ile karşılaşacağımızı biliyoruz. Bu hata türünü "except ValueError" şeklinde gösterip, Python'un bu hata türünü kabul etmesini istiyoruz. Aslında burada Python'a şu emri vermiş oluyoruz:

> *"Kullanıcıya, birbiriyle toplanacak sayıları sor! Daha sonra bu sayıları topla ve sonucu ekrana yaz! Eğer "try" bloğu içinde gerçekleşen işlemlerin herhangi bir aşamasında "ValueError" hatası oluşursa pat diye çökmek yerine ekrana "Yanlış değer!" ifadesini yazdır!"*

İsterseniz `try.. except..` bloğunun işlevini daha net görebilmek için yukarıdaki örneğimizi bir `while` döngüsü içine alalım. Böylece programımız tekrar tekrar çalışsın:

```
while True:
    topla1 = input("Lütfen toplama işlemine girecek ilk sayıyı giriniz: ")
    topla2 = input("Lütfen toplama işlemine girecek ikinci sayıyı giriniz: ")
    sonuç = int(topla1) + int(topla2)
    print(sonuç)
```

Gördüğünüz gibi, bu şekilde programımız her işlem sonunda başa dönüp kullanıcıya tekrar sayı soruyor. Tabii eğer kullanıcı harf yerine başka bir veri girerse programımızın çalışması sona eriyor. Bir de şuna bakalım:

```
while True:
    try:
        topla1 = input("Lütfen toplama işlemine girecek ilk sayıyı giriniz: ")
```

```
        topla2 = input("Lütfen toplama işlemine girecek ikinci sayıyı giriniz: ")
        sonuç = int(topla1) + int(topla2)
        print(sonuç)

    except ValueError:
        print("yanlış değer!")
```

Programımızı çalıştırdığımızda, kullanıcı ne tür bir veri girerse girsin programımız çalışmasına devam edecektir. Eğer kullanıcı iki adet sayı girerse bunları toplayıp sonucu ekrana yazdıracak; ama eğer kullanıcı sayı yerine başka bir şey girerse de önce kullanıcıyı yanlış değer girdiği konusunda uyaracak ve yine çalışmasına devam edecektir.

Bu demek oluyor ki, `try... except...` bloğu sayesinde programımız "ValueError" hatasıyla karşılaşsa bile çökmeden çalışmaya devam edebilecektir.

Buradan, Python'da hata yakalama konusunun önemli bir faydasını öğrenmiş oluyoruz: Bildiğiniz gibi, Python'u kendi haline bıraktığınızda, kullanıcı için pek bir şey ifade etmeyecek hata mesajları üretiyor. Bu mesajlar bir Python programcısı için çok şey ifade ediyor olsa da, programımızı kullanan kimselerin bu "kriptik" hata mesajlarını anlamasını bekleyemeyiz. O yüzden onlara bu tür bulanık hata mesajları yerine, yukarıda gördüğümüz `try... except...` bloklarını kullanarak daha anlamlı mesajlar gösterebiliriz... Ne de olsa "*yanlış değer!*" gibi bir ileti, "*ValueError: invalid literal for int() with base 10: 'a'*" gibi, oldukça bilgisayarvari bir hata mesajına göre çok daha anlamlı olacaktır son kullanıcı için.

İsterseniz daha önce yaptığımız "basit hesap makinesi"ne geri dönüp, yukarıda öğrendiğimiz yeni bilgiyi o kodlara uygulayalım. Zira hesap makinemiz de, eğer kullanıcı sayı yerine harf girerse çöküyordu:

```
#!/usr/bin/env python3.0

while True:

    giriş = """
    (1) topla
    (2) çıkar
    (3) çarp
    (4) böl
    (5) karesini hesapla

    Programdan çıkmak için "ç" harfine basınız...
    """

    print(giriş)

    soru = input("Yapmak istediğiniz işlemin numarasını girin: ")

    try:
        if soru == "1":
            sayı1 = int(input("Toplama işlemi için ilk sayıyı girin: "))
            sayı2 = int(input("Toplama işlemi için ikinci sayıyı girin: "))
            print(sayı1, "+", sayı2, "=", sayı1 + sayı2)

        elif soru == "2":
            sayı3 = int(input("Çıkarma işlemi için ilk sayıyı girin: "))
            sayı4 = int(input("Çıkarma işlemi için ikinci sayıyı girin: "))
            print(sayı3, "-", sayı4, "=", sayı3 - sayı4)

        elif soru == "3":
```

```
            sayı5 = int(input("Çarpma işlemi için ilk sayıyı girin: "))
            sayı6 = int(input("Çarpma işlemi için ikinci sayıyı girin: "))
            print(sayı5, "x", sayı6, "=", sayı5 * sayı6)

        elif soru == "4":
            sayı7 = int(input("Bölme işlemi için ilk sayıyı girin: "))
            sayı8 = int(input("Bölme işlemi için ikinci sayıyı girin: "))
            print(sayı7, "/", sayı8, "=", sayı7 / sayı8)

        elif soru == "5":
            sayı9 = int(input("Karesini hesaplamak istediğiniz sayıyı giriniz: "))
            print(sayı9, "sayısının karesi =", sayı9 ** 2)

        elif soru == "ç":
            print("Tekrar görüşmek üzere!")
            a = 0
        else:
            print("Yanlış giriş.")
            print("Aşağıdaki seçeneklerden birini giriniz:")

    except ValueError:
        print("Sadece sayı giriniz!")
```

Gördüğünüz gibi, burada da yaptığımız şey, kullanıcının sayı yerine harf girmesi durumunda Python'un "ValueError" hatası vereceğini bildiğimiz kodları `try...` bloğu içine almaktan ibaret... "ValueError" hatası alındığında kullanıcıya gösterilecek mesajı ise "except ValueError" bloğu içinde gösteriyoruz.

Yukarıda gördüğümüz `try... except...` blokları konusunda önemli bir nokta vardır dikkat etmemiz gereken... "except ValueError" şeklinde bir ifade kullandığımızda, Python yalnızca "ValueError" hatalarını yakalayacaktır. Bir program "ValueError" dışında da pek çok hata verebilir. Mesela yukarıda tekrar verdiğimiz hesap makinemizin bir sorunu daha var. Bu programda bölme işlemi sırasında bir sayıyı 0'a bölmeye çalışırsak programımız şöyle bir hata verecektir:

```
Traceback (most recent call last):
  File "deneme.py", line 38, in <module>
    print(sayı7, "/", sayı8, "=", sayı7 / sayı8)
ZeroDivisionError: int division or modulo by zero
```

Programlama dillerinde bir sayının sıfıra bölünmesi programın çökmesiyle sonuçlanır. O yüzden biz de yazdığımız programlarda yapılan işlemlerin bir aşamasında herhangi bir sayının sıfıra bölünmediğinden emin olmalıyız...

Yukarıdaki hata mesajında bizi ilgilendiren kısım "ZeroDivisionError". Dolayısıyla hata mesajını yakalarken bu ifadeyi kullanacağız. Yazacağımız kodun taslağı şöyle olacak:

```
try:
    ...hata vereceğini bildiğimiz kodlar...
except ZeroDivisionError:
    ...hata geldiğinde kullanıcıya gösterilecek mesaj veya yapılacak işlem...
```

Bu kodları, yukarıda "ValueError" için yazdığımız kodların yerine koyabiliriz. Bu basit bir iş. Peki ya aynı program içinde hem "ValueError" hatasını, hem de "ZeroDivisionError" hatasını yakalamak istersek ne yapacağız? Bu daha da basit bir iş. Taslağımız şöyle:

```
try:
    ...hata vereceğini bildiğimiz kodlar...
except ValueError:
```

```
    ...ValueError hatası aldığımızda yapacağımız işlem...
except ZeroDivisionError:
    ...ZeroDivisionError hatası aldığımızda yapacağımız işlem...
```

Gördüğünüz gibi, birden fazla hatayı aynı anda yakalamak istediğimizde, her bir hata için ayrı bir `except...` bloğu yazmamız yeterli oluyor. Elbette, yukarıdaki taslak, her bir hata mesajı için ayrı bir işlem yapmak istediğimizde geçerli. Eğer hangi hatayla karşılaşılırsa karşılaşılsın aynı mesaj gösterilecekse veya aynı işlem yapılacaksa kodlarımızı şöyle de yazabiliriz:

```
try:
    ...hata vereceğini bildiğimiz kodlar...
except (ValueError, ZeroDivisionError):
    ...hata alınınca yapılacak işlem...
```

Burada hata mesajlarını bir demet içinde topladığımıza özellikle dikkat edin.

Bir sonraki bölümde bu "hata yakalama" konusunu incelemeye devam edeceğiz. Ama öncelikle öğrenmemiz gereken başka şeyler var...

8.2 "break" Deyimi

Python'da `break` özel bir deyimdir. Bu deyim yardımıyla, devam eden bir süreci kesintiye uğratabiliriz. Bu deyimin kullanıldığı basit bir örnek verelim:

```
>>> tekrar = 1

>>> while tekrar <= 10:
...     tekrar = tekrar + 1
...     try:
...         soru = int(input("öğrencinin notu: "))
...     except ValueError:
...         break
```

Burada öncelikle "tekrar" adlı bir değişken tanımladık. Bu değişkenin değeri "1". "tekrar" adlı değişkenin değeri 10'dan küçük veya 10'a eşit olduğu müddetçe programımızın çalışmaya devam etmesi için "*while tekrar <=10*" şeklinde bir satır yazıyoruz. Daha sonra, tanımladığımız bu `while` döngüsü içinde, "tekrar" adlı değişkenin değerini her döngüde 1 sayı artırıyoruz. Böylece programımız "tekrar" adlı değişkenin değeri 10'a ulaşıncaya kadar çalışmaya devam edecek. Ardından, "*soru = int(input("öğrencinin notu: "))*" ifadesi yardımıyla bir dönüştürme işlemi yapıyoruz. `input()` fonksiyonu ile kullanıcıdan beklediğimiz verinin tipi "sayı" olacak. Kullanıcının sayı yerine bir harf girmesi ihtimaline karşı `input()` fonksiyonunu bir `try...` bloğu içine yerleştirdik. Eğer kullanıcı sayı yerine harf girerse programımızın "ValueError" hatası vereceğini biliyoruz. O yüzden bir `except...` bloğu oluşturarak bu olası hatayı yakalıyoruz. `break` deyimi bu noktada devreye giriyor. Eğer kullanıcı "ValueError" hatasının verilmesine yol açacak bir veri girerse programımız sessizce sonlanacaktır... Bu sessizce sonlanma işini yerine getiren kodumuz en son satırdaki `break` ifadesi...

Programı çalıştırdığımızda, eğer kullanıcı sayı yerine bir harf girerse, veya hiç bir veri girmeden "enter" tuşuna basarsa programımız kapanacaktır.

Gördüğünüz gibi, `break` ifadesinin temel görevi bir döngüyü sona erdirmek. Buradan anlayacağımız gibi, `break` ifadesinin her zaman bir döngü içinde yer alması gerekiyor. Aksi halde Python bize şöyle bir hata verecektir:

```
SyntaxError: 'break' outside loop
```

Yani; "*SözDizimiHatası: "break" döngü dışında*"

Bununla ilgili basit bir örnek daha verelim:

```
>>> while True:
...     parola = input("Lütfen bir parola belirleyiniz:")
...         if len(parola) < 5:
...             print("Parola 5 karakterden az olmamalı!")
...         else:
...             print("Parolanız belirlendi!")
...             break
```

Burada da, eğer kullanıcının girdiği parolanın uzunluğu 5 karakterden azsa, "Parola 5 karakterden az olmamalı!" uyarısı gösterilecektir. Eğer kullanıcı 5 karakterden uzun bir parola belirlemişse, kendisine "Parolanız belirlendi!" mesajını gösterip, `break` deyimi yardımıyla programdan çıkıyoruz...

8.3 "pass" Deyimi

"pass" kelimesi İngilizce'de "geçmek" anlamına gelir. Bu deyimin Python programlama dilindeki anlamı da buna çok yakındır. Bu deyimi Pyhon'da "görmezden gel, hiçbir şey yapma" anlamında kullanacağız:

```
>>> liste = []
>>> while True:
...     a = input("Herhangi bir karakter giriniz: ")
...     if a == "0":
...         pass
...     else:
...         liste.append(a)
...         print(liste)
```

Burada eğer kullanıcının girdiği karakter "0" ise hiçbir şey yapmıyoruz. Ama eğer kullanıcı "0" dışında herhangi bir karakter girerse, bu karakterleri alıp listeye ekliyoruz. Bu kodlarla Python'a şöyle bir şey söylemiş oluyoruz:

> *Eğer kullanıcı "0" karakterini girerse görmezden gel. Hiçbir şey yapma!... Ama eğer girilen karakter "0" dışında bir şeyse bunu listeye ekle ve listeyi ekrana bas!*

`pass` deyimini hata yakalama işlemlerinde de kullanabiliriz. Eğer bir hata yakalandığında programımızın hiç bir şey yapmadan yoluna devam etmesini istiyorsak bu deyim işimize yarayacaktır:

```
try:
    ....bir şeyler...
except IndexError:
    pass
```

Bu şekilde programımız "IndexError" hatasıyla karşılaşırsa hiç bir şey yapmadan yoluna devam edecek, böylece kullanıcımız programda ters giden bir şeyler olduğunu dahi anlamayacaktır!... Yazdığınız programlarda bunun iyi bir şey mi yoksa kötü bir şey mi olduğuna programcı olarak sizin karar vermeniz gerekiyor... Eğer bir hatanın kullanıcıya gösterilmesinin gerekmediğini düşünüyorsanız yukarıdaki kodları kullanın, ama eğer verilen hata önemli bir hataysa ve kullanıcının bu durumdan haberdar olması gerektiğini düşünüyorsanız, bu hatayı `pass` ile geçiştirmek yerine, kullanıcıya hatayla ilgili makul ve anlaşılır bir mesaj göstermeyi düşünebilirsiniz...

Yukarıda anlatılan durumların dışında, pass deyimini kodlarınız henüz taslak aşamasında olduğu zaman da kullanabilirsiniz. Örneğin, diyelim ki bir kod yazıyorsunuz. Programın gidişatına göre, bir noktada yapmanız gereken bir işlem var, ama henüz ne yapacağınıza karar vermediniz. Böyle bir durumda pass deyiminden yararlanabilirsiniz. Mesela birtakım if deyimleri yazmayı düşünüyor olun:

```
if .....:
    böyle yap
elif .....:
    şöyle yap
else:
    pass
```

Burada henüz else bloğunda ne yapılacağına karar vermemiş olduğunuz için, oraya bir pass koyarak durumu şimdilik geçiştiriyorsunuz. Program son haline gelene kadar oraya bir şeyler yazmış olacağız.

Sözün özü, pass deyimlerini, herhangi bir işlem yapılmasının gerekli olmadığı durumlar için kullanıyoruz.. İlerde işe yarar programlar yazdığınızda, bu pass deyiminin göründüğünden daha faydalı bir araç olduğunu anlayacaksınız...

8.4 “continue” Deyimi

continue ilginç bir deyimdir. İsterseniz continue‘yi anlatmaya çalışmak yerine bununla ilgili bir örnek verelim:

```
>>> while True:
...     s = input("Bir sayı girin: ")
...     if s == "iptal":
...         break
...     if len(s) <= 3:
...         continue
...     print("En fazla üç haneli bir sayı girebilirsiniz.")
```

Burada eğer kullanıcı klavyede “iptal” yazarsa programdan çıkılacaktır. Bunu;

```
if s == "iptal":
    break
```

satırıyla sağlamayı başardık.

Eğer kullanıcı tarafından girilen sayı üç haneli veya daha az haneli bir sayı ise, continue ifadesinin etkisiyle:

```
>>> print("En fazla üç haneli bir sayı girebilirsiniz.")
```

satırı es geçilecek ve döngünün en başına gidilecektir.

Eğer kullanıcının girdiği sayıdaki hane üçten fazlaysa ekrana:

```
En fazla üç haneli bir sayı girebilirsiniz.
```

cümlesi yazdırılacaktır.

Dolayısıyla buradan anladığımıza göre, continue deyiminin görevi kendisinden sonra gelen her şeyin es geçilip döngünün başına dönülmesini sağlamaktır. Bu bilgiye göre, yukarıdaki programda eğer kullanıcı, uzunluğu 3 karakterden az bir sayı girerse continue deyiminin etkisiyle

programımız döngünün en başına geri gidiyor. Ama eğer kullanıcı, uzunluğu 3 karakterden fazla bir sayı girerse, ekrana "En fazla üç haneli bir sayı girebilirsiniz," cümlesinin yazdırıldığını görüyoruz.

8.5 else... finally...

Python'da hata yakalama işlemleri için çoğunlukla `try...` `except...` bloklarını bilmek yeterli olacaktır. İşlerimizin büyük kısmını sadece bu blokları kullanarak halledebiliriz. Ancak Python bize bu konuda, zaman zaman işimize yarayabilecek başka araçlar da sunmaktadır. İşte `else...` `finally` blokları da bu araçlardan biridir. Bu bölümde kısaca `else...` ve `finally...` bloklarının ne işe yaradığından söz edeceğiz.

Öncelikle `else...` bloğunun ne işe yaradığına bakalım. Esasında biz bu else deyimini daha önce de "koşullu ifadeler" konusunu işlerken görmüştük. Buradaki kullanımı da zaten hemen hemen aynıdır. Diyelim ki elimizde şöyle bir şey var:

```
try:
    bölünen = int(input("bölünecek sayı: "))
    bölen = int(input("bölen sayı: "))
    print(bölünen/bölen)

except ValueError:
    print("hata!")
```

Burada eğer kullanıcı sayı yerine harf girerse "ValueError" hatası alırız. Bu hatayı "except ValueError:" ifadesiyle yakalıyoruz ve hata verildiğinde kullanıcıya uyarı vererek programımızın çökmesini engelliyoruz. Ama biliyoruz ki, bu kodları çalıştırdığımızda Python'un verebileceği tek hata "ValueError" değildir. Eğer kullanıcı bir sayıyı 0'a bölmeye çalışırsa Python "ZeroDivisionError" adlı hatayı verecektir. Dolayısıyla bu hatayı da yakalamak için şöyle bir şey yazabiliriz:

```
try:
    bölünen = int(input("bölünecek sayı: "))
    bölen = int(input("bölen sayı: "))
    print(bölünen/bölen)

except ValueError:
    print("hata!")

except ZeroDivisionError:
    print("Bir sayıyı 0'a bölemezsiniz!")
```

Bu şekilde hem "ValueError" hatasını hem de "ZeroDivisionError" hatasını yakalamış oluruz... Bu kodların özelliği, `except...` bloklarının tek bir `try...` bloğunu temel almasıdır. Yani biz burada bütün kodlarımızı tek bir `try...` bloğu içine tıkıştırıyoruz. Bu blok içinde gerçekleşen hataları da daha sonra tek tek `except...` blokları yardımıyla yakalıyoruz. Ama eğer biz istersek bu kodlarda verilebilecek hataları gruplamayı da tercih edebiliriz:

```
try:
    bölünen = int(input("bölünecek sayı: "))
    bölen = int(input("bölen sayı: "))

except ValueError:
    print("hata!")

else:
```

```
    try:
        print(bölünen/bölen)

    except ZeroDivisionError:
        print("Bir sayıyı 0'a bölemezsiniz!")
```

Burada yaptığımız şey şu: İlk `try...` `except...` bloğu yardımıyla öncelikle "int(input())" fonksiyonu ile kullanıcıdan gelecek verinin sayı olup olmadığını denetliyoruz. Ardından bir "else..." bloğu açarak, bunun içinde ikinci `try...` `except...` bloğumuzu devreye sokuyoruz. Burada da bölme işlemini gerçekleştiriyoruz. Kullanıcının bölme işlemi sırasında "0" sayısını girmesi ihtimaline karşı da "except ZeroDivisionError" ifadesi yardımıyla olası hatayı göğüslüyoruz... Bu şekilde bir kodlamanın bize getireceği avantaj, hatalar üzerinde belli bir kontrol sağlamamıza yardımcı olmasıdır. Yukarıdaki kodlar sayesinde hatalara bir nevi "teker teker gelin!" mesajı vermiş oluyoruz!... Böylelikle her blok içinde sadece almayı beklediğimiz hatayı karşılıyoruz. Mesela yukarıda ilk `try...` bloğu içindeki dönüştürme işlemi yalnızca "ValueError" hatası verebilir. `else:` bloğundan sonraki `try...` bloğunda yer alan işlem ise ancak "ZeroDivisionError" verecektir. Biz yukarıda kullandığımız yapı sayesinde her bir hatayı tek tek ve yeri geldiğinde karşılıyoruz. Bu durumun aksine, bölümün ilk başında verdiğimiz `try...` bloğunda hem "ValueError" hem de "ZeroDivisionError" hatalarının gerçekleşme ihtimali bulunuyor. Dolayısıyla biz orada bütün hataları tek bir `try...` bloğu içine sıkıştırmış oluyoruz. İşte `else:` bloğu bu sıkışıklığı gidermiş oluyor. Ancak sizi bir konuda uyarmak isterim: Bu yapı, her akla geldiğinde kullanılacak bir yapı değildir. Büyük programlarda bu tarz bir kullanım kodlarınızın darmadağın olmasına, kodlarınız üzerindeki denetimi tamamen kaybetmenize yol açabilir. Sonunda da elinizde bölük pörçük bir kod yığını kalabilir... Zaten açıkça söylemek gerekirse `try...` `except...` `else...` yapısının çok geniş bir kullanım alanı yoktur. Bu yapı ancak çok nadir durumlarda kullanılmayı gerektirebilir. Dolayısıyla bu üçlü yapıyı hiç kullanmadan bir ömrü rahatlıkla geçirebilirsiniz...

`try...` `except...` `else...` yapılarının dışında, Python'un bize sunduğu bir başka yapı da `try...` `except...` `finally...` yapılarıdır... Bunu şöyle kullanıyoruz:

```
try:
    ...bir takım işler...

except birHata:
    ...hata alınınca yapılacak işlemler...

finally:
    ...hata olsa da olmasa da yapılması gerekenler...
```

`finally..` bloğunun en önemli özelliği, programın çalışması sırasında herhangi bir hata gerçekleşse de gerçekleşmese de işletilecek olmasıdır. Eğer yazdığınız programda mutlaka ama mutlaka işletilmesi gereken bir kısım varsa, o kısmı `finally...` bloğu içine yazabiliriz.

`finally...` bloğu özellikle dosya işlemlerinde işimize yarayabilir. Henüz Python'da dosyalarla nasıl çalışacağımızı öğrenmedik, ama ben şimdilik size en azından dosyalarla çalışma prensibi hakkında bir şeyler söyleyeyim.

Genel olarak Python'da dosyalarla çalışabilmek için öncelikle bilgisayarda bulunan bir dosyayı okuma veya yazma kipinde açarız. Dosyayı açtıktan sonra bu dosyayla ihtiyacımız olan birtakım işlemler gerçekleştiririz. Dosyayla işimiz bittikten sonra ise dosyamızı mutlaka kapatmamız gerekir. Ancak eğer dosya üzerinde işlem yapılırken bir hata ile karşılaşılırsa dosyamızı kapatma işlemini gerçekleştirdiğimiz bölüme hiç ulaşılamayabilir. İşte `finally...` bloğu böyle bir durumda işimize yarayacaktır:

```
try:
    dosya = open("dosyaadı", "r")
```

```
    ...burada dosyayla bazı işlemler yapıyoruz...
    ...ve ansızın bir hata oluşuyor...

except IOError:
    print("bir hata oluştu!")

finally:
    dosya.close()
```

Burada `finally...` bloğu içine yazdığımız "dosya.close()" ifadesi dosyamızı kapatmaya yarıyor. Bu blok, yazdığımız program hata verse de vermese de işletilecektir.

8.6 except... as...

Python'daki hata mesajları temel olarak iki bölümden oluşur:

1. Hatanın adı,
2. Hatanın adı ile birlikte gösterilen mesaj.

Bir örnek üzerinde görelim bunu:

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

Burada "ValueError" hatanın adıdır. "invalid literal for int() with base 10: 'a"' ise gösterilen mesaj... Biz eğer istersek bu hatanın adını özelleştirebilir, o kısma istediğimiz başka bir ifade yazabiliriz. Bunun için `except... as...` bloklarından yararlanacağız. Hemen bir örnek verelim:

```
try:
    bölünen = int(input("bölünecek sayı: "))
    bölen = int(input("bölen sayı: "))
    print(bölünen/bölen)

except ZeroDivisionError as hata:
    print("Sıfıra Bölme Hatası. Python şu hata mesajını verdi:", hata)
```

Burada "ZeroDivisionError" adlı hatayı aslında bileşenlerine ayırmış olduk... "except ZeroDivisionError as hata" ifadesi yardımıyla bu hata mesajının ana gövdesini "hata" olarak adlandırdık. Son olarak da bu "hata" değişkenini `print()` fonksiyonu içinde kullanarak ekrana yazdırdık.

8.7 raise

Bazen, yazdığımız bir programda, kullanıcının yaptığı bir işlem normal şartlar altında hata vermeyecek olsa bile biz ona "Python tarzı" bir hata mesajı göstermek isteyebiliriz. Böyle bir durumda ihtiyacımız olan şey Python'un bize sunduğu `raise` adlı deyimdir. Bu deyim yardımıyla duruma özgü hata mesajları üretebiliriz. Bir örnek verelim:

```
bölünen = int(input("bölünecek sayı: "))

if bölünen == 23:
    raise Exception("Bu programda 23 sayısını görmek istemiyorum!")
```

```
bölen = int(input("bölen sayı: "))
print(bölünen/bölen)
```

Burada eğer kullanıcı “23” sayısını girerse, kullanıcıya bir hata mesajı gösterilip programdan çıkılacaktır... Biz bu kodlarda “Exception” adlı genel hata mesajını kullandık. Burada “Exception” yerine her istediğimizi yazamayız. Yazabileceklerimiz ancak Python’da tanımlı hata mesajları olabilir. Örneğin NameError, TypeError, ZeroDivisionError, IOError, vb...

`raise` deyimini, bir hata mesajına ek olarak bir işlem yapmak istediğimizde de kullanabiliriz. Örneğin:

```
try:
    bölünen = int(input("bölünecek sayı: "))
    bölen = int(input("bölen sayı: "))
    print(bölünen/bölen)

except ZeroDivisionError:
    print("bir sayıyı 0’a bölemezsiniz")
    raise
```

Burada, eğer kullanıcı bir sayıyı 0’a bölmeye çalışırsa, normal bir şekilde “ZeroDivisionError” hatası verilecek ve programdan çıkılacaktır. Ama bu hata mesajıyla birlikte kullanıcıya “bir sayıyı 0’a bölemezsiniz” uyarısını da gösterme imkanını elde edeceğiz... Yani burada “except ZeroDivisionError” bloğunu herhangi bir hatayı engellemek için değil, hataya ilave bilgi eklemek için kullanıyoruz. Bunu yapmamızı sağlayan şey tabii ki bu kodlar içinde görünen `raise` adlı deyimdir...

8.8 Bütün Hataları Yakalamak

Şimdiye kadar yaptığımız bütün örneklerde `except...` bloğunu bir hata mesajı adıyla birlikte kullandık. Yani örneklerimiz şuna benziyordu:

```
try:
    ....birtakım işler...

except ZeroDivisionError:
    ...hata mesajı...
```

Yukarıdaki kod yardımıyla sadece “ZeroDivisionError” adlı hatayı yakalayabiliriz. Eğer yazdığımız program başka bir hata daha veriyorsa, o hata mesajı yukarıdaki blokların kapsamı dışında kalacaktır. Ama eğer istersek yukarıdaki kodu şu şekilde yazarak olası bütün hataları yakalayabiliriz:

```
try:
    ....birtakım işler...

except:
    ...hata mesajı...
```

Gördüğünüz gibi, burada herhangi bir hata adı belirtmedik. Böylece Python, yazdığımız programda hangi hata oluşursa oluşsun hepsini yakalayabilecektir.

Bu yöntem gözünüze çok pratik görünmüş olabilir, ama aslında hiç de öyle sayılmaz. Hatta oldukça kötü bir yöntem olduğunu söyleyebiliriz bunun. Çünkü bu tarz bir kod yazımının bazı dezavantajları vardır. Örneğin bu şekilde bütün hata mesajlarını aynı kefeye koyarsak, programımızda ne tür bir hata oluşursa oluşsun, kullanıcıya hep aynı mesajı göstermek zorunda

kalacağız. Bu da, herhangi bir hata durumunda kullanıcıyı ne yapması gerektiği konusunda doğru düzgün bilgilendiremeyeceğimiz anlamına geliyor. Yani kullanıcı bir hataya sebep olduğunda tersliğin nereden kaynaklandığını tam olarak kestiremeyecektir.

Ayrıca, eğer kendimiz bir program geliştirirken sürekli olarak bu tarz bir yazımı benimsersek, kendi kodlarımızdaki hataları da maskelemiş oluruz. Dolayısıyla, Python yukarıdaki geniş çaplı `except...` bloğu nedeniyle programımızdaki bütün hataları gizleyeceği için, programımızdaki potansiyel aksaklıkları görme imkanımız olmaz. Dolayısıyla bu tür bir yapıdan olabildiğince kaçınmakta fayda var...

BÖLÜM 9

Genel Tekrar

Bu noktaya kadar Python ile ilgili epey şey öğrendik. Buraya kadar öğrendiklerimizi kullanarak Python programlama dili ile önemli ve yararlı işler yapabiliriz. Bu bölümden sonra Python'un derinliklerine dalmaya başlayacağız. Ama dalışa geçmeden önce teçhizatımızı kontrol etmemiz, sonra da derin bir nefes almamız gerektiğini düşünüyorum. O yüzden bir adım daha ileriye gitmeden önce şimdiye kadar öğrendiğimiz kısımların şöyle bir üzerinden geçmemizin faydalı olacağını zannediyorum.

Yalnız bu "genel tekrar" başlığı sizi yanıltmasın. Bu bölümde zaten bildiğimiz konuları tekrar etmeyeceğiz. Daha doğrusu, yine bildiğimiz konulardan bahsedeceğiz, ama buraya kadar geldiğimiz noktada, konuyu okur açısından boğucu ve sıkıcı hale getirmemek için göz ardı ettiğimiz, atladığımız bilgileri tartışacağız bu bölümde. Dolayısıyla bu "genel tekrar" bölümü, okurların hem eski bilgilerini tazelemesini, hem de buraya kadar olan kısımda bahsetmeden geçtiğimiz bazı önemli ayrıntıları öğrenmesini sağlayacaktır. Bu bölümün en önemli amacı, şimdiye kadar öğrenilen konuları pekiştirmek ve öğrenilen konulara ilişkin yeni bilgiler vermektir. Bu bölümde, "Temel Bilgiler" kategorisinde bulunan, ama yukarıda bahsettiğimiz sebeplerden ötürü bahsetmeden veya derinlemesine incelemeden geçtiğimiz konuları öğreneceğiz.

9.1 Python'u Başlatma Seçenekleri

Eğer bu noktaya kadar gelmişseniz, Python'u başarıyla kurmuş ve çalıştırmışsınız demektir... O yüzden bu bölümde Python'un bilgisayarınıza nasıl kurulacağından ve Python'un nasıl çalıştırılacağından bahsetmek yerine, bu konuyla ilgili bazı ayrıntılara değinmeye çalışacağız. Bu bölümün konusu, Python'u başlatma seçenekleri (startup options)... Yani bu bölümde, Python'u çalıştırırken hangi seçenekleri kullanabileceğimizi ve bu seçeneklerin ne işe yaradığını öğreneceğiz.

Ben bu bölümde, sizin Python'un etkileşimli kabuğuna ulaşmak veya Python ile yazdığınız programları çalıştırmak için şu komutu kullandığınızı varsayacağım:

```
python3
```

Eğer siz kendi sisteminizde Python'u çalıştırmak için başka bir komut kullanıyorsanız (python3.0 gibi...), "python3" gördüğünüz yere, kendi kullandığınız komutu yerleştireceksiniz...

İlk seçeneğimizle başlayalım...

python3 -h

Python'u "-h" seçeneği ile başlattığımızda karşımıza Python'un yardım (help) ekranı gelir. Buradan, Python'u nasıl çalıştıracağınıza, çalıştırırken hangi seçenekleri kullanabileceğinize ve bu seçeneklerin ne işe yaradığına dair kısa bilgilere erişebilirsiniz. İşte biz de bu bölümde, bu yardım ekranında görünen seçeneklerin en önemli olanlarına ve şu aşamada bizi ilgilendirenlerine değineceğiz...

Bu yardım ekranına şu komut ile de erişebilirsiniz:

```
python --help
```

Burada "-" işaretinin çift olduğuna dikkat edin.

python3 -c

Bu seçenek, Python'u bir komutla çalıştırmamızı sağlar. Bu ne demek oluyor? Hemen şöyle bir örnek verelim:

```
python3 -c "print('Merhaba Dünya')"
```

Gördüğünüz gibi, Python'un etkileşimli kabuğunu hiç açmadan veya kodumuzu bir metin düzenleyiciyle yazıp kaydetmeye gerek kalmadan, kodlarımızı doğrudan komut satırı üzerinde çalıştırabiliyoruz. Burada önemli nokta, "-c" seçeneğinden sonra gelecek kodları tırnak içine almak... Ancak bunu yaparken, kodun kendisinin içinde yer alan öbür tırnaklarla bu ilk tırnağın birbirine karışmamasına dikkat ediyoruz. Eğer "-c" seçeneğinden sonra çift tırnak ile başlamışsak, "Merhaba Dünya" karakter dizisini tek tırnak içine almaya özen gösteriyoruz...

Eğer "-c" seçeneğine vereceğimiz komut birden fazla ise her bir komutu ";" işareti ile ayırıyoruz:

```
python3 -c "print('Merhaba Dünya'); print('Sana da merhaba canım!')"

Merhaba Dünya
Sana da merhaba canım!
```

python3 -v

"-v" seçeneğin görevi, Python'un çalışmaya başladığı anda ve kapanırken hangi işlemleri yaptığını ayrıntılı olarak göstermektir. Kullanımı şöyledir:

```
python3 -v
```

Bu komutu verdiğimizde Python'un etkileşimli kabuğu başlayacak, ama kabuk başlarken Python'un arkaplanda neler karıştırdığı da ekrana dökülecektir. Aynı şekilde, etkileşimli kabuğu "CTRL+Z" tuş bileşimiyle kapattığımızda da, arkaplanda olup biten her şey ekrana dökülecektir.

python3 -V

Bu seçenekteki "V"nin büyük harf olduğuna dikkat ediyoruz. Bu seçeneğin kullanımı şöyledir:

```
python3 -V
```

Görevi ise, sistemimizde kurulu olan Python sürümünü göstermektir.

python3 -x

Eğer Python'u bu seçenekle başlatırsak, Python kodlarımızın ilk satırını görmezden gelecektir. Mesela bir örnek verelim:

Şu kodları içeren bir dosyamız olsun:

```
print("Merhaba")
print("Gülegüle")
```

Eğer bu dosyayı şu komutla çalıştırırsak:

```
python3 -x dosya.py
```

'print("Merhaba")' satırı görmezden gelinecek, ekrana "Gülegüle" çıktısı verilecektir.

Böylelikle Python'u başlatma seçenekleri içinde şu an için işimize yarayabilecek seçenekleri görmüş olduk. Python'a ilişkin bilgilerimiz arttıkça öteki seçenekler de bizim için bir anlam ifade etmeye başlayacaktır...

9.2 print() Fonksiyonunun Gücü

Python'un 2.x sürümlerinde `print()` bir deyimdi... Yani 3.x öncesi sürümlerde ekrana bir şeyler yazdırmak istediğimizde şu yapıyı kullanıyorduk:

```
>>> print "bir şeyler..."
```

Ancak Python'un 3.x sürümüyle birlikte, "print" bir fonksiyon halini aldı. Dolayısıyla artık ekrana bir şeyler yazdırmak için şöyle yapıyoruz:

```
>>> print("bir şeyler...")
```

`print()` fonksiyonundaki bu değişiklik, Python'un 2.x'ten 3.x'e geçişinde yapılan en önemli yeniliklerden biridir. Ayrıca `print()`'in bir fonksiyon haline gelmesi ile birlikte bu fonksiyon bazı güçlü özellikler de kazandı. İşte biz de bu bölümde yeni `print()` fonksiyonunun gücünü ortaya koymaya çalışacağız.

Diyelim ki elimizde şöyle bir liste var:

```
>>> liste = ["elma", "armut", "kebap"]
```

Eğer Python'un 2.x sürümlerinden birini kullanıyor olsaydık şöyle bir komut yazabilirdik:

```
>>> for i in liste:
...     print i
```

Bu komut bize şöyle bir çıktı verir:

```
elma
armut
kebap
```

Aynı işlemi Python 3.x ile yaptığımızda da tıpatıp aynı sonucu elde ederiz.

Python 2.x'te, liste öğelerini böyle alt alta değil de yan yana dizmek istersek şöyle bir yol izliyorduk:

```
>>> for i in liste:
...     print i,
...
elma armut kebap
```

Burada "i"nin yanına bir virgül işareti koyduğumuza dikkat edin. Aynı şeyi Python 3.x ile yapmaya çalıştığımızda ise beklentimizin karşılanmayacağını görüyoruz:

```
>>> for i in liste:
...     print(i,)
...
elma
armut
kebap
```

Gördüğünüz gibi, “i”nin yanına koyduğumuz virgülün herhangi bir etkisi olmadı. Peki biz liste öğelerini Python 3.x’i kullanarak nasıl yan yana dizeceğiz? İşte burada `print()` fonksiyonunun gücü devreye girecek. Bakın bu işlemi sadece `print()` fonksiyonunu kullanarak nasıl da halledebiliyoruz:

```
>>> print(*liste)

elma armut kebap
```

Burada Python 3.x ile birlikte kullanım alanı genişletilen “yıldızlı ifadeler” (starred expressions) adlı bir özellikten faydalandık. Yıldızlı ifadeyi `print()` fonksiyonu içinde bir listeye uyguladığımızda, sanki liste öğeleri üzerinde bir `for` döngüsü oluşturulmuş gibi bir sonuç elde edilecektir. Yıldızlı ifadelerin farklı bir kullanımına demetler konusunu işlerken de değinmiştik...

Peki ya bu liste öğelerini ekrana hem böyle yan yana dizmek, hem de her öğenin arasına bir virgül koymak istersek ne yapacağız?

Önce bu işlemi Python 2.x ile nasıl yaptığımıza bakalım:

```
>>> for i in liste:
...     print i + ", ",

elma,  armut,  kebap,
```

Bu kullanım en hafif tabiriyle “çirkin”dir. Ayrıca aslında istediğimiz şeyi de tam olarak yerine getirmez. Çünkü biz liste öğelerinin arasına birer virgül koymak istiyoruz, ama en son öğeden sonra virgül olmasını istemiyoruz. Bu isteğimizi Python 2.x ile gerçekleştirebilmek için karakter dizilerinin (henüz öğrenmediğimiz) `join()` adlı metodundan yararlanmamız gerekir:

```
>>> print ", ".join(liste)

elma, armut, kebap
```

Python 3.x’i kullanarak aynı şeyi yapmak istediğimizde ise `print()` fonksiyonunun gücünden yararlanmamız yeterli olacaktır:

```
>>> print(*liste, sep=", ")

elma, armut, kebap
```

Eğer kullandığınız konsol/terminal işlem sonunda yeni satıra geçmiyorsa bu komutu şöyle de yazabilirsiniz:

```
>>> print(*liste, sep=", ", end="\n")
```

Buradaki “sep” ve “end” parametreleri `print()` fonksiyonunun kıvraklığını artıran etkenler... “sep” parametresi `print()` fonksiyonuna yazılan öğelerin her birinin arasına istediğimiz bir karakter dizisini yerleştirme imkanı sağlıyor bize. Basit bir örnek verelim:

```
>>> print("elma", "armut", "kebap", sep=", ")

elma, armut, kebap
```

Bu örnekte, `print()` fonksiyonu içinde belirtilen karakter dizilerinin her birinin arasına, "sep" parametresi yardımıyla bir virgül işareti koyuyoruz...

"sep" parametresi ile birlikte öğrendiğimiz "end" adlı parametre ise, `print()` fonksiyonu içinde belirtilen ifadenin ne şekilde biteceğini gösteriyor. Biz yukarıdaki örnekte "n" işaretini kullandık. Böylece `print()` fonksiyonu yardımıyla ekrana bir şeyler yazdırdıktan sonra yeni satıra geçebiliyoruz...

`print()` fonksiyonu "sep" ve "end" dışında bir de "file" adlı bir parametre alır. Bu parametre yardımıyla, ekrana yazdıracağımız şeyleri ekran yerine bilgisayarımızdaki bir dosyaya yazdırabiliriz:

```
>>> print("Merhaba Dostlar!", file=open("pydeneme.txt", "w"))
```

Bu özelliğin ne kadar faydalı olabileceğini tahmin edebilirsiniz. Burada `open()` adlı bir fonksiyondan yararlandığımıza dikkat edin. Bu fonksiyonu "dosya işlemleri" konusunu işlerken daha detaylı olarak inceleyeceğiz. Aynı fonksiyona, "hata yakalama" konusunu işlerken de değinmiştik... Bu arada, bilginiz olması açısından, yukarıdaki `open()` fonksiyonu içinde kullandığımız "w" parametresinin, bilgisayarımızda yeni bir dosya oluşturmaya yaradığını söyleyelim... Hatırlarsanız, "hata yakalama" konusu içinde bu fonksiyona değinirken "w" yerine "r" adlı bir parametre kullanmıştık. O parametre dosyayı okumak üzere açmamızı sağlıyordu. "w" parametresi ise dosyayı yazmak üzere açmamızı sağlıyor.

Gelin isterseniz bu "file" parametresini kullanarak az çok yararlı bir uygulama yazalım:

```
ad = input("adınız: ")
soyad = input("soyadınız: ")
eposta = input("e.posta adresiniz: ")

print(ad, soyad, eposta, file=open("deneme.txt", "w"), sep="\n")
print("Girdiğiniz veriler dosyaya başarıyla işlendi!")
```

Burada kullanıcıya adını, soyadını ve e.posta adresini soruyoruz. Ardından kullanıcının girdiği bu verileri, "deneme.txt" adlı bir dosyaya kaydediyoruz. "sep" parametresini nasıl kullandığımıza dikkat edin. Kullanıcının girdiği her bir verinin dosya içinde ayrı bir satırda yer alması için, "sep" parametresine "n" değerini verdik...

9.3 Etkileşimli Kabuğun Hafızası

Python'daki "etkileşimli kabuk" denen şeyin ne olduğunu daha önce anlatmıştık. Etkileşimli kabuğun, Python'un gücüne güç katan en önemli özelliklerden birisi olduğunu söylesek abartmış olmayız. Özellikle böyle bir araca sahip olmayan diller üzerinde çalışmış arkadaşlarım ne demek istediğimi çok daha iyi anlayacaktır. Etkileşimli kabuk programcılar arasında o kadar hayranlık uyandırmıştır ki bazı yazarlar "sırf etkileşimli kabuğun bile Python'u Java'dan üstün kılmaya yeterli olduğunu" söylemekten çekinmemişlerdir.

Bu bölümde, etkileşimli kabuğun ufak ama oldukça pratik bir özelliğinden söz edeceğiz.

Diyelim ki etkileşimli kabukta bazı aritmetik işlemler yapıyorsunuz... Örneğin şöyle bir işlem yaptınız:

```
>>> 54345345 * 45

2445540525
```

Daha sonra da bu sayıyı 25'e bölmeye karar verdiniz... Bu işlemi yapmak için değişik yollar kullanabilirsiniz, ama bütün yollar içinde en kolayı "_" işlecinden yararlanmaktır. Bu işleç, yapılan

en son işlemin değerini barındırır. Etkileşimli kabukta “54345345 * 45” işlemini yaptıktan ve sonucu aldıktan sonra şu komutu verin:

```
>>> _

2445540525
```

Gördüğünüz gibi b u komut size en son yaptığınız işlemin sonucu olan sayıyı veriyor. Dolayısıyla en son elde ettiğiniz sayıyı mesela 25’e bölmek için şöyle bir şey yapabilirsiniz:

```
>>> _ / 25

97821621.0
```

Burada “_” işareti “2445540525” sayısını hafızasında tuttuğu için bu sayıyı pratik bir şekilde başka bir işlem içinde kullanmamıza imkan sağlıyor. Üstelik bu işleci sadece sayılarla kullanmak zorunda da değiliz. “_” işleci her türlü işlemin sonucunu hafızasında tutabilir. Mesela:

```
>>> "elma" + " armut"

'elma armut'

>>> _

'elma armut'
```

Hatta isterseniz bu işleci bir değişkene dahi atayabilirsiniz:

```
>>> sondeger = _

elma
```

Yalnız bu özellik sadece etkileşimli kabuk için geçerlidir. Bunu yazdığımız bir program içinde kullanamayız... Bu işlecin görevi sadece etkileşimli kabuk üzerinde çalışan programcının işini biraz kolaylaştırmaktır.

9.4 abs(), round(), min() ve max() Fonksiyonları

Python bize matematik ve aritmetik işlemlerinde kullanabileceğimiz `abs()`, `round()`, `min()` ve `max()` adında dört adet pratik fonksiyon sunar. Bu dört fonksiyonun yaptığı işler ilk bakışta çok gerekliymiş gibi görünmese de aslında bazı durumlarda işlerimizi bir hayli kolaylaştırmamızı sağlar. Öncelikle `abs()` fonksiyonundan başlayalım...

abs()

“abs”, İngilizce “absolute” kelimesinin kısaltmasıdır. Bu kelime Türkçe’de “mutlak” anlamına gelir. Bunun `abs()` fonksiyonundaki kullanımı ise “mutlak değer” anlamındadır. Dolayısıyla `abs()` fonksiyonu bir sayının mutlak değerini almamızı sağlar.

`abs()` fonksiyonunun kullanımı şöyledir:

```
>>> a = -8
>>> abs(a)

8
```

round()

İkinci fonksiyonumuz ise `round()`. Bu kelime Türkçe'de "yuvarlamak" anlamına gelir. Dolayısıyla `round()` fonksiyonunu kullanarak sayıların değerini yuvarlayabiliyoruz. Örneğin:

```
>>> a = 59 / 23
>>> round(a)

3
```

Böylece normalde küsürlü çıkacak bir bölme işlemini `round()` metodu yardımıyla yuvarlamış oluyoruz...

Başka bir örnek:

```
>>> round(150/4)

38
```

Bu fonksiyonu kullanırken, istersek noktadan sonra kaç basamak istediğimizi de belirtebiliriz:

```
>>> print(round(59/23, 0))

3.0

>>> print(round(59/23, 1))

2.6

>>> print(round(59/23, 2))

2.57

>>> print(round(59/23, 3))

2.565

>>> print(round(59/23, 4))

2.5652
```

Burada daha düzgün bir çıktı alabilmek için `print()` fonksiyonundan da yararlandığımıza dikkat edin. Eğer burada `print()` fonksiyonunu kullanmazsak bölme işlemlerinin sonucu insan gözüne "yabancı" görünecektir... İsterseniz burada `print()` fonksiyonu yerine `str()` fonksiyonundan da yararlanabilirsiniz düzgün çıktı elde edebilmek için:

```
>>> str(round(59/23, 4))

'2.5652'
```

Tabii `str()` fonksiyonuyla yaptığımız şey aslında sayıyı karakter dizisine çevirme işlemi olduğu için, yukarıdaki çıktının bir karakter dizisi olduğuna, dolayısıyla bunu kullanarak herhangi bir aritmetik işlem yapamayacağımıza dikkat ediyoruz...

min()

"min" kelimesi, tahmin edeceğiniz gibi "minimum" kelimesinin kısaltmasıdır. Bu fonksiyon bir sayı dizisi içindeki en küçük sayıyı verir:

```
>>> min(45, 90, 43, 23, 3353)

23
```

Bu fonksiyonu listelerle birlikte de kullanabiliriz:

```
>>> liste = [4234, 343543, 23323, 6161, 54564, 65675,
... 34243243, 4324, 4234]

>>> min(liste)

4234
```

max()

Evet, bu fonksiyon `min()` fonksiyonunun tam tersidir. Bir sayı dizisi içindeki en büyük sayıyı verir:

```
>>> max(45, 90, 43, 23, 3353)

3353
```

...veya...

```
>>> liste = [4234, 343543, 23323, 6161, 54564, 65675,
... 34243243, 4324, 4234]

>>> max(liste)

34243243
```

9.5 pow(), divmod() ve sum() Fonksiyonları

Bu bölümde, Python'un işlerimizi kolaylaştırmamız için bize sunduğu üç adet fonksiyondan söz edeceğiz. Bu fonksiyonlar `pow()`, `divmod()` ve `sum()`. Öncelikle `pow()` fonksiyonuyla başlayalım:

pow()

Hatırlarsanız "sayılar" konusunu işlerken bir sayının kuvvetini almak için şöyle bir yöntem izleyebileceğimizi söylemiştik:

```
x ** y
```

Burada "x" kuvvetini alacağımız sayıyı, "y" ise bu x sayısının kaçıncı kuvvetini alacağımızı gösteriyor. Örneğin 2 sayısının 3. kuvvetini almak istersek şunu yazıyoruz:

```
>>> 2**3

8
```

Python bu yöntemin dışında, bir sayının kuvvetini almak için bize `pow()` adlı bir fonksiyon daha sunuyor. Yukarıdaki işlemi `pow()` fonksiyonunu kullanarak şöyle yazabiliriz:

```
>>> pow(2, 3)

8
```

Burada tahmin edeceğiniz gibi parantez içindeki ilk parametre kuvvetini alacağımız sayıyı, ikinci parametre ise bu sayının kaçıncı kuvvetini alacağımızı gösteriyor...

divmod()

Şöyle bir bölme işlemi yaptığımızı düşünelim:

```
>>> 10 / 2

5.0
```

Gördüğünüz gibi burada bir "ondalık sayı" elde ediyoruz. Eğer elde ettiğimiz sayının ondalık değil de tamsayı olmasını istersek "//" işlecinden yararlanmamız gerektiğini biliyoruz:

```
>>> 10 // 2

5
```

Bir de şöyle bir işlem yaptığımızı düşünelim:

```
>>> 10 % 2

0
```

Burada da 10 sayısının 2'ye bölümünden kalan sayıyı buluyoruz. Demek ki 10 sayısı 2'ye bölündüğünde kalan "0" oluyormuş. Burada "%" işlecinin ne işe yaradığını "sayılar" konusunu işlerken öğrenmiştik...

İşte divmod() fonksiyonu yukarıda bahsettiğimiz bu iki işlemi birleştiriyor. Hemen bir örnekle bunu somutlaştıralım:

```
>>> divmod(10, 2)

(5, 0)
```

Gördüğünüz gibi `divmod()` fonksiyonu aynı anda noktasız bölme işlemi yapıyor ve bölme işleminden kalan sayıyı gösteriyor...

sum()

Şimdi göreceğimiz `sum()` adlı fonksiyon oldukça yararlı bir araçtır. Bu fonksiyon bir sayı dizisini alıp bu dizinin toplamını verir bize. Örneğin:

```
>>> liste = [34, 43, 32, 23, 23232, 32]

>>> sum(liste)

23396
```

Bu fonksiyonun yalnızca sayılarla kullanılabileceğini aklımızdan çıkarmıyoruz...

`sum()` fonksiyonu ayrıca ikinci bir parametre daha alabilir:

```
>>> sum(liste, 2)

23398
```

`sum()` fonksiyonu, listenin elemanlarını topladıktan sonra, elde edilen toplam değere ikinci parametreyi de ekliyor...

9.6 Bool Değerleri ve bool() Fonksiyonu

Daha önceki derslerimizde, üzerinde fazla durmamış olsak da aslında Bool değerlerini kullanmıştık. "True" ve "False" olarak ifade edilen değerlere Bool değerleri adı verilir (George Boole

adlı İngiliz matematikçi ve filozofun adından). “True” değeri sayısal olarak “1” ile gösterilirken, “False” değeri “0” ile gösterilir. Türkçe olarak söylemek gerekirse, “True” değerinin karşılığı “Doğru”, “False” değerinin karşılığı ise “Yanlış”tır... Biz bu değerleri mesela kümeler konusunu işlerken de görmüştük:

```
>>> a = set([1, 2, 3])
>>> b = set([1, 3, 4, 5])

>>> a.isdisjoint(b)

False
```

Burada a ve b ayrık kümeler olmadıkları için “Yanlış” anlamına gelen “False” değerini alıyoruz.

Bool değerleri temel olarak bir nesnenin doğruluk değerini sorgulamak için kullanılır. Yani bir şeyin doğru olup olmadığını anlamak için kullanıyoruz bu Bool değerlerini... Örneğin:

```
>>> a = 23

>>> if a == 23:
...     print("a'nın değeri 23'tür")
```

Burada aslında bir doğruluk değeri sorgulaması yapıyoruz (“Eğer a 23’e eşitse...”). Gerçi burada karşılaştırma işlemini yapan asıl öğe “==” işlecidir, ama aslında bu yapı içinde temel olarak Bool kavramından yararlanılıyor. Bunu etkileşimli kabukta test edebiliriz:

```
>>> a = 23

>>> a == 23

True

>>> a == 4

False
```

Burada olduğu gibi Bool kavramından karşılaştırma işlemlerinde yararlanmanın yanısıra, bir şeyin herhangi bir değere sahip olup olmadığını denetlemek için de Bool değerlerinden faydalanabiliriz. Örneğin:

```
>>> a = 23

>>> if a:
...     print("Evet, a'nın bir değeri var!..")
...
'Evet, a'nın bir değeri var!..'
```

Bir de şu örneğe bakalım:

```
>>> a = ""

>>> if a:
...     print("Evet, a'nın bir değeri var!..")
... else:
...     print("Hayır, a'nın herhangi bir değeri yok!")
...
'Hayır, a'nın herhangi bir değeri yok!"
```

Burada kullandığımız “if a:” ifadesi, “Eğer a doğru ise...” anlamına geliyor. Yani burada “Eğer a’nın Bool değeri True ise...” demiş oluyoruz. Bu durumu daha açık bir şekilde göstermek için

Python'daki `bool()` adlı fonksiyondan yararlanabiliriz. Bu fonksiyon, bize herhangi bir şeyin değerinin "True" mi yoksa "False" mi olduğunu söyleyecektir:

```
>>> a = ""

>>> bool(a)

False
```

Bir de şuna bakalım:

```
>>> a = "karakter dizisi"

>>> bool(a)

True
```

Dolayısıyla yukarıdaki `if` deyimini şöyle de yazabiliriz:

```
>>> a = ""

>>> if bool(a):
...     print("Evet, a'nın bir değeri var!..")
... else:
...     print("Hayır, a'nın herhangi bir değeri yok!")
```

a = "" ifadesinin "False" değerini vermesi bize Python'daki Bool değerleri hakkında önemli bir bilgi veriyor. Python'da bazı şeyler her zaman "False" değeri verir. Örneğin bütün boş veritipleri "False" değerine sahiptir. Örneğin:

```
>>> liste = []

>>> bool(liste)

False

>>> sozluk = {}

>>> bool(sozluk)

False

>>> demet = ()

>>> bool(demet)

False

>>> kume = set()

>>> bool(kume)

False
```

Yukarıdaki veritiplerinin "True" değeri verebilmesi için en az bir öğeye sahip olmaları gerekir...

Daha önce de dediğimiz gibi, sayısal olarak "True"nin değeri 1; "False"nin değeri ise 0'dır. Bu durumu şu kodlarla daha net bir şekilde görebiliriz:

```
>>> True + 1

2
```

Burada “True”nin değeri 1 olduğu için, “True + 1” ifadesi “2” sonucunu veriyor. Bir de şuna bakalım:

```
>>> False + 1

1
```

Burada ise “False”nin değeri 0 olduğu için, “False + 1” işleminin sonucu 1 oluyor...

Bu bilgiye göre, değeri 0 olan bir değişken False; değeri 1 olan bir değişken ise True olacaktır:

```
>>> a = 0

>>> bool(a)

False

>>> b = 1

>>> bool(b)

True
```

Ancak bu durumun sadece değişkenler için geçerli olduğunu unutmuyoruz. Liste, sözlük, demet, vb. veri tiplerinde durum biraz farklıdır:

```
>>> liste = [0]

>>> bool(liste)

True
```

Bunun dışında özel olarak “None” değeri de her zaman “False”dir:

```
>>> a = None

>>> bool(a)

False
```

Boş veritiplerinin False, dolu veritiplerinin True değeri vermesinden yola çıkarak şöyle bir uygulama yazabiliriz:

```
liste = []

print("""
Lütfen bir sayı girin. Her veri girişinden sonra Enter
tuşuna basın. İşlemi bitirip programdan
çıkmak için ise yine Enter tuşuna basın.""")

while True:
    veri = input("Sayı: ")
    if veri:
        try:
            sayı = int(veri)
            liste.append(sayı)
```

```
        except ValueError:
            continue

    else:
        toplam = sum(liste)
        print("girdiğiniz sayıların toplamı: ", toplam)
        print("tekrar görüşmek üzere!")
        break
```

Şimdi bu kodları satır satır inceleyelim...

Öncelikle "liste = []" satırı yardımıyla boş bir liste oluşturuyoruz. Kullanıcıdan alacağımız verileri bu liste içinde toplayacağız.

Ardından bir `print()` fonksiyonu içinde programımızın ne iş yapacağını, kullanıcıdan ne beklediğimizi anlatan bir karakter dizisi oluşturuyoruz. Buna göre, kullanıcı ekrana bir sayı girecek ve her veri girişinden sonra Enter tuşuna basacak. Ayrıca işlemin sonucunu alıp programdan çıkmak için de yine Enter tuşuna basması gerekecek...

Daha sonra bir `while` döngüsü oluşturuyoruz. Burada kullandığımız "while True:" ifadesine dikkat edin. Bu ifade, bir kod dizisini sürekli bir döngüye sokmak için kullanabileceğimiz standart bir yoldur. Türkçe olarak söylemek gerekirse, "while True:" ifadesi "Doğru olduğu müddetçe..." anlamına gelir. Bu sayede programımıza şu komutu vermiş oluyoruz:

"Doğru olduğu müddetçe aşağıdaki komutları çalıştırmaya devam et..."

Peki ne doğru olduğu müddetçe? Neyin doğru olduğunu açıkça belirtmediğimiz için Python burada "her şeyi doğru" kabul ediyor... Yani bir nevi, "aksi belirtilmediği sürece aşağıdaki komutları çalıştırmaya devam et!" emrini yerine getiriyor.

Sonraki satırda "veri = input("Sayı: ")" kodu ile kullanıcıdan veri alma işlemini başlatıyoruz. Burada henüz kullanıcıdan aldığımız veriyi `int()` fonksiyonu yardımıyla tamsayıya çevirmediğimize dikkat edin. Eğer daha bu aşamada dönüştürme işlemini yaparsak, daha sonra kullanıcının programdan çıkmak için kullanacağı Enter tuşunu yakalayamayız. Çünkü böyle bir durumda kullanıcı Enter tuşuna basarsa programımız hata verecektir.

Bundan sonraki adımda "if veri:" satırını yazıyoruz. Bunun, "eğer veri adlı karakter dizisi True değerine sahipse..." anlamına geldiğini biliyoruz. Zira bildiğiniz gibi, eğer veri adlı karakter dizisi hiçbir şey içermezse, yani boşsa, sahip olduğu Bool değeri "False" olacaktır. Yani eğer kullanıcımız hiçbir sayı girmeden Enter tuşuna basarsa, veri adlı karakter dizisinin içeriği boş olacağı için bu karakter dizisi "False" değerine sahip olacaktır. Dolayısıyla eğer kullanıcımız bir sayı girip Enter tuşuna basarsa, veri adlı karakter dizisinin Bool değeri True olacağı için "if veri:" bloğu içindeki kodlar işletilecektir. Yok eğer kullanıcımız herhangi bir sayı girmeden Enter tuşuna basarsa aşağıdaki "else:" bloğu içindeki kodlar çalıştırılacaktır...

"if veri:" kodunu yazdıktan sonra bir `try... except...` bloğu oluşturuyoruz. Sayıya dönüştürme işlemini bu blok içinde gerçekleştireceğiz. "sayı = int(veri)" satırı yardımıyla, kullanıcıdan aldığımız karakter dizisini sayıya çeviriyoruz. Hemen ardından gelen "liste.append(sayı)" satırı ise, kullanıcıdan alıp tamsayıya çevirdiğimiz veriyi, en başta oluşturduğumuz listeye eklememizi sağlıyor.

Sonraki satırda ise `except...` bloğumuzu yazıyoruz. Bir önceki `try...` bloğu içine yazdığımız `int()` fonksiyonunun, kullanıcının sayı yerine harf girmesi durumunda "ValueError" hatası vereceğini bildiğimiz için bu blok içinde bu hata tipini yakalıyoruz. Eğer kullanıcı sayı yerine harf girer ve ValueError hatası alınmasına yol açarsa, `continue` ifadesi yardımıyla döngünün en başına geri gidiyoruz.

Sıra geldi `else:` bloğumuzu yazmaya... Kodlarımızın başında "if veri:" şeklinde bir şey yazmıştık. Bu kod, kullanıcının ekrana bir veri girip girmediğini denetliyordu. İşte kullanıcının

herhangi bir veri girmediği durumlar için bu `else:` bloğunu yazıyoruz. Eğer kullanıcı hiçbir veri girmeden "Enter" tuşuna basarsa bu blok içindeki kodlar işletilecektir.

`else:` bloğu içinde yaptığımız ilk iş, listeye eklediğimiz sayıların hepsini toplamak olacak... Bunu daha önce öğrendiğimiz `sum()` fonksiyonu yardımıyla yapıyoruz. Ardından da `print()` fonksiyonunu kullanarak bu toplam değeri kullanıcıya gösteriyoruz ve kendisine veda ettikten sonra `break` ifadesi yardımıyla da programdan çıkıyoruz.

9.7 Bool İşleçleri (Boolean Operators)

Bu bölümde Python'daki "Bool işleçleri" konusundan söz edeceğiz. Burada değineceğimiz işleçler `and`, `or` ve `not` adlı Bool işleçleridir... Bu işleçlere ayrıca "mantık işleçleri" (logical operators) adı da verilir.

and

Bu kelime Türkçe'de "ve" anlamına gelir. Şöyle bir örnek verelim:

```
>>> a = 4
>>> b = 12

>>> a == 4 and b == 12

True
```

Yukarıdaki örnekte and işleci, a ve b'nin her ikisinin de değeri "True" olursa, "True" sonucunu verecektir. Eğer karşılaştırma işlemlerinden herhangi birisi "False" ise, sonuç da "False" olacaktır:

```
>>> a == 3 and b == 12

False
```

Lisede Mantık dersleri almış olanlar bu işleci gayet iyi tanırlar. Eğer iki önerme birbirine "ve" ile bağlanmışsa, doğru sonuç elde edebilmemiz için bu iki önermenin her ikisinin de doğru olması gerekir.

and işleçlerini Python'da mesela parola ve kullanıcı adı denetlemek için kullanabiliriz. Bildiğiniz gibi, bir sisteme kullanıcı adı ve parola ile giriş yapabilmek için hem kullanıcı adının hem de parolanın doğru olması gerekir. Örnek olarak şöyle bir uygulama yazalım:

```
kullanıcı_adı = "istihza"
parola = "parola"

print("""Sitemize hoşgeldiniz! Hizmetlerimizden yararlanmak için
lütfen kullanıcı adınızı ve parolanızı giriniz.""")

while True:
    kull = input("kullanıcı adı: ")
    par = input("parola: ")

    if kull == kullanıcı_adı and par == parola:
        print("Sisteme hoşgeldiniz!")
        break

    else:
        print("Kullanıcı adınız veya parolanız yanlış! Tekrar deneyin.")
        continue
```

Burada kullanıcının sisteme girebilmesi için hem parolanın hem de kullanıcı adının önceden belirlediğimiz kullanıcı adı ve parolayla eşleşmesi lazım. Bunlardan herhangi biri eşleşmezse kullanıcı sisteme alınmayacaktır.

or

Bu kelime "veya" anlamına gelir. and işlecini anlatırken, bir sonucun "True" olabilmesi için her iki koşulun da "True" olması gerektiğini söylemiştik. `or` işlecinde ise, koşullardan herhangi birinin "True" olması, sonucun "True" olması için yeterli olacaktır. Mesela yukarıdaki örneği `or` işleci ile yazarsak, kullanıcı adı veya paroladan herhangi birinin doğru olması, kullanıcının sisteme girmesi için yeterli olacaktır:

```
kullanıcı_adı = "istihza"
parola = "parola"

print("""Sitemize hoşgeldiniz! Hizmetlerimizden yararlanmak için
lütfen kullanıcı adınızı ve parolanızı giriniz.""")

while True:
    kull = input("kullanıcı adı: ")
    par = input("parola: ")

    if kull == kullanıcı_adı or par == parola:
        print("Sisteme hoşgeldiniz!")
        break

    else:
        print("Kullanıcı adınız ve parolanız yanlış! Lütfen tekrar deneyin.")
        continue
```

Dediğimiz gibi, `or` işlecinin "True" sonucu verebilmesi için önermelerden herhangi birinin "True" olması yeterli olacaktır. Eğer önermelerden her ikisi de yanlış ise sonucumuz "False" olur:

```
>>> a = 12
>>> b = 2

>>> a == 12 or b == 4

True

>>> a == 5 or b == 43

False
```

not

Bu bölümde işleyeceğimiz son işlecimiz `not` adlı Bool işlecidir. Bu kelime "değil" anlamına gelir ve cümleye olumsuzluk anlamı katar. Örneğin:

```
try:
    soru = int(input("bir çift sayı giriniz: "))

    if not soru % 2 == 0:
        print("Çift sayı girmediniz!")

    else:
        print("Çift sayı girdiniz!")
```

```
except ValueError:
    print("Sayı... Sadece sayı...")
```

Burada “if not soru % 2 == 0:” ifadesiyle şöyle demiş oluyoruz:

Eğer soru değişkeninin değerini 2’ye böldüğümüzde kalan 0 değil ise...

Benzer bir şeyi etkileşimli kabuk üzerinde test edelim:

```
>>> a = 13

>>> not a % 2 == 0

True
```

Burada, “a değişkeninin değeri 2’ye bölündüğünde kalan 0 değildir” gibi bir önerme veriyoruz. Python da cevap olarak bize “True” (Doğru”) diyor... Çünkü gerçekten de a değişkeninin değeri olan 13 sayısını 2’ye böldüğümüzde kalan sayı 0 değildir... Bir de şuna bakalım:

```
>>> a = 34

>>> not a % 2 == 0

False
```

Burada da, a değişkeninin değeri olan 34 sayısı 2’ye bölündüğünde kalan sayı 0 olduğu için Python, “a 2’ye bölündüğünde kalan sayı 0 değildir” önermemize “False” (Yanlış) cevabını veriyor...

9.8 all() ve any() Fonksiyonları

Bir önceki bölümde and ve `or` işleçlerini öğrenmiştik. Python bize bu and ve `or` işleçlerinin kullanımı için bazı kolaylıklar sağlar. İşte bu bölümde Python’un bu konuda bize sunduğu kolaylıkların ne olduğunu göreceğiz.

all()

Bu fonksiyon bir dizi içindeki bütün değerlerin Bool değerlerine göre doğruluklarını aynı anda denetlememizi sağlar. Eğer o dizi içindeki bütün değerler “True” ise bu fonksiyon “True” sonucunu verecektir. Hatırlarsanız bir önceki bölümde kullanıcı adı ve parola soran bir uygulama yazmıştık. O uygulamayı `all()` fonksiyonunu kullanarak şöyle de yazabiliriz:

```
kullanıcı_adı = "istihza"
parola = "parola"

print("""Sitemize hoşgeldiniz! Hizmetlerimizden yararlanmak için
lütfen kullanıcı adınızı ve parolanızı giriniz.""")

while True:
    kull = input("kullanıcı adı: ")
    par = input("parola: ")

    if all([kull == kullanıcı_adı, par == parola]):
        print("Sisteme hoşgeldiniz!")
        break

    else:
```

```
        print("Kullanıcı adınız veya parolanız yanlış!")
        print("Tekrar deneyin.")
        continue
```

Burada `all()` fonksiyonu içinde bir liste kullandığımıza dikkat edin. Çünkü all() fonksiyonu sadece tek bir parametre alır. Bu fonksiyona birden fazla parametre atayabilmek için parametreleri bir liste veya demet içinde toplamanız gerekir...

any()

`any()` fonksiyonu ise bir dizi içindeki herhangi bir öğenin doğruluk değeri "True" ise sonuç olarak "True" verecektir. Yine yukarıdaki örneği `or` ile yazmak yerine, istersek `any()` fonksiyonundan yararlanabiliriz:

```
kullanıcı_adı = "istihza"
parola = "parola"

print("""Sitemize hoşgeldiniz! Hizmetlerimizden yararlanmak için
lütfen kullanıcı adınızı ve parolanızı giriniz.""")

while True:
    kull = input("kullanıcı adı: ")
    par = input("parola: ")

    if any([kull == kullanıcı_adı, par == parola]):
        print("Sisteme hoşgeldiniz!")
        break

    else:
        print("Kullanıcı adınız veya parolanız yanlış! Lütfen tekrar deneyin.")
        continue
```

9.9 Liste Üreteçleri (List Comprehensions)

Hatırlarsanız, sözlükler konusunu işlerken şöyle bir kod kullanmıştık:

```
>>> [i for i in dir(dict) if "_" not in i]
```

Bu kod, birkaç satırda yapabileceğimiz işlemleri tek satıra indirmemizi sağlıyordu. Ayrıca bu yapının normal bir `for` döngüsüne göre daha performanslı ve hızlı olduğunu da söylemek gerek... Bu tür yapılara Python'da "Liste Üreteçleri" (List Comprehensions) adı verilir. Liste üreteçleri, Python'da bir değerler dizisinden liste oluşturmanın etkili bir yöntemidir.

Diyelim ki elimizde "elma" adlı bir karakter dizisi var. Yine diyelim ki biz bu karakter dizisinin bütün öğelerini bir liste haline getirmek istiyoruz. Yani elde etmek istediğimiz şey şu:

```
['e', 'l', 'm', 'a']
```

Liste üreteçlerini kullanmadan bu görevi şöyle halledebiliriz:

```
>>> liste = []
>>> a = "elma"

>>> for i in a:
...     liste.append(i)

>>> print(liste)
```

```
['e', 'l', 'm', 'a']
```

Gördüğünüz gibi öncelikle boş bir liste oluşturuyoruz. Ardından da "a" adlı değişken üzerinde bir for döngüsü kurarak bunun bütün öğelerini append() metodu yardımıyla "liste"ye ekliyoruz. Ancak aynı işlemi yapmanın çok daha kolay ve verimli bir yolu vardır. Liste üreteçlerini kullanarak bu işlemi tek satırda halledebiliriz:

```
>>> [i for i in a]

['e', 'l', 'm', 'a']
```

Burada liste üreteçlerinin sözdizimi ile `for` döngülerinin sözdizimi arasındaki benzerliğe dikkat edin. Gerçekten de liste üreteçleri görünüm olarak `for` döngülerinin tek satıra indirilmiş hali gibi görünür. Şu `for` döngüsünün şöyle evrildiğini düşünün:

`for` döngümüz...

```
>>> for i in range(10):
...     print(i)
```

liste üretecinin taslağı:

```
>>> print(i) for i in range(10)
```

liste üretecinin kendisi:

```
>>> [i for i in range(10)]
```

Elbette liste üreteçlerinin yapabildiği tek şey bu olsaydı dünya daha güzel bir yer olmazdı... Çünkü aynı şeyi sadece `list()` fonksiyonunu kullanarak da yapabiliriz!

```
>>> list(a)

['e', 'l', 'm', 'a']
```

...veya...

```
>>> list(range(10))

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

Liste üreteçlerinin gücü daha karmaşık işlemlerde ortaya çıkar. Mesela bir öğe dizisini belli bir ölçüte göre süzmek istediğimizde liste üreteçlerinden yararlanabiliriz. Örneğin bir isim dizisi içindeki "A" harfi ile başlayan isimleri süzelim:

```
>>> isimler = ["Mehmet", "Ayşe", "Süleyman", "Ali", "Kenan",
... "Kerem", "Aycan", "Ayhan", "Metin", "Ayla"]

>>> [i for i in isimler if i[0] == "A"]

['Ayşe', 'Ali', 'Aycan', 'Ayhan', 'Ayla']
```

Eğer bunu bir for döngüsü ile halletmek isteseydik şöyle yazacaktık bu kodları:

```
>>> for i in isimler:
...     if i[0] == "A":
...         print(i)
...
Ayşe
```

```
Ali
Aycan
Ayhan
Ayla
```

Yalnız burada elde ettiğimiz şeyin bir liste olmadığına dikkat edin. Eğer süzülen öğeleri bir liste haline getirmek istersek aşağıdaki yazım tarzını benimsememiz gerekir:

```
>>> liste = []

>>> for i in isimler:
...     if i[0] == "A":
...         liste.append(liste)

>>> print(liste)

[’Ayşe’, ’Ali’, ’Aycan’, ’Ayhan’, ’Ayla’]
```

Liste üreteçlerini, bir önceki bölümde öğrendiğimiz `all()` ve `any()` fonksiyonlarıyla birlikte de kullanabiliriz. Mesela elimizde şöyle bir liste olsun:

```
>>> a = [10, 15, 6, 3, 32, 45, 25]
```

Diyelim ki bu listedeki sayıların hepsinin 20’den büyük olup olmadığını denetlememiz gerekiyor. O zaman şöyle bir kod yazabiliriz:

```
>>> all([i > 20 for i in a])

False
```

Yaptığımız işlemden “False” sonucunu aldık. Demek ki listedeki bütün sayılar 20’den büyük değilmiş...

Python’daki liste üreteçleri pek çok konuda işlerimizi kolaylaştırmamıza yardımcı olur. Mesela bir dizi içindeki bütün sayıları hızlı bir şekilde belli bir işleme tabi tutmak istersek liste üreteçlerinden yararlanabiliriz:

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

>>> [pow(i, 3) for i in liste]

[1, 8, 27, 64, 125, 216]
```

Bu kodlar, liste içindeki bütün sayıların tek tek 3. kuvvetlerini hesaplamamızı sağladı.

9.10 Python’da Kodlara Yorum Eklemek

Programcılıkta en zor şey başkasının yazdığı kodları okuyup anlamaktır. Hatta yazılmış bir programı düzeltmeye çalışmak, bazen o programı sıfırdan yazmaktan daha zor olabilir. Bunun nedeni, program içindeki kodların ne işe yaradığını anlamanın zorluğudur. Programı yazan kişi kendi düşünüşüne göre bir yol izlemiş ve programı geliştirirken karşılaştığı sorunları çözmek için kimi yerlerde enteresan çözümler üretmiş olabilir. Ancak kodlara dışardan bakan birisi için o programın mantık düzenini ve içindeki kodların tam olarak ne yaptığını anlamak bir hayli zor olacaktır. Böyle durumlarda, kodları okuyan programcının en büyük yardımcısı, programı geliştiren kişinin kodlar arasına eklediği notlar olacaktır. Tabii programı geliştiren kişi kodlara yorum ekleme zahmetinde bulunmuşsa...

Python'da yazdığımız kodların başkalarının da anlayabilmesini sağlamak için, programımızın yorumlarla desteklenmesi tavsiye edilir. Elbette programınızı yorumlarla desteklemeseniz de programınız sorunsuz bir şekilde çalışacaktır. Ama programı yorumlarla desteklemek en azından nezaket gereğidir.

Ayrıca işin başka bir boyutu daha var. Sizin yazdığınız kodları nasıl başkaları okurken zorlanıyorsa, kendi yazdığınız kodları okurken siz bile zorlanabilirsiniz... Özellikle uzun süredir ilgilenmediğiniz eski programlarınızı gözden geçirirken böyle bir sorunla karşılaşabilirsiniz. Programın içindeki bir kod parçası, programın ilk yazılışının üzerinden 5-6 ay geçtikten sonra size artık hiçbir şey ifade etmiyor olabilir... Kodlara bakıp, "Acaba burada ne yapmaya çalışmışım?" diye düşündüğünüz zamanlar da olacaktır... İşte bu tür sıkıntıları ortadan kaldırmak veya en aza indirmek için kodlarımızın arasına açıklayıcı notlar ekleyeceğiz.

Python'da yorumlar "#" işareti ile gösterilir. Yorumlarla desteklenmiş örnek bir kod parçası şöyle olabilir:

```
#Boş bir liste oluşturuyoruz. Bu liste
#kullanıcının girdiği verileri tutacak
liste = []

#Şimdi kullanıcıya adını soruyoruz...
ad = input("Lütfen adınızı giriniz: ")

#Kullanıcıdan aldığımız veriyi listeye ekliyoruz.
liste.append(ad)
```

Burada dikkat edeceğimiz nokta her yorum satırının başına "#" işaretini koymayı unutmamaktır.

Yazdığımız yorumlar Python'a hiç bir şey ifade etmez. Python bu yorumları tamamen görmezden gelecektir. Bu yorumlar bilgisayardan ziyade programcı için bir anlam taşır.

Elbette yazdığınız yorumların ne kadar faydalı olacağı, yazdığınız yorumların kalitesine bağlıdır... Dediğimiz gibi, yerli yerinde kullanılmış yorumlar bir programın okunaklılığını artırır, ama her tarafı yorumlarla kaplı bir programı okumak da bazen hiç yorum girilmemiş bir programı okumaktan daha zor olabilir!...

Yukarıda olduğu gibi yorumlarımızı ilgili olduğu koddan önce yazmak yerine, yorumlarımızı kodlarımızın farklı yerlerine de ekleyebiliriz:

```
liste = [] #boş bir liste oluşturuyoruz...
```

Python'da yorum eklerken önemli olan şey, kaş yapmaya çalışırken göz çıkarmamaktır. Yani yorumlarımızı, bir kodun okunaklılığını artırmaya çalışırken daha da bozmayacak şekilde yerleştirmeye dikkat etmeliyiz...

Python'da yorumlar asıl amaçlarının dışında da bazı başka amaçlara hizmet edebilir.

Örneğin, yazdığımız programa bir özellik eklemeyi düşünüyoruz, ama henüz bu özelliği yeni sürüme eklemek istemiyoruz. O zaman şöyle bir şey yapabiliriz:

```
#Boş bir liste oluşturuyoruz. Bu liste
#kullanıcının girdiği verileri tutacak
liste = []

#Şimdi kullanıcıya adını soruyoruz...
ad = input("Lütfen adınızı giriniz: ")

#Kullanıcıdan aldığımız veriyi listeye ekliyoruz.
liste.append(ad)
```

```
#Kullanıcı "p" harfine basarsa liste içeriğini ekrana dökelim.
#if ad == "p":
#    print(liste)
```

Burada, programa henüz eklemek istemediğimiz bir özelliği, yorum içine alarak şimdilik iptal ediyoruz (İngilizce'de bu "yorum içine alma" işlemine "comment out" deniyor...). Python yorum içinde bir kod bile yer alsa o kodları çalıştırmayacaktır. Çünkü Python "#" işareti ile başlayan satırların içeriğini görmez (`#!/usr/bin/env python` satırı hariç).

Peki eklemek istemediğimiz özelliği yorum içine almaktansa doğrudan silsek olmaz mı? Elbette olur. Ama programın daha sonraki bir sürümüne ilave edeceğimiz bir özelliği yorum içine almak yerine silecek olursak, vakti geldiğinde o özelliği nasıl yaptığımızı hatırlamakta zorlanabiliriz!... Hatta bir süre sonra programımıza hangi özelliği ekleyeceğimizi dahi unutmuş olabiliriz... "Hayır, ben hafızama güveniyorum!" diyorsanız karar sizin!...

Yorum içine alarak iptal ettiğiniz bu kodları programa ekleme vakti geldiğinde yapacağınız tek şey, kodların başındaki "#" işaretlerini kaldırmak olacaktır. Hatta bazı metin düzenleyiciler bu işlemi tek bir tuşa basarak da gerçekleştirme yeteneğine sahiptir. Örneğin IDLE ile çalışıyorsanız, yorum içine almak istediğiniz kodları fare ile seçtikten sonra ALT+3 tuşlarına basarak ilgili kodları yorum içine alabilirsiniz. Bu kodları yorumdan kurtarmak için ise ilgili alanı seçtikten sonra ALT+4 tuşlarına basmanız yeterli olacaktır ("yorumdan kurtarma" işlemine İngilizce'de "uncomment" diyorlar).

Bütün bunların dışında, isterseniz yorum işaretini kodlarınızı süslemek için dahi kullanabilirsiniz:

```
############################################################
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
#                     FALANCA v.1                          #
#                 Yazan: Keramet Su                        #
#                   Lisans: GPL v2                         #
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
############################################################

#Boş bir liste oluşturuyoruz. Bu liste
#kullanıcının girdiği verileri tutacak
liste = []

#Şimdi kullanıcıya adını soruyoruz...
ad = input("Lütfen adınızı giriniz: ")

#Kullanıcıdan aldığımız veriyi listeye ekliyoruz.
liste.append(ad)
```

Yani kısaca, Python'un görmesini, çalıştırmasını istemediğimiz her şeyi yorum içine alabiliriz... Unutmamamız gereken tek şey, yorumların yazdığımız programların önemli bir parçası olduğu ve bunları mantıklı-makul bir şekilde kullanmamız gerektiğidir...

9.11 Karakter Dizilerini Biçimlendirme

Python'daki özel bir fonksiyon/metot yardımıyla karakter dizilerine müdahale edebilir, bunları istediğimiz şekilde biçimlendirebiliriz.

Öncelikle elimizdekilerle neler yapabildiğimize bakalım. Şimdiye kadar öğrendiklerimizi kullanarak şöyle bir kod yazabiliyoruz:

```
>>> lider = input("Şampiyonun ismini giriniz: ")

>>> print("Bu senenin şampiyonu", lider, "Tebrikler.")
```

Burada kullanıcımız "lider" değişkeni için ne yazarsa, `print()` fonksiyonu içinde ilgili yere o kelime yerleştirilecektir.

Gördüğünüz gibi, yukarıdaki yapıyı kullanarak isteğimizi yerine getirebiliyoruz. Ama Python'da bu tür işlemleri ve daha fazlasını çok daha esnek bir biçimde yerine getirmenin yolları vardır.

Öncelikle yukarıda yaptığımız örneğe bakalım. Dediğim gibi, yukarıdaki kodlar temel olarak istediğimiz şeyi yerine getiriyor. Ama tam olarak değil... Diyelim ki kullanıcı burada "Şampiyonun ismini giriniz: " cümlesine karşılık olarak "Sarıyer" yazdı... O zaman çıktımız şöyle görünecektir:

```
Bu senenin şampiyonu Sarıyer Tebrikler.
```

Burada aslında çıktı düzgün değil. Türkçe'nin kurallarına göre bu çıktının şöyle olması gerekiyordu:

```
Bu senenin şampiyonu Sarıyer. Tebrikler.
```

Yukarıdaki kodları kullanarak "Sarıyer" kelimesine yapışık şekilde bir nokta işareti koyabilmek için deveye hendek atlatmanız gerekir...

```
>>> lider = input("Şampiyonun ismini giriniz: ")

>>> print("Bu senenin şampiyonu", lider, "\b.",  "Tebrikler.")
```

Buradaki "\b" işareti henüz öğrenmediğimiz bir kaçış dizisidir ("escape sequence"). Bu konuyu birkaç bölüm sonra göreceğiz.

Peki ya kullanıcının girdiği kelimeyi tırnak içine almak isterseniz ne yapacaksınız? İşte bunu yapmanız yukarıdaki yöntemle imkansızdır.

Bu tür işlemleri yapabilmek için karakter dizilerinin `format()` adlı metodunu devreye sokacağız. Yukarıdaki uygulamayı temel alarak basit bir örnek verelim:

```
>>> lider = input("Şampiyonun ismini giriniz: ")

>>> print("Bu senenin şampiyonu {0}. Tebrikler.".format(lider))
```

Bu yapı gözünüze biraz karışık mı göründü? Hiç endişelenmeyin aslında çok basittir kullanımı. İsterseniz daha belirgin bir örnek verelim:

```
>>> "{0}".format("istihza")

'istihza'
```

Gördüğünüz gibi, `format()` metodunun, daha önce öğrendiğimiz herhangi bir metottan farkı yok... Örneğin listeleri anlatırken liste metotlarından da bahsetmiştik. Liste metotları "liste.metot()" şeklinde kullanılıyordu. `format()` metodu da böyledir. Bu da "karakter_dizisi.format()" şeklinde kullanılır.

Burada "{0}" şeklinde gösterdiğimiz ifade, `format()` metodunun parantezi içinde yer alan "istihza" adlı karakter dizisinin yerini tutuyor. Parantez içine ne yazarsanız, "{0}" ifadesinin yerinde o görünecektir:

```
>>> lider = input("Şampiyonun ismini giriniz: ")

>>> print("{0}".format(lider))
```

`format()` metodu içindeki ifadenin aslında bir "demet" olduğunu söylersek sanırım "{0}" sözünün anlamı biraz daha netleşecektir. Buradaki "0" sayısı, parantez içinde yazdığımız demetin öğelerinin sırasını gösteriyor. Şöyle bir örnek verelim:

```
>>> print("Yıllardan {0}, aylardan {1}".format(1980, "Mayıs"))

Yıllardan 1980, aylardan Mayıs
```

Gördüğünüz gibi, alışık olduğumuz şekilde saymaya sıfırdan başlıyoruz... "(1980, "Mayıs")" demetinin 0. ve 1. öğelerini karakter dizisi içine tek tek yerleştiriyoruz. Burada önemli nokta şu: Karakter dizisi içinde belirlediğimiz biçimlendirme işaretlerinin sayısı kadar öğeyi parantez içindeki demete yazıyoruz.

Bu arada, biçimlendirme işaretlerinin sayı sırasına göre yazılması şart değil. Yani şöyle bir şey de yapabiliriz:

```
>>> print("Yıllardan {1}, aylardan {0}".format(1980, "Mayıs"))
```

Tabii bu şekilde elde ettiğimiz çıktıdaki öğelerin yeri de farklı olacaktır...

```
Yıllardan Mayıs, aylardan 1980
```

En baştaki örneğimize geri dönelim:

```
>>> lider = input("Şampiyonun ismini giriniz: ")

>>>print("Bu senenin şampiyonu", lider, "Tebrikler.")
```

Burada "lider" değişkenini karakter dizisi içinde mesela tırnak içine alamıyoruz... Ama artık `format()` adlı metodu bildiğimize göre bu işlemi yapmak bizim için çocuk oyuncağı:

```
>>> lider = input("Şampiyonun ismini giriniz: ")

>>> print("Bu senenin şampiyonu '{0}'. Tebrikler.".format(lider))
```

Gördüğünüz gibi, karakter biçimlendirme işaretleri bize epey esneklik kazandırıyor.

Bu metoda alışmak için çok basit bir örnek daha verelim:

```
>>> print("{0}; {1} ve {2}'i  seviyor!".format("Ali", "Ayşe", "Zeynep"))

Ali; Ayşe ve Zeynep'i seviyor!
```

Gördüğünüz gibi, `format()` metodu istediğimiz sayıda öğe alabiliyor.

Biçimleyici olarak sadece sayılardan yararlanmak zorunda da değiliz. Eğer istersek sayı yerine daha anlamlı kelimeler de kullanabiliriz:

```
>>> lider = input("Şampiyonun ismini giriniz: ")
>>> ikinci = input("İkinci olan takımın ismini giriniz: ")

>>> print("Şampiyon {bir}. {iki} ikinci oldu...".format(bir=lider, iki=ikinci))
```

Burada biçimlendirme işaretleri olarak "bir" ve "iki" değişkenlerini kullandığımıza dikkat edin. Daha sonra da `format()` metodunun parantezleri içinde bu "bir" ve "iki" adlı değişkenleri tanımlıyoruz...

Şöyle bir örnek verelim:

```
>>> a = 10
>>> b = 3

>>> print("{0} ile {1} bölünürse {2} elde edilir.".format(a, b, a/b))

10 ile 3 bölünürse 3.33333333333 elde edilir.
```

Burada istersek ”.” işaretinden sonra gelen basamak sayısını keyfimize göre ayarlayabiliriz:

Eğer noktadan sonra herhangi bir sayı olmasını istemezsek şu kodları kullanıyoruz:

```
>>> print("{0} ile {1} bölünürse {2:.0f} elde edilir.".format(a, b, a/b))

10 ile 3 bölünürse 3 elde edilir.
```

Burada dilimleme (slicing) işlemine benzer bir şey yaptığımıza dikkat edin. Küme parantezleri içinde yer alan ”.0f” ifadesi noktadan sonra kaç basamak gösterileceğini belirliyor. Buna göre noktadan sonra sıfır sayıda basamak olacak... Bir de şuna bakalım:

```
>>> print("{0} ile {1} bölünürse {2:.1f} elde edilir.".format(a, b, a/b))

10 ile 3 bölünürse 3.3 elde edilir.
```

Burada noktadan sonra bir basamak istediğimizi belirttik. ”.1f” ifadesi içindeki sayıyı artırarak basamak sayısını istediğiniz şekle getirebilirsiniz. Buradaki “f” harfi “fixed-point number” (sabit noktalı sayı) ifadesinin baş harfi... Bu harf, sayımızın sabit noktalı bir sayı cinsinden verileceğini gösteriyor.

Python 3.0 ile birlikte gelen bu `format()` adlı metot oldukça gelişmiş özellikler sunar. Hatta bu yapı Python içinde minik bir dil gibidir. Biz bu bölümde bu metodun bazı temel özelliklerini vermekle yetindik. İlerki bir bölümde bu metot daha ayrıntılı bir şekilde incelenecektir.

9.12 enumerate() Fonksiyonu

Bu fonksiyon Python’daki en pratik ve yararlı metotlardan bir tanesidir. “enumerate” kelimesi Türkçe’de “numaralandırmak” anlamına gelir. İsminden de anlaşılacağı gibi bu fonksiyon yardımıyla numaralandırma işlemleri yapacağız.

Örneğin, Python’daki bir veri tipinin bütün metotlarını nasıl sıralayacağımızı biliyoruz:

```
>>> dir(list)
```

Bu komut “list” (liste) adlı veri tipinin bütün metotlarını listeleyecektir. İşte bu metotları numaralandırmak istememiz halinde `enumerate()` adlı fonksiyon yardımımıza koşacaktır.

İşe bu metodu biraz tanıyarak başlayalım. Mesela şöyle bir şey yazalım:

```
>>> enumerate("istihza")

<enumerate object at 0x00C16238>
```

Gördüğünüz gibi, `enumerate()` fonksiyonunu doğrudan kullandığımızda bize bir “numaralandırma nesnesi” (enumerate object) veriyor.

Bu nesnenin içeriğini görmek için `list()` fonksiyonunu kullanabiliriz. Hatırlarsanız `range()` fonksiyonu için de `list()`‘ten yararlanmıştık...

```
>>> list(enumerate("istihza"))

[(0, 'i'), (1, 's'), (2, 't'), (3, 'i'), (4, 'h'), (5, 'z'), (6, 'a')]
```

Şimdi bu çıktıyı dikkatle inceleyelim. Burada bir liste içinde her biri bir sayı ve harften oluşan, iki öğeli demetler görüyoruz. Bu tabloyu daha net görebilmek için `for` döngülerinden yararlanabiliriz:

```
>>> for i in enumerate("istihza"):
...     print(i)
...
(0, 'i')
(1, 's')
(2, 't')
(3, 'i')
(4, 'h')
(5, 'z')
(6, 'a')
```

Gördüğünüz gibi, bir sayı ve bir harften oluşan çiftler elde ediyoruz. Alıştırma olsun diye, `enumerate()` fonksiyonunu liste üreteçleri (list comprehensions) ile birlikte kullanalım:

```
>>> [i for i in enumerate("istihza")]

[(0, 'i'), (1, 's'), (2, 't'), (3, 'i'), (4, 'h'), (5, 'z'), (6, 'a')]
```

Bu fonksiyonun nasıl bir çıktı verdiğini öğrendiğimize göre, bu çıktıyı ihtiyaçlarımız ve isteklerimiz doğrultusunda evirip çevirebiliriz. Mesela "istihza" kelimesindeki harfleri düzgün bir şekilde numaralandıralım:

```
>>> for i in enumerate("istihza"):
...     print(i[0], i[1])
...
0 i
1 s
2 t
3 i
4 h
5 z
6 a
```

Hatta istersek sayı ve harflerin arasına birer işaret de yerleştirebiliriz:

```
>>> for i in enumerate("istihza"):
...     print(i[0], i[1], sep="==>")
...
0==>i
1==>s
2==>t
3==>i
4==>h
5==>z
6==>a
```

Hatta ve hatta artık `format()` gibi oldukça esnek bir metodu da biliyor olduğumuza göre daha karmaşık şeyler de yapabiliriz:

```
>>> for i in enumerate("istihza"):
...     print("{0} : '{1}'".format(i[0], i[1]))
```

```
...
0 : 'i'
1 : 's'
2 : 't'
3 : 'i'
4 : 'h'
5 : 'z'
6 : 'a'
```

enumerate() fonksiyonunun verdiği çıktının çift öğeli demetlerden oluştuğunu görmüştük. Bu bilgiyi kullanarak yapıyı biraz sadeleştirebiliriz:

```
>>> for k, v in enumerate("python"):
...     print("{0} : '{1}'".format(k, v))
...
0 : 'p'
1 : 'y'
2 : 't'
3 : 'h'
4 : 'o'
5 : 'n'
```

Burada "for k, v..." satırındaki "k" harfi `enumerate()` fonksiyonu tarafından üretilen iki öğeli demetin birinci öğesini, "v" harfi ise o demetin ikinci öğesini temsil ediyor. Yani "k" sayıyı, "v" ise harfi temsil ediyor... Daha sonra `print()` fonksiyonuyla öğeleri ekrana yazdırırken de bu harflerden yararlanıyoruz...

Daha önce veri tiplerini anlatırken sık sık kullandığımız bir kod parçasını temel alarak `enumerate()` ile ilgili bir örnek daha yapalım.

Hatırlarsanız `len()` fonksiyonunu anlatırken şöyle bir örnek vermiştik:

```
>>> liste = ["kitap", "defter", "kalem", "silgi", "tebeşir",
... "tahta", "sıra", "öğrenci", "okul", "önlük"]

>>> for sondarbe in range(len(liste)):
...     print(sondarbe+1, liste[sondarbe], sep=". ")
...
1. kitap
2. defter
3. kalem
4. silgi
5. tebeşir
6. tahta
7. sıra
8. öğrenci
9. okul
10.önlük
```

Artık `enumerate()` fonksiyonunu öğrendiğimize göre bu işlemi çok basit ve verimli bir şekilde halledebiliriz:

```
>>> liste = ["kitap", "defter", "kalem", "silgi", "tebeşir",
... "tahta", "sıra", "öğrenci", "okul", "önlük"]

>>> for k, v in enumerate(liste):
...     print(k, v, sep=". ")
...
0. kitap
1. defter
```

```
2. kalem
3. silgi
4. tebeşir
5. tahta
6. sıra
7. öğrenci
8. okul
9. önlük
```

Yalnız, gördüğünüz gibi burada bir eksiklik var. Listemizdeki sayılar 0'dan başlıyor. Halbuki biz saymaya 1'den başlanmasını istiyoruz. `enumerate()` fonksiyonunu kullanarak bunu yapmak çok basittir:

```
>>> for k, v in enumerate(liste, 1):
...     print(k, v, sep=". ")
...
1. kitap
2. defter
3. kalem
4. silgi
5. tebeşir
6. tahta
7. sıra
8. öğrenci
9. okul
10. önlük
```

Gördüğünüz gibi, `enumerate()` fonksiyonuna ikinci parametre olarak verdiğimiz sayı, ilk öğenin hangi numaraya sahip olacağını gösteriyor. Biz burada ilk öğenin "1" sayısına sahip olmasını istedik...

Son bir örnek daha verip bu konuyu kapatalım:

```
>>> for k, v in enumerate(dir(dict), 1):
...     print(k, v)
```

Bu kodlarla "dict" adlı veri tipinin bütün metotlarını numaralandırarak ekrana basıyoruz...

9.13 Kaçış Dizileri (Escape Sequences)

Hatırlarsanız ilk derslerimizin birinde şöyle bir örnek vermiştik:

```
>>> print("Ahmet, \"Bugün Adana'ya gidiyoruz,\" dedi.")
```

O derste, buradaki "\" işaretinin bir "kaçış dizisi" olduğundan söz etmiştik. Bu işaret, karakter dizisini başlatan çift tırnak ile, cümle içinde geçen çift tırnakların birbirine karışmasını engelliyordu. Aynı şekilde, yeni bir satıra geçmemizi sağlayan "\n" işaretinin de bir kaçış dizisi olduğunu söylemiştik. Python'da bol miktarda kaçış dizisi bulunur ve bunların her biri birbirinden farklı görevleri yerine getirir.

Python'da temel olarak iki adet kaçış dizisi vardır: "r" ve "\". Bu bölümde bu kaçış dizilerini ve bunların varyasyonlarını tek tek inceleyeceğiz:

"\"

Bu işaret hemen hemen bütün programlama dillerinin demirbaşıdır. Bunun düz bölü değil ters bölü olduğuna özellikle dikkat etmeliyiz...

Bu işaretin ne işe yaradığını daha önceki derslerimizde görmüştük. Yukarıya da aldığımız örneğe baktığımızda bu işaretin görevi açıkça ortaya çıkıyor. "\" işareti kendisinden sonra gelen herhangi bir karakterin özel bir anlamı olduğunu gösterir. Mesela şu örneğe bakalım:

```
>>> print("Kezban, \"Nen var kuzum,\" dedi.")
```

Burada "\" işareti, kendisinden sonra gelen çift tırnak karakterinin özel bir anlamı olduğunu belirtiyor. Yani bu çift tırnağın, karakter dizisini başlatan ve bitiren çift tırnaklardan farklı olduğunu gösteriyor. Aynı işareti öteki özel çift tırnağın da önüne yerleştirdiğimizde yukarıdaki karakter dizisini hatasız olarak ekrana yazdırabiliyoruz.

Mesela bu kaçış dizisi "n" harfinden önce gelirse, bu harfin de özel bir anlamı olduğunu belirtecektir. "\n" adlı kaçış dizisi yeni satıra geçilmesini sağlar:

```
>>> print("Bu birinci satır.\nBu da ikinci satır...")

Bu birinci satır.
Bu da ikinci satır...
```

Python'daki kaçış dizileri genel olarak "\" işaretinin başka harflerle birleşerek özel bir anlam oluşturmasıyla meydana getirilir. Şimdi bu "\" işaretinin yanına gelip özel bir anlam kazanan harfleri ve bu şekilde ortaya çıkan kaçış dizilerini inceleyelim...

"\t"

Eğer "\" işaretini "t" harfinden önce getirirsek, ortaya çıkan işaret ("\t"), karakter dizisi içinde bir sekme (tab) oluşturacaktır:

```
>>> print("Aramızda \tdağlar var...")

Aramızda        dağlar var...
```

Eğer daha büyük bir sekmeye ihtiyacınız varsa, "\t" kaçış dizisini birden fazla sayıda kullanabilirsiniz:

```
>>> print("Uzak... \t\tçok uzak...")

Uzak...                 çok uzak...
```

Elbette birden fazla farklı kaçış dizisini bir arada da kullanabiliriz:

```
>>> print("Aşağıda \n\thava nasıl?")

Aşağıda
        hava nasıl?
```

Gördüğünüz gibi "\t" adlı kaçış dizisi bizi boşluk tuşuna üst üste birkaç kez basmaktan kurtarıyor. Karakter dizimiz içinde sekmeye ihtiyacımız olduğunda bu kaçış dizisi epey işimize yarayacak.

Şimdi başka bir kaçış dizisine bakalım.

"\\"

Şimdiye kadar öğrendiğimiz gibi, "\", "\t" ve "\n" işaretlerini bir karakter dizisi içinde kullandığımızda her biri özel bir görevi yerine getiriyor. Peki ya biz yazdığımız bir karakter dizisi içinde bu işaretleri göstermek istersek ne yapacağız? Mesela "\n" ifadesinin kendisini bir karakter dizisi içine yazıp kullanıcılarımıza gösteremez miyiz?

Elbette gösterebiliriz. Eğer yazdığımız bir karakter dizisi içinde bu kaçış dizilerinin kendisini göstermek istersek şöyle bir yol tutuyoruz:

```
>>> print("Öğrendiğimiz karakter dizileri şunlardır: \\n, \\t")

Öğrendiğimiz karakter dizileri şunlardır: \n, \t
```

Gördüğünüz gibi, “\t”, “\n” veya “\” kaçış dizisinin kendisini ekrana basabilmek için “\” işaretini çiftleyerek “\\” şeklinde yazmamız gerekiyor...

Eğer sadece “\” işaretini göstermek istersek bazı noktalara dikkat etmemiz gerekiyor.

Mesela şu örneğe bakalım:

```
>>> print("demirbaş kaçış dizisi: \.")

demirbaş kaçış dizisi: \.
```

Gördüğünüz gibi, eğer “\” işaretini tek başına kullanacaksak özel bir şey yapmamıza gerek yok. Yalnız burada şuna dikkat etmemiz gerek: “\” işaretini çiftlemeden yazabilmemiz için, bu işaretin sağına herhangi bir özel karakterin gelmemesi lazım. Bu ne demek? Hemen bir örnek verelim:

```
>>> print("demirbaş kaçış dizisi: \")

SyntaxError: EOL while scanning string literal
```

Bu örnek Python’un bize bir hata mesajı göstermesine neden oluyor... Bunun sebebi aslında “\” işaretinin bu örnekte tek başına kullanılmıyor olması. Yani “\” işaretinden sonra karakter dizisinin bitiş tırnağı geliyor... Bildiğimiz gibi, “\” işareti çift veya tek tırnak ile birlikte kullanılırsa, Python bu tek veya çift tırnağı farklı bir şekilde algılayacaktır. Mesela bu örnekte, kapanış tırnağından hemen önce gelen “\” kaçış dizisi nedeniyle Python bu kapanış tırnağını sanki yokmuş gibi varsayacak ve bize hata mesajı gösterecektir... Bunun olmasını engellemek için, “\” işaretini çiftlememiz gerekiyor:

```
>>> print("demirbaş kaçış dizisi: \\")

demirbas kaçış dizisi: \
```

Bir de şu örneğe bakalım:

```
>>> print("Öğrendiğimiz karakter dizileri şunlardır: \, \\n, \\t")

Şimdiye kadar öğrendiğimiz karakter dizileri şunlardır: \, \n, \t
```

Burada “\” kaçış dizisini çiftlememize gerek kalmadığına dikkat edin. Çünkü bu işaretten hemen sonra gelen virgül işareti Python açısından özel anlam taşıyan bir karakter değildir. Bu yüzden yukarıdaki karakter dizisinin ekrana basılmasında herhangi bir sorunla karşılaşmıyoruz. Ama “\” işaretini yukarıdaki örnekte en sona yerleştiremeyeceğimize dikkat edin:

```
>>> print("Öğrendiğimiz karakter dizileri şunlardır: \\n, \\t, \")

SyntaxError: EOL while scanning string literal
```

Bu hatayı almamızın nedeni, “\” işaretini en sona koyduğumuzda bu işaretin, karakter dizisini kapatan tırnak işaretleriyle çakışmasıdır... Yukarıda verdiğimiz print(“demirbaş kaçış dizisi: \.”) örneğinde “\” işaretinden sonra bir “nokta” getirerek sorundan kaçtığımıza dikkat edin... Burada nokta koymak yerine boşluk karakteri de koyabilirdik:

```
>>> print("demirbaş kaçış dizisi: \ ")

demirbaş kaçış dizisi: \
```

Sözün özü, "\" işaretini tek başına kullandığımızda sağındaki karakterle çakışma riski varsa, çözüm olarak "\" işaretini çiftleme yoluna gidiyoruz.

"\v"

Eğer "\" işaretini "v" harfiyle birlikte kullanırsak "düşey sekme" denen şeyi elde ederiz. Hemen bir örnek verelim:

```
>>> print("düşey\vsekme")

düşey
     sekme
```

Yalnız bu "\v" adlı kaçış dizisi her işletim sisteminde çalışmayabilir... Dolayısıyla, birden fazla platform üzerinde çalışmak üzere tasarladığınız programlarınızda bu kaçış dizisini kullanmanızı önermem.

"\b"

"\" kaçış dizisinin, biraraya geldiğinde özel bir anlam kazandığı bir başka harf de "b"dir... "\b" kaçış dizisini karakter dizisi biçimlendirme bölümünde bir örnek içinde kullanmıştık. Örneğimiz şöyleydi:

```
>>> lider = input("Şampiyonun ismini giriniz: ")

>>> print("Bu senenin şampiyonu", lider, "\b.",  "Tebrikler.")
```

Burada "\b" adlı kaçış dizisi, kendisinden önceki bir karakteri silme görevini üstleniyor. Bu örnekte "lider" değişkeni ekrana yazdırılırken kendisinden sonra bir adet boşluk karakteri olacaktır çıktıda... İşte "\b" kaçış dizisi bu boşluk karakterini siliyor. "\b" kaçış dizisinin görevi klavyedeki "backspace" tuşuyla aynıdır...

"\b" kaçış dizisini bir örnekte daha görelim:

```
>>> print("merhaba\b")
```

Ne oldu? Bu karakter dizisi hiç bir şeyi değiştirmedi, değil mi? Çok normal. Çünkü bu kaçış dizisi biraz kaprislidir. "\b" adlı kaçış dizisi, çalışabilmek için kendisinden sonra da en az bir karakter olmasını ister. Dolayısıyla yukarıdaki örnekte "\b" kaçış dizisi görevini yerine getirmiyor, ama şu örnekte bu kaçış dizisi görevini eksiksiz olarak yerine getirecektir:

```
>>> print("merhaba\b dünya")

merhab dünya
```

Burada "\b" kaçış dizisinden sonra boşluk karakteri var. Bu yüzden bu kaçış dizisi böyle bir ortamda görevini yerine getirebiliyor. Eğer sadece "print("merhaba")" yazıp "merhab" çıktısını elde etmek isterseniz şöyle bir hileye başvurabilirsiniz:

```
>>> print("merhaba\b ")

merhab
```

Burada gördüğünüz gibi, "\b" kaçış dizisinden sonra bir boşluk bırakarak kandırmaca yoluna gittik...

Bu kaçış dizisini art arda yazarak tuhaf şeyler de yapabilirsiniz...

```
>>> print("merhaba\b\b ")

merha a
```

Burada "\b\b" kaçış dizilerinin yaptığı şey, sola doğru iki karakter atlayıp, ulaştığı noktaya bir adet boşluk karakteri yerleştirmektir. Şu örnekte durum biraz daha açık görünecektir:

```
>>> print("merhaba\b\bf")

merhafa
```

Yani bu karakter dizisini kullanarak şöyle saçma bir şey de yapabilirsiniz:

```
>>> print("merhaba\b\b\b\b\b\b\bgülegüle")

gülegüle
```

Bu kodları şöyle yazarak biraz beyin jimnastiği yapmak isteyebilirsiniz:

```
>>> print("merhaba{0}{1}".format("\b"*len('merhaba'), "gülegüle"))

gülegüle
```

Gördüğünüz gibi, bu karakter dizisi rahatlıkla suyu çıkarılabilecek bir araçtır...

"\r"

"\" işaretini birarada kullanacağımız başka bir harfimiz ise "r" harfidir...

"\r" kaçış dizisi, bir karakter dizisinin en başına gelmemizi sağlar. Mesela bu karakter dizisini kullanarak "tank" kelimesini "bank"a çevirebiliriz!

```
>>> print("tank\rb")

bank
```

"\a"

"\" kaçış dizisini son olarak "a" harfi ile birlikte kullanacağız. "\a" kaçış dizisi bilgisayarımızın "bip" sesi çıkarmasını sağlar:

```
>>> print("\a")

==bip!==
```

Tıpkı "\v" kaçış dizisi gibi, bu kaçış dizisi de her işletim sisteminde çalışmayabilir. Örneğin bu kaçış dizisinin GNU/Linux'ta çalışacağının garantisi yoktur...

"r"

Bu karakter dizisi kimi durumlarda oldukça faydalı olabilecek bir araçtır. Diyelim ki bilgisayarımızda tam yolu "C:\test\rehber.txt" olan bir dosya var. Bu dosyanın tam yolunu `print()` fonksiyonu yardımıyla ekrana yazdırmak istersek şöyle bir çıktı alırız:

```
>>> print("C:\test\rehber.txt")

ehber.txtst
```

Burada olan şey şu: Python, "C:\test\rehber.txt" içindeki "\t" ve "\r" ifadelerini birer kaçış dizisi olarak algıladı. Dolayısıyla "C:"den sonra sanki sekme tuşuna basılmış gibi davrandıktan sonra,

“\r” kaçış dizisi zannettiği şeyin etkisiyle de karakter dizisinin başına dönüp “ehber.txt” kısmını oraya yapıştırdı... Bu örnek, Python’daki kaçış dizilerini kullanmasak bile onları en azından tanımamız gerektiğini açık seçik gösteriyor. İşte bu tür sürprizleri engellemek için “r” adlı kaçış dizisinden yararlanacağız. Bunun, daha önce gördüğümüz “\r” kaçış dizisinden farklı olduğuna dikkat edin:

```
>>> print(r"C:\test\rehber.txt")

C:\test\rehber.txt
```

Gördüğünüz gibi, “r” kaçış dizisinin kullanım yeri öteki kaçış dizilerine göre farklılık gösteriyor. Bu kaçış dizisini karakter dizisinin içinde değil, dışında kullanıyoruz. Bu kaçış dizisinin görevi, karakter dizisi içinde geçen bütün kaçış dizilerini etkisiz hale getirmektir... Dolayısıyla daha önce verdiğimiz şu örneği:

```
>>> print("Öğrendiğimiz karakter dizileri şunlardır: \\n, \\t, \\")
```

...şöyle de yazabiliriz...

```
>>> print(r"Öğrendiğimiz karakter dizileri şunlardır: \n, \t")
```

Burada kullandığımız “r” kaçış dizisi, karakter dizisi içindeki bütün kaçış dizilerini etkisiz hale getirdiği için, bizim cümle içinde geçen karakter dizilerini çiftlememize gerek kalmadı.

Ancak “r” adlı karakter dizisi “\” kaçış dizisini etkisizleştiremez. Dolayısıyla şöyle bir yazım, hata almamıza yol açacaktır:

```
>>> print(r"Öğrendiğimiz karakter dizileri şunlardır: \n, \t, \\, \")
```

Bu durumdan kurtulmanın tek yolu, en sona bir boşluk karakteri yerleştirmektir:

```
>>> print(r"Öğrendiğimiz karakter dizileri şunlardır: \n, \t, \\, \ ")
```

Böylece Python’daki kaçış dizilerini gözden geçirmiş olduk. Gelin isterseniz kaçış dizilerine şöyle topluca bir göz atalım:

\n Yeni satır oluşturur.

\’ Tek tırnak içine aldığımız karakter dizilerinde tek tırnak kullanmamızı sağlar.

\” Çift tırnak içine aldığımız karakter dizilerinde çift tırnak kullanmamızı sağlar.

\a Sistemimizin “bip” sesi çıkarmasını sağlar (Her platformda çalışmaz).

\b Kendisinden önce gelen karakteri siler.

\r Karakter dizisinin en başına dönülmesini sağlar.

\t Karakter dizisi içinde sekme oluşturur.

\v Düşey sekme oluşturur (Her platformda çalışmaz).

**** Herhangi bir kaçış dizisini etkisiz hale getirir.

r Karakter dizisi içinde geçen bütün kaçış dizilerini etkisiz hale getirir.

BÖLÜM 10

Fonksiyonlar

Bir önceki bölümün tamamlanmasıyla birlikte Python'da "Temel Bilgiler" olarak adlandırabileceğimiz konuları bitirmiş olduk. Böylelikle artık Python programlama dilini "temel düzeyde" bildiğimizi rahatlıkla iddia edebiliriz...

Bu bölümden itibaren Python'un derinliklerine doğru yolculuğumuza başlayacağız. Bu yolculuktaki ilk durağımız "Python'da Fonksiyonlar" olacak... Peki nedir bu "fonksiyon" denen şey?

En basit tanımıyla fonksiyon, birbiriyle ilişkili deyimleri, kod parçalarını bir araya toplamamızı sağlayan bir "kod bloğu"dur. Python'daki fonksiyonlar bize işlerimizi otomatikleştirme imkanı sağlar. Fonksiyonların bu otomatikleştirme işini nasıl yerine getirdiğini bu bölümde inceleyeceğiz. Bu bölümü bitirip fonksiyonlar konusunu öğrendiğimizde, Python'da ileriye doğru çok önemli ve büyük bir adım atmış, deyim yerindeyse "level atlamış" olacağız...

Aslında fonksiyon bizim yabancısı olduğumuz bir kavram değil. Şimdiye kadar Python'da pek çok fonksiyon gördük ve kullandık. Örneğin, `print()` komutunun bir fonksiyon olduğunu ta en başta söylemiştik... Aynı şekilde, daha önceki derslerimizde gördüğümüz ve işlerimizi bir hayli kolaylaştırmış olan `range()`, `len()`, `round()`, `str()`, `int()`, `list()`, `dict()`, `tuple()` vb. de birer fonksiyondur. İşte şimdi biz de bu bölümde buna benzer fonksiyonları nasıl oluşturabileceğimizi öğreneceğiz... O halde isterseniz lafı daha fazla uzatmadan işe koyulalım...

10.1 Fonksiyon Tanımlamak

Giriş kısmında söylediğimiz gibi, fonksiyonlar işlerimizi otomatikleştirmemizi sağlar. Hatırlarsanız, bundan önceki derslerimizin birinde `sum()` adlı bir fonksiyondan söz etmiştik. Bu fonksiyon, mesela bir liste içindeki bütün sayıların toplamını veriyordu bize. Bu fonksiyonu şöyle kullanıyorduk:

```
>>> a = [2, 3, 4]
>>> sum(a)

9
```

Peki bu fonksiyon nasıl oluyor da işlerimizi otomatikleştirmemizi sağlıyor?

Bir an için Python'da fonksiyon diye bir şeyin olmadığını varsayalım. Yukarıdaki gibi, bir liste içindeki bütün sayıları toplamak istediğimizde şöyle bir şey yazmamız gerekecekti:

```
>>> lst = [2, 3, 4]
>>> a = 0

>>> for i in lst:
...     a = a + i
... print(a)
```

Başka bir grup sayıyı toplamak istediğimizde ise bu kodları en baştan tekrar yazmak zorunda kalacaktık:

```
>>> lst2 = [10, 20, 56, 55]
>>> a2 = 0

>>> for v in lst2:
...     a2 = a2 + v
... print(a2)
```

Eğer sum() fonksiyonu olmasaydı, program içinde ne zaman bir grup sayıyı toplamak istesek yukarıdaki kodları her defasında yeni baştan yazmanız gerekecekti... Ama Python'un bize sunduğu sum() fonksiyonu sayesinde bu işlemlerin hiçbirini yapmak zorunda kalmıyoruz. Programımız içinde bir grup sayıyı toplamak istediğimizde sadece sum() fonksiyonunu çağırmamız yeterli oluyor. Yani bu fonksiyon bir anlamda işlerimizi otomatik bir hale getirmemizi sağlıyor...

sum() oldukça faydalı bir fonksiyondur. Bu fonksiyon sayesinde, bir kere yazılmış bir kod parçasını farklı yerlerde ve programlarda tekrar tekrar kullanabiliyoruz. Zaten fonksiyonların ana görevlerinden biri de, bir kere yazılmış kodların tekrar tekrar kullanılabilmesini sağlamaktır. Yabancılar buna "code reusability" adını veriyor...

sum() fonksiyonu, bir grup sayıyı kolay bir şekilde toplamamızı sağlayan bir araçtır. Ancak Python'da tıpkı bu sum() fonksiyonuna benzer şekilde bir liste içindeki bütün sayıları çarpan, doğrudan kullanabileceğimiz özel bir fonksiyon bulunmaz. Ama biz istersek fonksiyonlar yardımıyla kendimiz için böyle bir fonksiyon yazabiliriz. Şimdi bunu nasıl yapacağımıza bakalım. Hem böylece fonksiyonlar konusuna da hızlı bir giriş yapmış olalım.

Python'da bir fonksiyonu kullanabilmek için öncelikle o fonksiyonu tanımlamamız gerekir. Bu tanımlama işini def adlı bir parçacık yardımıyla yapıyoruz. Python'da bir fonksiyon şöyle tanımlanır (Bu kodları dosyaya kaydedelim):

```
def fonksiyon_adı():
```

Madem biz çarpma yapan bir fonksiyon yazacağız, o halde fonksiyonumuzu şöyle tanımlayabiliriz:

```
def çarp():
```

Burada def parçacığı fonksiyonumuzu tanımlamamızı sağlıyor. "çarp()" ise fonksiyonumuzun adı... Bu arada "çarp" kelimesinin sonuna "()" işaretini ve ardından da iki nokta üst üsteyi koymayı asla unutmuyoruz. Şimdi yapmamız gereken şey fonksiyonumuzun içeriğini yazmak olacak:

```
def çarp():
    a = 1
    for i in [2, 3, 4]:
        a = a * i
    print(a)
```

Burada girintilere özellikle dikkat ediyoruz... Bu kodlarda yaptığımız şey, "[2, 3, 4]" listesinin

tek tek bütün öğelerini “1” sayısıyla çarpıp, sonucu a değişkenine atamaktan ibaret... Bu işlem nihai olarak listedeki bütün sayıların birbiriyle çarpılmasını sağlayacaktır.

Böylece fonksiyonumuzu tanımlamış olduk. Şu anda yapmamız gereken tek bir şey kaldı. O da yazdığımız fonksiyonumuzu çağırmak:

```
çarp()
```

Bunu da kodlarımız arasına ekleyelim:

```
def çarp():
    a = 1
    for i in [2, 3, 4]:
        a = a * 1
    print(a)

çarp()
```

Yine girintilere çok dikkat ediyoruz... Ayrıca tanımladığımız fonksiyonun sınırlarına da dikkat edin. Bir fonksiyon, `def` parçacığıyla başlar, kendisinden sonra gelen girintili kod bloğuyla devam eder. Bu kod bloğunun dışında kalan her şey fonksiyonun da dışındadır. Mesela burada yazdığımız “çarp()” satırı fonksiyonun bir parçası değildir. Tanımladığımız fonksiyon sadece koyu harflerle gösterilen kodlardan ibarettir. “çarp()” satırının görevi ise tanımladığımız fonksiyonu çağırmaktır. Bu demek oluyor ki, Python’da bir fonksiyonu kullanabilmek için önce o fonksiyonu tanımlamamız, ardından da bu fonksiyonu çağırmamız gerekiyor. Yukarıdaki örnekte, koyu harflerle gösterilen kısım fonksiyonun tanımlandığı kısımdır... Bundan sonra gelen “çarp()” ifadesi ise fonksiyonumuzu çağırdığımız kısım oluyor... Bu önemli bilgiyi de öğrendiğimize göre yolumuza devam edebiliriz...

Bu arada, bu bölümün en başında örnek olarak verdiğimiz toplama işlemi yapan kodları da bir fonksiyon haline getirelim isterseniz:

```
def topla():
    a = 0
    for i in [2, 3, 4]:
        a = a + i
    print(a)
```

Son olarak fonksiyonumuzu çağırıyoruz:

```
topla()
```

Bunu da kodlarımızın arasına yerleştirelim:

```
def topla():
    a = 0
    for i in [2, 3, 4]:
        a = a + 1
    print(a)

topla()
```

Yukarıda tanımladığımız `çarp()` ve `topla()` adlı fonksiyonları bir dosyaya kaydedip çalıştırdığımızda Python bize ilgili işlemlerin sonucunu verecektir. Yani `çarp()` fonksiyonu “2”, “3” ve “4” sayılarının çarpımını, `topla()` fonksiyonu ise “2”, “3” ve “4” sayılarının toplamını gösterecektir. Burada fonksiyonların işimizi nasıl da kolaylaştırdığına dikkat edin. Yukarıda tanımladığımız `çarp()` fonksiyonu sayesinde, bir grup sayıyı çarpmak istediğimiz zaman, bütün kodları her defasında yeni baştan yazmak yerine sadece `çarp()` fonksiyonunu çağırmamız yeterli olacaktır...

Yalnız dikkat ederseniz, burada yazdığımız fonksiyonlar kısıtlı. Bu fonksiyonlar sadece “2”, “3” ve “4” sayıları üzerinde işlem yapabiliyor. Bunun nedeni bizim bu üç sayıyı fonksiyon içinde doğrudan tanımlamış olmamız... Eğer fonksiyonumuzu daha esnek bir hale getirmek istersek, örneğin `çarp()` fonksiyonunu şu şekilde yazmamız gerekir:

```
def çarp(liste):
    a = 1
    for i in liste:
        a = i * a
    print(a)
```

Burada, bir önceki yazımdan farklı olarak herhangi bir sayı belirtmedik. Onun yerine “liste” adlı bir değişken oluşturup bunu fonksiyona parametre olarak verdik... Bu sayede fonksiyonumuz herhangi bir sayı listesinin içeriğini çarpabilecek. Bir deneme yapalım:

```
sayılar = [24, 56, 77, 87]

çarp(sayılar)
```

Yazdığımız kodlar tam olarak şöyle görünecek:

```
def çarp(liste):
    a = 1
    for i in liste:
        a = i * a
    print(a)

sayılar = [24, 56, 77, 87]

çarp(sayılar)
```

Bu kodları bir dosyaya kaydedip çalıştırdığımızda şu sonucu alırız:

```
9003456
```

Gördüğünüz gibi fonksiyonlar, yazdığımız kodları bir araya toplamamızı, gruplandırmamızı da sağlamanın yanısıra, yapacağımız işlemler için bir şablon oluşturma vazifesi de görüyor. Pek çok kez kullanmamız gereken işlevleri bir şablon haline getirerek, bu şablon üzerinden, benzer işlemleri tek bir komutla halledebiliyoruz. İlerleyen derslerimizde “modüller” konusunu öğrendiğimizde bu fonksiyonları çok daha verimli bir biçimde kullanabileceğiz.

Fonksiyonlar, yazdığımız programlarda işlerimizi bir hayli kolaylaştırır. Fonksiyonlar olmasaydı bir program içinde defalarca yazmak zorunda kalacağımız kodları bir kez fonksiyon olarak tanımladıktan sonra program içinde tekrar tekrar kullanabiliriz. Eğer kodlarda bir değişiklik yapmamız gerekirse sadece fonksiyon içinde değişiklik yapmamız yeterli olacaktır. Öbür türlü, program içinde oraya buraya dağılmış kod parçalarını tek tek bulup düzeltmemiz gerekecekti... Özellikle arayüz tasarlarken fonksiyonlardan bolca yararlanacağız...

Fonksiyonlarla ilgili çok basit bir örnek verelim. Diyelim ki bir program yazdık ve programın bir yerinde kullanıcıdan sayı girmesini istiyoruz. Bunun, programımızın hata vermesine yol açma potansiyeli taşıyan bir işlem olduğunu varsayalım. Programın çalışması sırasında bir hatayla karşılaşıldığında kullanıcıya genel bir hata mesajı göstermek istersek şöyle bir kod yazabiliriz:

```
try:
    soru = input("gerekli sayıyı giriniz: ")

except:
    print("Beklenmeyen bir hata oluştu.")
    print("Bu hatayı, hata takip sistemine raporlayınız!..")
```

Yine diyelim ki “except” bloğu içinde yazdığımız genel hata mesajını programımızın başka yerlerinde de aynen kullanmak istiyoruz. Yani mesela şöyle bir şey yapmak istiyoruz:

```
try:
    soru = int(input("gerekli sayıyı giriniz: "))

except:
    print("Beklenmeyen bir hata oluştu.")
    print("Bu hatayı, hata takip sistemine raporlayınız!..")

#Burada programımıza ait başka kodlar yer alıyor...

try:
    soru2 = int(input("başka bir sayı giriniz: "))

except:
    print("Beklenmeyen bir hata oluştu.")
    print("Bu hatayı, hata takip sistemine raporlayınız!..")
```

Burada dikkat ederseniz, genel hata mesajını göstermek istediğimiz her yerde bu mesajları tekrar tekrar yazmamız gerekiyor. İşte fonksiyonlar böyle durumlarda işlerimizi kolaylaştırabilir... Yukarıdaki kodları fonksiyonlar yardımıyla şu şekilde sadeleştirebiliriz:

```
def genelHata():
    print("Beklenmeyen bir hata oluştu.")
    print("Bu hatayı, hata takip sistemine raporlayınız!..")

try:
    soru = int(input("gerekli sayıyı giriniz: "))

except:
    genelHata()

#Burada programımıza ait başka kodlar yer alıyor...

try:
    soru2 = int(input("başka bir sayı giriniz: "))

except:
    genelHata()
```

Bu verdiğimiz çok basit bir örnektir, ama bu basit örnekte bile, fonksiyonların işimizi ne kadar kolaylaştırabileceğini görebiliyoruz. Burada kullandığımız fonksiyon öncelikle bizi aynı şeyleri tekrar tekrar yazma zahmetinden kurtarıyor. `genelHata()` adlı fonksiyonu bir kez tanımladıktan sonra, gereken yerlerde sadece bu fonksiyonu çağırarak işimizi halledebiliyoruz. Ayrıca eğer kullanıcıya göstermek istediğimiz hata mesajında değişiklik yapmak istersek, sadece fonksiyon içinde gerekli değişiklikleri yapmamız yeterli olacaktır. Eğer burada fonksiyon kullanmasaydık, hata mesajlarını kodlar içinde tek tek bulup değiştirmemiz gerekecekti...

Bu bölümde gördüğümüz örneklerden de bildiğimiz gibi, yazdığımız fonksiyonlara parametre atayarak bunları daha da esnek bir hale getirebiliyoruz:

```
def genelHata(eposta):
    print("Beklenmeyen bir hata oluştu.")
    print("Bu hatayı, {0} adresine raporlayınız!".format(eposta))

try:
    soru = int(input("gerekli sayıyı giriniz: "))
```

```
except:
    genelHata("abc123@abc.com")

try:
    soru2 = int(input("başka bir sayı giriniz: "))

except:
    genelHata("123fds@cbc.com")
```

Bu programın çalışması esnasında eğer ilk "try... except..." bloğu içinde bir hata meydana gelirse kullanıcıya "Bu hatayı abc123@abc.com adresine raporlayınız!" mesajı; eğer hata ikinci "try... except..." bloğu içinde meydana gelirse de "Bu hatayı 123fds@cbc.com adresine raporlayınız!" mesajı gösterilecektir.

Bu arada dikkat ederseniz, `genelHata()` adlı fonksiyondaki "eposta" parametresi aslında bir "yer tutucu" olmaktan öte bir anlam ifade etmiyor. Yani oraya istediğiniz bir kelimeyi yazabilirsiniz. Amacımız oraya bir parametre geleceğini belirtmek. Tabii ki parametrelerimizi adlandırırken anlamlı isimler yazmamız her zaman için işimizi kolaylaştıracaktır... Bu "parametre" meselesini bir sonraki bölümde biraz daha ayrıntılı olacak inceleyeceğiz.

Bu bölümün son sözü olarak fonksiyonların adlandırılması konusunda birkaç şey söyleyelim... Fonksiyon adlarını küçük harflerle yazmak adettendir. Yani:

```
def fonk():
```

Burada görüldüğü gibi, fonksiyon adlarını yazarken küçük harfleri kullanıyoruz. Eğer yazacağımız fonksiyonun adı birden fazla kelimeden oluşuyorsa, fonksiyon adını oluşturan kelimeleri birbirinden "_" işareti ile ayırıyoruz. Yani:

```
def fonksiyon_adı():
```

Daha nadir olarak, birden fazla kelimeden oluşan fonksiyon adlarının şu şekilde yazıldığını da görebiliriz:

```
def fonksiyonAdı():
```

Son olarak, fonksiyon adlarının büyük harflerle başladığı pek görülmez...

10.2 Fonksiyonların Parametreleri

Bir önceki bölümde fonksiyonları parametrelerle birlikte kullanarak epey esnek sonuçlar elde edebileceğimizi görmüştük. Bu arada, bizim burada "parametre" olarak adlandırdığımız şey bazı yerlerde "argüman" olarak da karşınıza çıkabilir. Bu iki terim özünde birbirinden farklı olsa da, çoğunlukla fonksiyonlar için birbiriyle eş anlamlı olarak kullanılmaktadır... Yani fonksiyonlar söz konusu olduğunda "parametre" ve "argüman" aynı şeyi ifade eder. Peki "argüman" ile "parametre" arasındaki fark nedir?

Şu örnekte "x" bir parametredir:

```
def fonk(x):
```

Şu örnekte ise "50" bir argümandır:

```
fonk(50)
```

Yani bir fonksiyonun tanımlanması sırasında, parantez içinde belirtilen değişkenlere "parametre" adı verilirken, tanımlanan fonksiyonun çağrılması esnasında, bu değişkenlere verilen

değerlere ise “argüman” adı veriliyor. Dolayısıyla yukarıdaki fonksiyon hakkında şöyle bir cümle kurabiliriz:

“fonk” adlı fonksiyonun x parametresine argüman olarak 50 değerini verdik...

Ancak “parametre” ile “argüman” arasındaki bu farklılık çoğu zaman gözardı edilir ve bu iki kavram çoğunlukla birbiriyle eş anlamlı olarak kullanılır.

Dediğimiz gibi, bir önceki bölümde parametreleri temel olarak nasıl kullanabileceğimizi öğrenmiştik. Bu bölümde ise “parametre” kavramına biraz daha ayrıntılı olarak bakmaya çalışacağız.

Önce şu çok basit fonksiyona bir bakalım:

```
def bas(kelime):
    print("merhaba", kelime)
```

Burada `bas()` adlı bir fonksiyon tanımladık. Bu fonksiyon tek bir parametre alıyor. Fonksiyonumuzda bu parametreyi bir karakter dizisi içinde kullanıyoruz. Bu fonksiyonu çalıştırdığımızda, `bas()` fonksiyonuna argüman olarak verdiğimiz kelime “merhaba” karakter dizisi ile birlikte ekrana basılacaktır. Mesela bu fonksiyonu “istihza” argümanıyla çağıralım:

```
bas("istihza")
```

Kodlarımız şöyle görünecek:

```
def bas(kelime):
    print("merhaba", kelime)

bas("istihza")
```

Bu kodları bir dosyaya kaydedip çalıştırdığımızda şöyle bir çıktı elde ederiz:

```
merhaba istihza
```

Gördüğünüz gibi, `bas()` fonksiyonuna verdiğimiz “istihza” argümanı “merhaba” karakter dizisi ile birlikte ekrana basılıyor. Kodlarımızı biraz geliştirelim:

```
def bas(kelime):
    print("merhaba", kelime)

soru = input("adınız nedir? ")
bas(soru)
```

Bu kodlar, “merhaba” karakter dizisi ile birlikte ekrana dökülecek kelimeyi doğrudan kullanıcıdan alıyor... Bu arada, yazdığımız fonksiyonun nerede başlayıp nerede bittiğine dikkat edelim. “*soru = input(....*” satırı ve ondan sonraki kısım fonksiyonumuzun dışında yer alıyor.

Dediğimiz gibi, `bas()` fonksiyonu tek bir argüman alıyor. Dolayısıyla bu fonksiyonu argümansız olarak çağıramayız:

```
def bas(kelime):
    print("merhaba", kelime)

soru = input("adınız nedir? ")
bas()
```

Eğer `bas()` fonksiyonunu bu şekilde argümansız olarak yazıp çalıştırırsak aşağıdakine benzer bir hata alırız:

```
TypeError: bas() takes exactly 1 positional argument (0 given)
```

Bu hata mesajı şöyle çevrilebilir: “TipHatası: bas() tamı tamına 1 adet sıralı argüman alır (0 verilmiş)”

Demek ki `bas()` fonksiyonunu çalıştırabilmek için “1” adet argüman girmemiz gerekiyormuş... Ne bir eksik, ne bir fazla... Bu arada hata mesajı içinde geçen “sıralı argüman” (positional argument) kavramından da biraz sonra bahsedeceğiz.

Fonksiyonlara istediğimiz sayıda parametre verebiliriz. Ancak parametrelerimizi “makul” sayıda tutmak, fonksiyonun hangi parametrelere ihtiyaç duyduğunu hatırlayabilmek açısından önemlidir:

```
def deneme(bir, iki, liste, ölçüt):
    sonuç = bir * iki
    if bir and iki != ölçüt:
        liste.append(sonuç)
    else:
        print("{0} ile çarpmaya izin vermiyoruz!".format(ölçüt))
    print(liste)
```

Gördüğünüz gibi, fonksiyonlara birden fazla parametre verebiliyoruz... Burada tanımladığımız fonksiyon, “bir”, “iki”, “liste” ve “ölçüt” olmak üzere dört adet parametre alıyor. Yani bu fonksiyonu kullanabilmek için fonksiyonumuzu dört adet argüman ile çağırmamız gerekiyor. Fonksiyonumuzun içeriğine baktığımız zaman ilk olarak şu satırı görüyoruz:

```
sonuç = bir * iki
```

Buna göre, `deneme()` adlı fonksiyondaki “bir” ve “iki” adlı parametreler birbiriyle çarpılıp “sonuç” adlı değişkeni oluşturacak. Ardından gelen satırda bir “if... else...” bloğu görüyoruz. Eğer “bir” ve “iki” parametrelerinin değeri “ölçüt” parametresinin değerine eşit değilse, bir önceki satırda belirlediğimiz “sonuç” adlı değişkenin değerini “liste” parametresine ekliyoruz. Burada and adlı bool değerinin anlamına dikkat edin. “sonuç” adlı değişkenin listeye eklenebilmesi için hem “bir” değerinin, hem de “iki” değerinin “ölçüt”e eş olmaması gerekiyor.

Daha sonra `else` bloğunu tanımlıyoruz. Buna göre eğer “bir” ve “iki” parametrelerinden herhangi birinin değeri “ölçüt”e eşitse, kullanıcıya “ölçüt ile çarpmaya izin vermiyoruz!” şeklinde bir uyarı gösteriyoruz. Son satırda ise “liste” adlı parametrenin değerini ekrana basıyoruz...

Şimdi sıra geldi fonksiyonumuzu çağırmaya:

```
öğeler = []
deneme(23, 5, öğeler, 4)
```

Kodlarımızı topluca görelim:

```
def deneme(bir, iki, liste, ölçüt):
    sonuç = bir * iki
    if bir and iki != ölçüt:
        liste.append(sonuç)
    else:
        print("{0} ile çarpmaya izin vermiyoruz!".format(ölçüt))
    print(liste)

öğeler = []
deneme(23, 5, öğeler, 4)
```

Burada öncelikle “öğeler” adlı boş bir liste oluşturduk. Çünkü `deneme()` fonksiyonunun parametrelerinden birisi liste olmak zorunda... Ardından da `deneme()` fonksiyonunu çağırıyoruz. Fonksiyonumuzu dört adet argüman ile çağırdığımıza dikkat edin. Eğer argüman sayısı dörtten az veya fazla olursa programımız hata verecektir.

Bu kodları çalıştırdığımızda ekrana, fonksiyon içinde belirttiğimiz listenin öğeleri basılacaktır. Fonksiyonumuzu bir de şu argümanlarla çağıralım:

```
def deneme(bir, iki, liste, ölçüt):
    sonuç = bir * iki
    if bir and iki != ölçüt:
        liste.append(sonuç)
    else:
        print("{0} ile çarpmaya izin vermiyoruz!".format(ölçüt))
    print(liste)

öğeler = []
deneme(23, 5, öğeler, 5)
```

Bu defa şöyle bir çıktı alıyoruz:

```
5 ile çarpmaya izin vermiyoruz!
[]
```

Bu çıktıyı almamızının sebebi "iki" adlı parametrenin değerinin (5), "ölçüt" parametresinin değeriyle aynı olması (5). Fonksiyonumuz içinde böyle bir durum için `else` bloğunu yazmış ve böyle bir duruma izin vermediğimiz konusunda kullanıcıyı uyarmıştık...

10.3 Varsayılan Değerli Argümanlar

Bu bölümde "Varsayılan Değerli Argümanlar" diye bir şeyden söz edeceğiz. Bakalım neymiş bu "Varsayılan Değerli Argümanlar"...

Önceki derslerimizden `range()` fonksiyonunu biliyorsunuz. `range()` fonksiyonunu anlatırken bu fonksiyonun bazı varsayılan değerleri olduğunu söylemiştik. Yani bu `range()` fonksiyonu aslında üç argüman almasına rağmen biz bunu tek bir argüman ile de kullanabiliyorduk...

`range()` fonksiyonu şu parametrelerden oluşuyordu:

```
range(başlangıç_değeri, bitiş_değeri, atlama_değeri)
```

`range()` fonksiyonunu kullanabilmek için bu parametreler içinde sadece "bitiş_değeri" adlı parametreye bir argüman vermemiz yeterli olacaktır:

```
list(range(10))
```

Öteki iki parametrenin birer varsayılan değeri olduğu için o parametreleri belirtmesek de olur. Yine `range()` fonksiyonunu anlattığımız bölümde o parametrelerin varsayılan değerlerinin şöyle olduğunu söylemiştik:

```
başlangıç_değeri = 0
atlama_değeri = 1
```

Dolayısıyla, biz `range(10)` yazdığımızda Python bunu "range(0, 10, 1)" şeklinde algılayacak ve ona göre işlem yapacaktır.

Peki varsayılan değerli argümanların bize ne gibi bir faydası var? Varsayılan değerli argümanları şuna benzetebiliriz: Diyelim ki bilgisayarınıza bir program kuruyorsunuz. Eğer bu programı ilk defa kuruyorsanız, kurulumla ilgili bütün seçeneklerin ne işe yaradığını bilmiyor olabilirsiniz... Dolayısıyla eğer program size kurulumun her aşamasında bir soru sorarsa her soruya ne cevap vereceğinizi kestiremeyebilirsiniz. O yüzden, makul bir program, kullanıcının en az seçimle programı kullanılabilir hale getirmesine müsaade etmelidir. Yani bir programın

kurulumu esnasında bazı seçeneklere programı yazan kişi tarafından bazı mantıklı varsayılan değerler atanabilir. Örneğin kullanıcı programı kurarken herhangi bir kurulum dizini belirtmezse program otomatik olarak varsayılan bir dizine kurulabilir... Veya programla ilgili, kullanıcının işine yarayacak bir özellik varsayılan olarak açık gelebilir. İşte biz de yazdığımız programlarda kullanacağımız fonksiyonlara bu şekilde varsayılan değerler atarsak programımızı kullanacak kişilere kullanım kolaylığı sağlamış oluruz... Örneğin `range()` fonksiyonunu yazan Python geliştiricileri bu fonksiyona bazı varsayılan değerler atayarak bizim için bu fonksiyonun kullanımını kolaylaştırmışlardır. Bu sayede `range()` fonksiyonunu her defasında üç argüman vermek yerine tek argüman ile çalıştırabiliyoruz...

Biz de kendi yazdığımız fonksiyonlarda böyle varsayılan değerler belirleyebiliriz. Mesela şu örneğe bir bakalım:

```
def böl(bir, iki, kip=True):
    sonuç = bir / iki
    if kip == True:
        print(sonuç)
    if kip == False:
        print(int(sonuç))
```

Bu fonksiyonu şu argümanlarla çağıralım:

```
böl(10, 2)
```

Kodlarımız tam olarak şöyle görünecek:

```
def böl(bir, iki, kip=True):
    sonuç = bir / iki
    if kip == True:
        print(sonuç)
    if kip == False:
        print(int(sonuç))

böl(10, 2)
```

Bu programı çalıştırdığımızda şu sonucu alırız:

```
5.0
```

Gördüğünüz gibi, sonuç içinde ondalık kısım da görünüyor... Eğer tamsayı şeklinde bir sonuç elde etmek istersek fonksiyonumuzu şöyle çağıracağız:

```
böl(10, 2, False)
```

Bu şekilde şöyle bir sonuç alırız:

```
5
```

Fonksiyonumuza baktığımız zaman, `böl()` fonksiyonunun üç adet parametre aldığını görüyoruz. Ama sonuncu parametreye varsayılan bir değer atadığımız için, fonksiyonumuzu kullanırken sadece iki argüman vermemiz yeterli olacaktır. Eğer üçüncü parametreyi belirtmezsek, bu parametrenin değeri “True” varsayılacaktır...

Bu arada, yukarıdaki fonksiyonu şu şekilde tanımlayabileceğimizi de biliyoruz:

```
def böl(bir, iki, kip=True):
    sonuç = bir / iki
    if kip:
        print(sonuç)
```

```
    if not kip:
        print(int(sonuç))
```

Burada "*if kip:*" ifadesi "*if kip == True:*" ile eşanlamlıdır. "*f not kip:*" ise "*if kip == False*" ile aynı anlama gelir...

Hatta istersek fonksiyonumuz için şöyle bir yazım tarzı dahi belirleyebiliriz:

```
def böl(bir, iki, kip=True):
    if kip == True:
        print(bir / iki)
    if kip == False:
        print(bir // iki)
```

"//" ve "/" işleçlerinin farkını daha önceki derslerimizde görmüştük. "//" işlecini kullanarak bölme işlemi yaptığımızda elde ettiğimiz sonuç bir tamsayı oluyor. Yani bölme işleminin sonucu ondalık kısmı içermeyecek şekilde veriliyor...

Varsayılan değerli fonksiyonlar tanımlarken dikkat etmemiz gereken önemli bir kural var. Bu tür fonksiyonlarda varsayılan değerin, parametre sıralamasında en sonda gelmesi gerekiyor. Yani şöyle bir şey yazamayız:

```
def böl(kip = True, bir, iki):
```

Eğer varsayılan değerli argümanı en sona değil de başa veya ortaya alırsak Python bize şöyle bir hata verir:

```
SyntaxError: non-default argument follows default argument
```

Yani:

```
SözdizimiHatası: varsayılan değersiz argüman, varsayılan değerli argümandan sonra geliyor
```

Dolayısıyla kural olarak, varsayılan değere sahip argümanları parametre listesinin en sonuna yerleştirmeye özen göstermemiz gerekiyor...

10.4 Sıralı Argümanlar

Önceki bölümlerden birinde, parametre gerektiren bir fonksiyonu parametresiz olarak çağırdığımızda şöyle bir hata almıştık:

```
TypeError: bas() takes exactly 1 positional argument (0 given)
```

İşte bu dersimizde bu "sıralı argüman" (positional argument) ifadesinin neye karşılık geldiğini öğreneceğiz.

Esasında şimdiye kadar verdiğiniz fonksiyon örneklerinde çoğunlukla sıralı argümanlardan yararlandık. Sıralı argümanlar, bir fonksiyon içinde, bulunduğu konuma, yani sırasına göre belirtilen argümanlardır. Bu ne demek oluyor? Daha önce yaptığımız şu örneğe bakalım:

```
def deneme(bir, iki, liste, ölçüt):
    sonuç = bir * iki

    if bir and iki != ölçüt:
        liste.append(sonuç)
    else:
        print("{0} ile çarpmaya izin vermiyoruz!".format(ölçüt))
```

```
    print(liste)

öğeler = []
deneme(23, 5, öğeler, 4)
```

Burada deneme() fonksiyonunu çağırırken "bir", "iki", "liste" ve "ölçüt" parametrelerinin hangi sırayla dizildiğine dikkat etmeliyiz. Yani "iki" parametresine yazmamız gereken şeyi "liste" parametresinin yerine yazamayız. Bu tür parametrelere sıralı parametreler veya sıralı argümanlar adı verilir. Dolayısıyla, adlarından da anlaşılacağı gibi, bu argümanların fonksiyon içindeki sırası önemlidir. Mesela range() fonksiyonunu düşünelim:

```
for i in range(1, 10, 2):
    print(i)
```

Burada range() fonksiyonuna verdiğimiz "1" parametresi saymaya kaçtan başlayacağımızı, "10" parametresi ise saymayı kaçta bitireceğimizi gösterir. En sonda verdiğimiz "2" parametresi ise kaçar kaçar sayacağımızı gösterir. Bu üç parametrenin, fonksiyon içinde bulunduğu sıra önemlidir. Yani bu sayıların sırasını değiştirirsek farklı sonuçlar elde ederiz...

Sıralı argümanların çok fazla bir özelliği yoktur. Burada dikkat etmemiz gereken tek şey, parametrelerin konumunu ve sırasını karıştırmamaktır. Çünkü farklı sıralandırmalar farklı sonuçlar ortaya çıkarabilir. Bu basit konuyu da gözden geçirdiğimize göre ilerlemeye devam edebiliriz.

10.5 İsimli Argümanlar

Bir önceki bölümde sıralı argümanları görmüştük. Karşımıza çıkacak fonksiyonlar ve bizim kendi yazacaklarımız genellikle sıralı argümanlara sahip fonksiyonlar olacaktır... Ancak bazı durumlarda sıralı argümanlar bize birtakım zorluklar çıkarabilir. Özellikle bir fonksiyonun çok sayıda sıralı argümana sahip olduğu durumlarda parametrelerin neler olduğunu, bunların sayısını ve sırasını hatırlamak hiç de kolay olmayabilir. İşte bu tür durumlarda, fonksiyonların argümanlarını isimleriyle çağırma yoluna gidebiliriz... Hemen bir örnek verelim:

```
def deneme(bir, iki, liste, ölçüt):
    sonuç = bir * iki

    if bir and iki != ölçüt:
        liste.append(sonuç)
    else:
        print("{0} ile çarpmaya izin vermiyoruz!".format(ölçüt))
    print(liste)

öğeler = []

deneme(ölçüt=23, bir=5, liste=öğeler, iki=4)
```

Gördüğünüz gibi, argümanları isimleriyle çağırarak, yani isimli argümanları kullanarak kendimizi argümanların sırasını ezberleme zahmetinden kurtarmış oluyoruz...

Burada da kullandığımız isimli argümanları isimsiz argümanlardan sonra getirmeye dikkat ediyoruz. Aksi halde Python bize bir hata mesajı gösterecektir. Dolayısıyla yukarıdaki fonksiyonu aşağıdaki şekilde çağırmamız mümkün değildir:

```
deneme(ölçüt=23, bir=5, liste=öğeler, 4)
```

Buradaki sorun, isimli argümanlar olan ölçüt, bir ve listenin; isimsiz bir argüman olan 4'ten önce gelmesidir... Böyle bir durumda programımız hata verecektir.

10.6 Rastgele Sayıda İsimsiz Argüman Verme

Hatırlarsanız birkaç bölüm önce şöyle bir fonksiyon tanımlamıştık:

```
def çarp(liste):
    a = 1
    for i in liste:
        a = i * a
    print(a)
```

Bu fonksiyon, liste halinde verilmiş sayıları birbiriyle çarpma vazifesi görüyordu. Biz bu fonksiyonu, Python'daki `sum()` adlı fonksiyona özenerek yazmıştık. Hatırlarsanız bu `sum()` fonksiyonu kendisine verilen liste halindeki sayıları birbiriyle toplayabiliyordu. Bizim tanımladığımız `çarp()` fonksiyonu da buna benzer şekilde, kendisine verilen liste halindeki sayıları birbiriyle çarpabiliyor... Ancak `çarp()` ve `sum()` adlı fonksiyonların önemli bir eksikliği var. Bildiğimiz gibi, `sum()` fonksiyonunu şu şekilde kullanıyoruz:

```
li = [2, 3, 4]
sum(li)
```

Aynı şekilde, yukarıda tanımladığımız `çarp()` fonksiyonunu da şöyle kullanıyoruz:

```
li = [2, 3, 4]
çarp(li)
```

Ancak biz bu iki fonksiyonu mesela şöyle kullanamıyoruz:

```
sum(2, 3, 4)
```

...veya...

```
çarp(2, 3, 4)
```

Bu fonksiyonları yukarıdaki şekilde çağırmayı denediğimizde Python bize hata mesajı gösterecektir.

`çarp()` fonksiyonu şöyle bir hata verir:

```
TypeError: çarp() takes exactly 1 positional argument (3 given)
```

Yani:

```
TipHatası: çarp() tamı tamına 1 adet sıralı argüman alır (3 adet verilmiş)
```

sum() fonksiyonu ise şu hatayı verir:

```
TypeError: sum expected at most 2 arguments, got 3
```

Yani:

```
TipHatası: sum en fazla 2 argüman almayı beklerken 3 argüman almış
```

Gördüğünüz gibi, ne `sum()` ne de `çarp()` fonksiyonu kendilerine verilen 2, 3 ve 4 sayıları üzerinde işlem yapabiliyor. Biz burada `sum()` fonksiyonuna müdahale edemeyiz, ama istersek `çarp()` fonksiyonuna böyle bir yetenek kazandırabiliriz.

Öncelikle şu basit fonksiyonu inceleyelim:

```
def bas(isim):
    print("merhaba", isim)
```

Eğer tanımladığımız bu fonksiyonu “Fırat” parametresiyle çalıştırırsak şöyle bir sonuç alırız:

```
bas("Fırat")

merhaba Fırat
```

Bu fonksiyonu birden fazla argüman ile çalıştıramayız. Çünkü fonksiyonu tanımlarken yalnızca tek bir parametre belirledik (isim). Eğer birden fazla argümanla çalıştırmak istersek, istediğimiz argüman sayısı kadar parametreyi fonksiyon tanımı içine yerleştirmeliyiz:

```
def seksenler(bir, iki, üç):
    print("bir {0}, bir {1}, bir de {2} alacağım!".format(bir, iki, üç))

seksenler("kalem", "pergel", "çikolata")
```

Peki ya parametre sayısını belirtmeden, istediğimiz kadar argümana izin vermek istersek ne olacak? Şu örneği daha önce de vermiştik:

```
def çarp(sayılar):
    a = 1
    for i in sayılar:
        a = a * i
    print(a)

li = [2, 3, 4]
çarp(li)
```

Gördüğünüz gibi, birden fazla sayıyı birbiriyle çarpabilmek için bunları önce bir liste haline getirmemiz gerekiyor (demet haline de getirebiliriz...). “çarp(2, 3, 4)” gibi bir şey yazdığımızda hata alacağımızı biliyoruz. İşte şimdi bu engeli nasıl aşacağımızı göreceğiz. Dikkatlice bakın:

```
def çarp(*sayılar):
    a = 1
    for i in sayılar:
        a = a * i
    print(a)

çarp(2, 3, 4)
```

Burada tek fark, parantez içindeki “sayılar” parametresinin solundaki ufacık bir “*” işaretinden ibaret. Bu yıldız işareti, fonksiyonumuzun alabileceği kadar parametreyi alabilmesini sağlar. Hatta bu işaret, fonksiyonumuzu parametresiz olarak da çağırmamıza olanak tanır. Bu arada ek bilgi olarak, bir fonksiyonun en fazla 255 parametre alabileceğini söyleyelim...

Yıldız işareti, fonksiyona argüman olarak verilen bütün değerlerin bir demet haline getirilmesini sağlar. Bunu şu şekilde teyit edebiliriz:

```
def çarp(*sayılar):
    print(sayılar)

çarp(2, 3, 4)
```

Bu kodları çalıştırdığımızda şöyle bir çıktı elde ederiz:

```
(2, 3, 4)
```

Çıktıdaki parantez işaretlerinden de anladığımız gibi, bu fonksiyon bize bir demet veriyor... Aslında burada olan şey, basit bir "demetleme" (tuple packing) işleminden ibarettir. Yazdığımız fonksiyonlarda "*" işaretini kullandığımız zaman, Python fonksiyona argüman olarak verilen öğeleri bir demet haline getirir. Böylece demet haline gelmiş bu öğeler birbirleriyle işleme sokulabilir.

Python'daki bu özelliğin yapısını gösteren basit bir örnek verelim:

```
def test(arg, *arglar):
    print("Bu fonksiyondaki sıralı argümanımız şudur: ", arg)
    print("Bu fonksiyondaki isimsiz argümanlarımız şunlardır: ", *arglar)

test("bir", "iki", "üç", "dört", "beş")
```

Yukarıdaki fonksiyon bir adet sıralı argüman ve rastgele sayıda isimsiz argüman alıyor. Bu arada, "isimsiz argüman" sözü kafanızı karıştırmasın. Aslında isimsiz argüman dediğimiz şey "sıralı argüman"ın ta kendisidir... Ancak yıldızlı yapılarda sıra söz konusu olmadığı için, bu argümanları "sıralı argüman" yerine "isimsiz argüman" (non-keyword argument) olarak adlandırıyoruz. Eğer istersek yukarıdaki test fonksiyonunu şöyle de yazabiliriz:

```
def test(arg, *arglar):
    print("Bu fonksiyondaki sıralı argümanımız şudur: ", arg)

    for i in arglar:
        print("Bu fonksiyondaki isimsiz argümanlardan biri şudur: ", i)

test("bir", "iki", "üç", "dört", "beş")
```

Bu yıldız işaretini yukarıda gösterdiğimiz şekilde fonksiyonun tanımlandığı sırada kullanabileceğimiz gibi, fonksiyonu çağırma aşamasında da kullanabiliriz. Yani şöyle bir şey yazabiliriz:

```
def test(arg, *arglar):
    print("Bu fonksiyondaki sıralı argümanımız şudur: ", arg)

    for i in arglar:
        print("Bu fonksiyondaki isimsiz argümanlardan biri şudur: ", i)

li = ["bir", "iki", "üç", "dört", "beş"]

test(*li)
```

Burada `test()` adlı fonksiyonu çağırırken, argümanları doğrudan parantez içine yazmak yerine, bu argümanları "li" adlı bir liste içinde topladık. Ardından da `test()` fonksiyonu içinde bu "li" adlı listeyi yıldız işaretiyle birlikte kullanarak istediğimiz sonucu elde ettik.

Yıldız işaretini, yukarıda anlattıklarımızdan farklı bir işlevi yerine getirmek için de kullanabiliriz. Bu işlevin ne olduğunu isterseniz bir örnek üzerinde görelim...

Hatırlarsanız, isimli argümanları kullanarak şöyle bir şey yazabiliyorduk:

```
def topla(bir, iki):
    print(bir + iki)
```

Tanımladığımız bu fonksiyonu şu şekilde çağırabiliriz:

```
topla(bir=3, iki=5)
```

Bildiğiniz gibi, isimli argümanları kullanmak, fonksiyon parametrelerini istediğimiz bir sırada kullanmamıza olanak veriyor. Yani yukarıdaki `topla()` adlı fonksiyonu şu şekilde de çağırabiliyoruz:

```
topla(iki=5, bir=3)
```

İsimli argümanlar bizi parametre sırasını ezberleme zahmetinden kurtarıyor. Ama elbette istersek biz yukarıdaki fonksiyonu şu şekilde de çağırabiliriz:

```
topla(3, 5)
```

Bu fonksiyonu bu şekilde çağırdığımızda, "bir" ve "iki" adlı parametreleri sıralı argüman olarak kullanmış olduk. Dolayısıyla, `topla()` fonksiyonuna verdiğimiz "3" argümanı "bir" parametresinin; "5" argümanı ise "iki" adlı parametrenin yerini tutmuş oldu... Gelelim yıldız işaretinin burada anlattığımız durumla ne ilgisi olduğuna...

Gördüğünüz gibi, tanımladığımız fonksiyonların parametrelerini hem isimli hem de sıralı argüman olarak kullanabiliyoruz. Ancak bazı durumlarda, kullanıcının sadece isimli argümanlar kullanmasına izin vermek istiyor olabiliriz... Mesela yukarıdaki örnekte şöyle bir kullanıma müsaade etmek istemiyor olabiliriz:

```
topla(3, 5)
```

Biz istiyoruz ki `topla()` fonksiyonu sadece isimli argümanlar ile çağrılabilsin... Eğer isteğimiz sadece isimli argümanlara müsaade etmek ise şöyle bir yol izleyebiliriz:

```
def topla(*, bir, iki):
    print(bir + iki)
```

Kodlarımızı bu şekilde yazdığımızda, yıldız işaretinin sağında kalan bütün parametrelerin, fonksiyonun çağrılması esnasında isimli argüman olarak kullanılmaları gerekir. Dolayısıyla yukarıdaki `topla()` fonksiyonunu ancak şu şekilde kullanabiliriz:

```
topla(bir=4, iki=6)
```

Yani sadece şöyle bir yazım bizi hüsrana uğratacaktır:

```
topla(4, 6)

TypeError: topla() takes exactly 0 positional arguments (2 given)
```

Burada aldığımız hata mesajı bize şöyle diyor:

```
TipHatası: topla() tamı tamına 0 adet sıralı argüman alır (2 adet verilmiş)
```

Gördüğünüz gibi, `topla()` fonksiyonunu kullanabilmek için mutlaka isimli argümanları kullanmamız gerekiyor. Parametrelerin en başına yerleştirdiğimiz "*" işareti sayesinde, bu işaretin sağında kalan hiç bir parametreyi isimsiz olarak kullanamıyoruz...

Bu arada, hatırlarsanız, bu bölümün en başında şöyle bir fonksiyon yazmıştık:

```
def çarp(*sayılar):
    a = 1
    for i in sayılar:
        a = a * i
    print(a)

çarp(2, 3, 4)
```

Yıldızlı parametre sayesinde bu fonksiyonu çağırırken istediğimiz sayıda argüman kullanabiliyoruz. Ancak bu fonksiyonun bir problemi var. Bu fonksiyona argüman olarak tek tek öğeler değil de bir liste (veya demet) verirsek beklediğimiz çıktıyı alamıyoruz:

```
def çarp(*sayılar):
    a = 1
    for i in sayılar:
        a = a * i
    print(a)

li = [2, 3, 4]
çarp(li)

[2, 3, 4]
```

Gördüğünüz gibi, fonksiyonumuz 2, 3 ve 4 sayılarını birbirleriyle çarpmak yerine bu sayıları ekrana bastı... Ama bizim istediğimiz şey bu değil. Burada öyle bir kod yazalım ki, kullanıcı argümanları liste halinde de verse, tek tek de yazsa, `çarp()` fonksiyonumuz bütün sayıları birbiriyle çarpabilsin...

```
def çarp(*sayılar):
    a = 1
    try:
        for k in sayılar[0]:
            a = a * k
        print(a)
    except TypeError:
        for v in sayılar:
            a = a * v
        print(a)
```

Artık `çarp()` fonksiyonuna argüman olarak ister liste verelim, ister tek tek öğe yazalım, fonksiyonumuz bütün öğeleri başarıyla çarpacak ve sonucu ekrana basacaktır... Yukarıdaki kodlarda neler olup bittiğini anlayabilmek için isterseniz bu kodları biraz daha yakından inceleyelim. Öncelikle "sayılar" adlı parametrenin ne çıktı verdiğini kontrol edelim:

```
def çarp(*sayılar):
    print(sayılar)
```

Şimdi bu fonksiyonu bir liste ile çağıralım:

```
li = [2, 3, 4]
çarp(li)
```

Buradan şu çıktıyı alıyoruz:

```
([2, 3, 4],)
```

Gördüğünüz gibi, çıktımız tek öğeli bir demet... Bu demetin mevcut tek öğesi de bir liste... Bu listenin öğelerine erişmek için fonksiyonumuzu şöyle yazmamız gerekiyor:

```
def çarp(*sayılar):
    for k in sayılar[0]:
        print(k)
```

Şimdi fonksiyonumuzu çağırıyoruz:

```
li = [2, 3, 4]
çarp(li)
```

```
2
3
4
```

Gördüğünüz gibi, bu şekilde listenin bütün öğelerini tek tek aldık... Tek tek aldığımız bu öğeleri birbirleriyle çarpmak için şöyle bir şey yazmamız gerektiğini biliyoruz:

```
def çarp(*sayılar):
    a = 1
    for k in sayılar[0]:
        a = a * k
    print(a)

li = [2, 3, 4]
çarp(li)
```

Kodlarımızı bu şekilde yazdığımızda, eğer `çarp()` fonksiyonuna argüman olarak bir liste verirsek fonksiyonumuz sorunsuz bir şekilde çalışacaktır. Ama eğer argüman olarak tek tek sayılar yazarsak Python bize bir hata mesajı gösterecektir:

```
çarp(2, 3, 4)

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

Çünkü daha önce de gördüğümüz gibi, "int" (sayı) veri tipi döngülenebilen bir nesne değildir. Dolayısıyla "int" veri tipi üzerinde bir `for` döngüsü kuramıyoruz... O halde bizim burada yapmamız gereken, bu hatayı yakalayıp buna göre bir işlem yapmak... Yani kodlarımız taslak olarak şöyle görünmeli:

```
def çarp(*sayılar):
    a = 1
    try:
        for k in sayılar[0]:
            a = a * k
        print(a)
    except TypeError:
        pass
```

Burada "pass" ile gösterdiğimiz yere uygun kodları yazacağız... O kısma, bu bölümün en başında gösterdiğimiz kodları yazmamız yeterli olacaktır:

```
def çarp(*sayılar):
    a = 1
    try:
        for k in sayılar[0]:
            a = a * k
        print(a)
    except TypeError:
        for v in sayılar:
            a = a * v
        print(a)
```

Böylelikle, "TypeError" adlı hata alındığında, Python `çarp()` fonksiyonuna verilen argümanların döngülenebilen bir nesne olmadığını anlayacak ve bu argümanları alıp döngülenebilen bir nesne olan demete dönüştürecektir...

10.7 Rastgele Sayıda İsimli Argüman Verme

Bir önceki bölümde fonksiyonları kullanarak, nasıl rastgele sayıda isimsiz argüman verebileceğimizi öğrenmiştik. Bu bölümde ise rastgele sayıda isimli argüman verme konusunu inceleyeceğiz.

Bildiğiniz gibi isimli argümanlar, adından da anlaşılacağı gibi, bir isimle birlikte kullanılan argümanlardır. Örneğin şu fonksiyonda isimli argümanlar kullanılmıştır:

```
def adres_defteri(isim, soyisim, telefon):
    defter = {}

    defter["isim"] = isim
    defter["soyisim"] = soyisim
    defter["telefon"] = telefon

    for k, v in defter.items():
        print("{0}\t:\t{1}".format(k, v))

adres_defteri(isim="Fırat", soyisim="Özgül", telefon="02122121212")
```

Burada öncelikle “defter” adlı bir sözlük tanımladık. Adres bilgilerini bu sözlük içine kaydeceğiz. Daha sonra bu “defter” adlı sözlük içinde yer alacak alanları belirliyoruz. Yazdığımız kodlara göre defter adlı sözlük içinde “isim”, “soyisim” ve “telefon” olmak üzere üç farklı alan bulunacak. Fonksiyon parametresi olarak belirlediğimiz “isim”, “soyisim” ve “telefon” öğelerine karşılık gelen değerler, defter adlı sözlükteki ilgili alanlara yerleştirilecek.

Daha sonra gelen satırda şöyle bir kod görüyoruz:

```
for k, v in defter.items():
    print("{0}\t:\t{1}".format(k, v))
```

Burada “defter” sözlüğünün `items()` metodunu kullanarak, sözlük içindeki “anahtar” ve “değer” çiftlerini birer demet halinde alıyoruz. Eğer yukarıdaki kodu şöyle yazacak olursanız neler olup bittiği biraz daha netleşecektir:

```
for öğeler in defter.items():
    print(öğeler)

('soyisim', 'Özgül')
('adres', 'İstanbul')
('telefon', '02122121212')
('eposta', 'firatozgul@frtzgl.com')
('cep', '05994443322')
('isim', 'Fırat')
```

Gördüğünüz gibi, yukarıdaki kod bize defter adlı sözlüğün “anahtar” ve “değer” çiflerini içeren birer demet veriyor. “for k, v in defter.items()” satırı ile bu demetlerdeki öğelere tek tek erişebiliyoruz. Burada “k” harfi, demetlerin ilk öğelerini, “v” harfi ise ikinci öğelerini temsil ediyor. Alt satırdaki “print(“{0}t:t{1}”.format(k, v))” kodu yardımıyla yaptığımız şey, “k” ve “v” ile temsil edilen öğeleri biraz biçimlendirerek ekrana basmaktan ibaret... Bu satır içinde gördüğümüz “\t” harflerinin ne işe yaradığını biliyoruz. Bunlar karakter dizileri içine sekme (tab) eklemek için kullanılan kaçış dizileridir.

Bu fonksiyona göre, “isim”, “soyisim” ve “telefon” alanlarını doldurarak fonksiyonu çalışır hale getirebiliyoruz. Şimdi yukarıdaki fonksiyona şöyle bir ekleme yapalım:

```
def adres_defteri(isim, soyisim, telefon, **arglar):
    defter = {}

    defter["isim"] = isim
    defter["soyisim"] = soyisim
    defter["telefon"] = telefon

    for i in arglar.keys():
        defter[i] = arglar[i]

    for k, v in defter.items():
        print("{0}\t:\t{1}".format(k, v))

adres_defteri(isim="Fırat",
              soyisim="Özgül",
              telefon="02122121212",
              eposta="firatozgul@frtzgl.com",
              adres="İstanbul",
              cep="05994443322")
```

Burada yaptığımız eklemeler sayesinde `adres_defteri()` adlı fonksiyona "isim", "soyisim" ve "telefon" parametrelerini yerleştirdikten sonra istediğimiz sayıda başka isimli argümanlar da belirtebiliyoruz. Burada "**arglar" parametresinin bir sözlük (dictionary) olduğuna özellikle dikkat edin. Bu parametre bir sözlük olduğu için, sözlüklerin bütün özelliklerine de doğal olarak sahiptir. Yukarıdaki kodları şu şekilde yazarak, arka planda neler olup bittiğini daha açık bir şekilde görebiliriz:

```
def adres_defteri(isim, soyisim, telefon, **arglar):
    print("isim:\n\t", isim)
    print("soyisim:\n\t", soyisim)
    print("telefon:\n\t", telefon)
    print("öteki argümanlar:\n\t", arglar)


adres_defteri(isim="Fırat",
              soyisim="Özgül",
              telefon="02122121212",
              eposta="firatozgul@frtzgl.com",
              adres="İstanbul",
              cep="05994443322")
```

Bu kodları çalıştırdığımızda şuna benzer bir çıktı alacağız:

```
isim:
         Fırat
soyisim:
         Özgül
telefon:
         02122121212
öteki argümanlar:
         {'adres': 'İstanbul',
          'eposta': 'firatozgul@frtzgl.com',
          'cep': '05994443322'}
```

Gördüğünüz gibi, "arglar" parametresi bize bir sözlük veriyor. Dolayısıyla şöyle bir kod yazmamız mümkün olabiliyor:

```
for i in arglar.keys():
    defter[i] = arglar[i]
```

Bu kodlar yardımıyla “arglar” adlı sözlüğün öğelerini “defter” adlı sözlüğe ekliyoruz. Bu yapının kafanızı karıştırmasına izin vermeyin. Aslında yaptığımız şey çok basit. Sözlükler konusunu anlatırken verdiğimiz şu örneği hatırlayın:

```
sözlük = {}

sözlük["ayakkabı"] = 2
sözlük["elbise"] = 1
sözlük["gömlek"] = 4
```

Bu örnekte “sözlük” adlı sözlüğe “ayakkabı”, “elbise” ve “gömlek” adlı öğeler ekliyoruz. Bu öğelerin değerini sırasıyla “2”, “1” ve “4” olarak ayarlıyoruz. Dediğim gibi, yukarıdaki örnek “for i in arglar.keys()...” satırıyla yaptığımız şeyden farklı değildir. Şöyle düşünün:

```
defter = {}

defter["adres"] = "İstanbul"
defter["eposta"] = "firatozgul@frtzgl.com"
defter["cep"] = "05994443322"
```

“defter[i]” dediğimiz şey, “adres”, “eposta” ve “cep” öğelerine; “arglar[i]” dediğimiz şey ise “İstanbul”, “firatozgul@frtzgl.com” ve “05994443322” öğelerine karşılık geliyor.

Gördüğünüz gibi, çift yıldızlı parametreler fonksiyonlara istediğimiz sayıda isimli argüman ekleme imkanı tanıyor bize... Dediğim gibi, bu parametreler bir sözlük olduğu için yukarıdaki örneği şu şekilde de yazabilirsiniz:

```
def adres_defteri(isim, soyisim, telefon, **arglar):
    defter = {}

    defter["isim"] = isim
    defter["soyisim"] = soyisim
    defter["telefon"] = telefon

    for i in arglar.keys():
        defter[i] = arglar[i]

    for k, v in defter.items():
        print("{0}\t:\t{1}".format(k, v))

sözlük = {"eposta": "firatozgul@frtzgl.com",
          "adres": "İstanbul",
          "cep": "05994443322"}

adres_defteri(isim="Fırat",
              soyisim="Özgül",
              telefon="02122121212",
              **sözlük)
```

Burada sözlüğü önceden tanımladığımıza ve bunu fonksiyonu çağırırken doğrudan argüman olarak eklediğimize dikkat edin. Ayrıca “sözlük” argümanını fonksiyona yazarken yine çift yıldızlı yapıyı kullanmayı da unutmuyoruz.

10.8 Gömülü Fonksiyonlar

Python'daki bazı fonksiyonlara "gömülü fonksiyonlar" adı verilir. Örneğin daha önceki derslerimizde gördüğümüz `range()`, `sum()`, `len()` gibi fonksiyonlar bu sınıfa girer. Bu fonksiyonlara "gömülü fonksiyonlar" denmesinin nedeni, adından da anlaşılacağı gibi, bu fonksiyonların Python'un içine tam anlamıyla gömülü olmasıdır... Gömülü fonksiyonların karşıtı kullanıcı tanımlı fonksiyonlardır. Kullanıcı tanımlı fonksiyonlar, yine adından anlaşılacağı gibi, kullanıcının bizzat kendisi tarafından tanımlanmış, "el yapımı" fonksiyonlardır... Örneğin bir önceki bölümde tanımlamış olduğumuz `çarp()` fonksiyonu kullanıcı tanımlı bir fonksiyondur. Kullanıcı tanımlı fonksiyonların aksine gömülü fonksiyonlar Python'u geliştiren kişiler tarafından tanımlanmış ve Python'un içine yerleştirilmiştir. Gömülü fonksiyonlar istediğimiz her an kullanılabilir vaziyettedir. Yani mesela `len()` fonksiyonunu, yazdığımız bir program içinde kullanabilmek için bu fonksiyonu programımız içinde yeniden tanımlamamıza gerek yoktur. Kullanıcı tanımlı fonksiyonları ise iki şekilde kullanabiliriz. Bu fonksiyonları ya mevcut program içinde bir yerlerde tanımlamış olmalıyız, ya da eğer bu fonksiyon başka bir dosya içinde tanımlanmışsa, o dosyayı (yani modülü) mevcut programımız içine aktarmalıyız. Bu "modül" ve "içe aktarma" kavramlarının kafanızı karıştırmasına izin vermeyin. Birkaç bölüm sonra bu kavramların ne anlama geldiğini ayrıntılı bir şekilde inceleyeceğiz. Burada amacımız sizi sadece "gömülü fonksiyonlar"ın varlığı konusunda bilgilendirmekten ibaret...

Python'daki bütün gömülü fonksiyonların listesini http://docs.python.org/dev/3.0/library/functions.html adresinde bulabilirsiniz. Orada da göreceğiniz gibi, listedeki gömülü fonksiyonların bir kısmını zaten biliyoruz...

Python'daki gömülü fonksiyonların özelliği, bu fonksiyonların kaynak kodlarının Python'la yazılmamış olmasıdır. Bu fonksiyonlar Python geliştiricileri tarafından C dili kullanılarak yazılmıştır. Gömülü fonksiyonların tanımlandığı C dosyasına http://svn.python.org/view/python/trunk/Python/bltinmodule.c?view=markup&pathrev=52315 adresinden erişebilirsiniz.

Bu fonksiyonlar C dili ile yazıldığı için son derece hızlı ve performanslı çalışırlar. Dolayısıyla eğer yapmak istediğiniz bir işlem için uygun bir hazır fonksiyon varsa, gerekli işlevi yerine getiren fonksiyonu kendiniz yazmak yerine Python içindeki o hazır fonksiyonu kullanmalısınız.

10.9 Fonksiyonların Kapsamı ve global Deyimi

Bu bölümde Python'daki "isim alanı" (namespace) ve kapsam (scope) kavramından söz edeceğiz biraz... Kabaca ifade etmek gerekirse "isim alanı" denen şey, Python'daki herhangi bir nesnenin etki alanıdır. Python'daki nesnelerin birer kapsama alanı vardır. Örneğin Python'daki fonksiyonlar birer nesnedir. Esasında Python'daki her şey bir nesnedir. Şimdilik bu "nesne" meselesini fazla kafanıza takmayın. Bu konuyu yeri geldiğinde esaslı bir şekilde inceleyeceğiz... Ne diyorduk? Evet, Python'daki fonksiyonlar birer nesnedir ve bu nesnelerin de bir kapsama alanı vardır. Ya da başka bir deyişle fonksiyonlar belirli bir isim alanı içinde yer alır. Hemen ufak bir örnek verelim:

```
def deneme():
    sayı = 5
    print(sayı)
```

Burada "sayı" adlı değişken, `deneme()` adlı fonksiyonun isim alanı içinde yer alır. Yani bu değişkenin kapsamı `deneme()` adlı fonksiyonla sınırlıdır. "sayı" adlı değişken bu fonksiyonun dışında var olamaz. Peki bu ne anlama geliyor? Yukarıdaki kodlara şöyle bir ekleme yaparak durumu birazcık da olsa somutlaştıralım:

```
def deneme():
    sayı = 5
    print(sayı)

print(sayı)
```

Bu kod parçasını çalıştırdığımızda Python bize şöyle bir hata mesajı gösterecektir:

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

Yani:

```
İsimHatası: "sayı" ismi tanımlanmamış
```

Halbuki biz "sayı" adlı değişkeni deneme() adlı fonksiyonun içinde tanımlamıştık, değil mi? Evet, biz bu değişkeni "deneme" adlı fonksiyon içinde tanımlamıştık. Ama önümüzde şöyle bir gerçek var: Bu bölümün başında da söylediğimiz gibi, Python'daki nesnelerin bir kapsama alanı vardır. Yani bu nesneler bir isim alanı içinde yer alır. Dolayısıyla "sayı" adlı değişkenin kapsamı deneme() adlı fonksiyonun isim alanı ile sınırlıdır. Yani bu değişken deneme() adlı fonksiyonun dışında var olamaz... Bu "deneme" fonksiyonunun kapsama alanı da fonksiyonun içi ile sınırlıdır.

Şimdi şuna bir bakalım:

```
def deneme():
    sayı = 5
    print("deneme() fonksiyonu içindeki sayı: ", sayı)

sayı = 10
print("deneme() fonksiyonu dışındaki sayı: ", sayı)
deneme()
```

Gördüğünüz gibi, "sayı" adlı değişkeni fonksiyon dışında da kullanabilmek için, bu değişkeni fonksiyonun dışında da tanımlamamız gerekiyor. Yukarıda iki farklı değerle gösterilen "sayı" adlı değişkenler farklı isim alanları içinde yer aldığı için, aynı ada sahip olmalarına rağmen birbirlerine karışmıyorlar. Aslında "isim alanı" kavramı çok güzel bir özelliktir. Büyük programlar yazarken isim alanı kavramının çok işinize yaradığını göreceksiniz. Bu kavram sayesinde değişkenlerin birbirine karışmasını önleyebiliyoruz. Hele bir de aynı program üzerinde farklı kişiler çalışıyorsa, bu "isim alanı" özelliği hayat kurtarıcı olabilir...

Ancak bazı durumlarda bir isim alanı içinde yer alan bir değişkene o isim alanı dışından da erişebilmeniz gerekebilir. Mesela şöyle bir şey yazmak isteyebilirsiniz:

```
def sor():
    sayı1 = int(input("bir sayı: "))
    sayı2 = int(input("başka bir sayı: "))

def topla():
    print(sayı1 + sayı2)

try:
    sor()
except ValueError:
    pass
else:
    topla()
```

Burada sayı alma ve alınan bu sayıları toplama işlemleri için ayrı birer fonksiyon oluşturduk.

Yukarıdaki yapıya göre, `sor()` fonksiyonu içinde geçen "sayı1" ve "sayı2" adlı değişkenleri `topla()` fonksiyonu içinde işleme tabi tutabilmemiz gerekiyor. Ancak burada bir problem var. Python'un yapısı gereğince, `sor()` ve `topla()` adlı fonksiyonlar farklı isim alanlarına, dolayısıyla da farklı kapsamlara sahip... Bundan ötürü, bu iki fonksiyon içindeki değişkenlere dışarıdan erişemiyoruz. Zaten yukarıdaki kodları çalıştırdığımızda Python bize bununla ilgili bir hata mesajı gösterecektir...

Yukarıdaki sorunu aşabilmek için Python bize `global` adlı bir araç sunuyor... `global` deyimi yardımıyla, bir isim alanı içindeki değişkenlere o isim alanı dışından da erişebiliyoruz. Mesela yukarıdaki örneği şöyle yazalım:

```
def sor():
    global sayı1
    global sayı2

    sayı1 = int(input("bir sayı: "))
    sayı2 = int(input("başka bir sayı: "))

def topla():
    print(sayı1 + sayı2)

try:
    sor()

except ValueError:
    pass

else:
    topla()
```

Bu kodların başına eklediğimiz "global sayı1" ve "global sayı2" ifadeleri sayesinde "sayı1" ve "sayı2" adlı değişkenlere fonksiyon dışından da erişilebilmesini sağlıyoruz. Yukarıdaki kodları çalıştırdığımızda artık programımız hatasız bir şekilde görevini yerine getirecektir. Sayı dışında bir değer girilmesi durumunda dahi programımızın çökmemesi için `try... except...` bloklarından yararlandığımıza dikkat ediyoruz...

Bu arada eğer istersek yukarıdaki "global sayı1" ve "global sayı2" ifadelerini birleştirebiliriz:

```
def sor():
    global sayı1, sayı2
    ...
    ...
```

Gelin isterseniz tanıdık bir program üzerinde uygulayalım öğrendiklerimizi...

```
def sor():
    global sayı1
    global sayı2

    sayı1 = int(input("bir sayı: "))
    sayı2 = int(input("başka bir sayı: "))

def topla():
    print(sayı1 + sayı2)

def çıkar():
    print(sayı1 - sayı2)

def çarp():
```

```
    print(sayı1 * sayı2)

def böl():
    try:
        print(sayı1 / sayı2)
    except ZeroDivisionError:
        print("Bir sayıyı sıfıra bölemezsiniz!")

print("""Topla >> 1
Çıkar >> 2
Çarp  >> 3
Böl   >> 4
Çıkış >> 5
""")

while True:
    seçenek = input("Yapmak istediğiniz işlem: ")
    if seçenek == "5":
        break
    else:
        try:
            sor()
        except ValueError:
            print("Yanlış seçenek!..")
        else:
            if seçenek == "1":
                topla()
            elif seçenek == "2":
                çıkar()
            elif seçenek == "3":
                çarp()
            elif seçenek == "4":
                böl()
```

Gördüğünüz gibi, `global` deyimi epey işe yarıyor. Ancak size bu deyimle ilgili kötü bir haberim var. Bu deyim ne kadar faydalıymış gibi görünse de aslında çoğu zaman işleri karıştırabiliyor. Şöyle bir örnek verelim:

```
a = 5

def değişkeni_değiştir():
    a = 10
    print("bu değişken fonksiyon içinde: ", a)

değişkeni_değiştir()

print("bu değişken fonksiyon dışında: ", a)
```

Bu kodları çalıştırdığımızda şöyle bir çıktı alıyoruz:

```
bu değişken fonksiyon içinde:  10
bu değişken fonksiyon dışında:  5
```

Bir de şuna bakalım:

```
a = 5

def değişkeni_değiştir():
    global a
```

```
    a = 10
    print("bu değişken fonksiyon içinde: ", a)

değişkeni_değiştir()

print("bu değişken fonksiyon dışında: ", a)
```

Bu kodlar ise bize şöyle bir çıktı veriyor:

```
bu değişken fonksiyon içinde:  10
bu değişken fonksiyon dışında:  10
```

Gördüğünüz gibi, fonksiyon içindeki işlem fonksiyonun dış bölgesini de etkiledi ve normalde değeri "5" olması gereken "a" adlı değişkeni değişikliğe uğrattı. `global` deyimi bu özelliğinden ötürü hiç beklenmedik bir anda programınızı allak bullak edebilir. Özellikle büyük boyutlu ve birden fazla kişinin üzerinde çalıştığı programlarda `global` deyimi züccaciye dükkanına fil girmiş gibi bir etki yaratabilir... O yüzden Python programcıları genellikle `global` deyimini kullanmaktan kaçınır. Hatta Python camiasında şöyle bir söz vardır: "Eğer yazdığınız programda `global` deyimini kullanmanız gerektiğini düşünüyorsanız, programınızı gözden geçirmenizin zamanı gelmiş demektir!"

Python'da `global` yerine benimseyebileceğimiz daha sağlıklı yollar vardır. Mesela sınıflı yapıları kullanmak gibi... Hiç endişe etmeyin. Yeri geldiğinde sınıflı yapıları da enine boyuna inceleyeceğimizden emin olabilirsiniz. Peki `global` deyimini hangi durumlarda kullanmamızın bir sakıncası yoktur? Esasında, dediğimiz gibi, en iyisi kişinin kendini `global` deyimine hiç alıştırmamasıdır. Ama eğer uğraştığınız kod ufak boyutlu bir şeyse ve sizden başka kimsenin bu kodlar üzerinde çalışmayacağından eminseniz, o anda karşılaştığınız bir sorunu çözmek için `global` deyimine başvurmayı tercih edebilirsiniz... Elbette `global` deyimini kullanmak günah değildir! Sadece, bunu kullanmak iyi bir programcılık tekniği olarak kabul edilmez...

10.10 return Deyimi

Bu bölümde inceleyeceğimiz bir başka önemli konu da `return` deyimi olacak. `return` kelime olarak "döndürmek" anlamına gelir. Ama "döndürmek" derken bir şeyi kendi etrafında çevirmeyi kastetmiyoruz. Buradaki "döndürmek" ifadesi daha çok "iade etmek" veya "vermek" kavramına yakındır... Peki fonksiyonlarla bu kavramın ne ilgisi olabilir? İsterseniz bunu bir örnekle görmeye çalışalım:

```
bool(5>2)

True
```

Burada `bool()` fonksiyonunu kullanarak "5" sayısının "2" sayısından büyük olup olmadığını sorguladık. `bool()` fonksiyonu da bize karşılık olarak True değerini verdi. İşte bu durum için, "`bool()` fonksiyonu True değerini döndürdü," diyoruz... Şu örneğe bakalım:

```
li = [2, 3, 4]
sum(li)

9
```

Burada da `sum()` fonksiyonu bir sayı döndürüyor...

İşte `return` deyimi Python'da bu anlama geliyor. Yani kabaca bir fonksiyonun değer üretmesi, bir sonuç ortaya çıkarması "döndürmek" (return) olarak adlandırılıyor. Peki biz bu deyimi fonksiyonlarımız içinde nasıl kullanacağız? Daha doğrusu, fonksiyonlarımızın bir değer

döndürmesini nasıl sağlayacağız? Esasında Python'daki bütün fonksiyonlar, biz belirtsek de belirtmesek de mutlaka bir değer döndürür. Şu örneği inceleyelim:

```
def bas(kelime):
    print(kelime)

a = bas("www.istihza.com")
print(a)
```

Bu kodları çalıştırdığımızda şöyle bir sonuç elde edeceğiz:

```
www.istihza.com
None
```

İşte bu çıktıda gördüğümüz "None", yazdığımız fonksiyonun dönüş değeridir. "www.istihza.com" ise fonksiyon içinde yazdığımız `print()` fonksiyonunun marifetidir ve onun yaptığı şey `return`'un yaptığı şeyden farklıdır. `return` ile `print()` aynı şeyler değildir ve aynı işlevi görmezler.

Yukarıdaki örneğe göre, Python'daki bütün fonksiyonlar bir değer döndürüyor. Özel olarak bir dönüş değeri belirtilmezse, fonksiyonun döndüreceği değer "None" olacaktır. "Eee, ne olmuş?" dediğinizi duyar gibiyim... Konuyu biraz daha somutlaştırmak için şöyle bir örnek verelim:

```
def santtan_faha(sant):
    fah = sant * (9/5) + 32
    return fah
```

Burada kendisine verilen santigrat dereceyi fahrenhayt'a çeviren bir uygulama yazdık... Bu fonksiyonu şu şekilde çağırıyoruz:

```
print(santtan_faha(22))
```

Yani kodlarımız tam olarak şöyle görünecek:

```
def santtan_faha(sant):
    fah = sant * (9/5) + 32
    return fah

print(santtan_faha(22))
```

Buradan şu çıktıyı alıyoruz:

```
71.6
```

Demek ki 22 santigrat derece 71.6 fahrenhayt'a karşılık geliyormuş... Neyse... İşin bizi ilgilendiren asıl kısmı bu değil. Biz buradaki `return` deyiminin ne işe yaradığını anlamaya çalışıyoruz.

Burada fonksiyonu nasıl çağırdığımıza dikkat edin. Daha önceki fonksiyonlarımızda şöyle bir çağırma şekli kullanıyorduk:

```
santtan_faha(22)
```

Burada `print()` fonksiyonunu kullanmadığımıza özellikle dikkatinizi çekmek isterim. Peki neden önceki örneklerde `print()` kullanmadık da yukarıdaki örnekte `print()` kullandık? Önceki örneklerde `print()` kullanmamamızın nedeni, o örneklerde hep fonksiyon içinde `print()`'i kullanmış olmamızdır. Yani önceki örneklerde yazdığımız fonksiyonlar zaten belli bir değeri ekrana basıyordu. Yukarıdaki fahrenhayt fonksiyonunda ise fonksiyon içinde herhangi bir `print()` işlemi yapmadık. Dolayısıyla fonksiyonumuz kendi başına herhangi bir değeri ekrana basma özelliğine sahip değil. Bu fonksiyonun görevi "fah" değişkenini "döndürmekten"

ibaret... Mesela daha önce bahsettiğimiz, gömülü fonksiyonlardan biri olan `sum()` fonksiyonu da ekrana herhangi bir değer basmaz. Bu fonksiyonun görevi kendisine parametre olarak verilen döngülenebilir nesnenin öğelerinin toplamını "döndürmek"tir. Görelim:

```
li = [2, 3, 4]
sum(li)
```

Bu kodları doğrudan bir dosyaya kaydedip çalıştırdığımızda herhangi bir çıktı almayız. Çünkü `sum()` fonksiyonu kendi başına herhangi bir değeri ekrana basmaz. Bu fonksiyonun bir değeri ekrana basabilmesi için şöyle bir şey yapmamız gerekir:

```
print(sum(li))
```

Aynen `sum()` fonksiyonunda olduğu gibi, yukarıda tanımladığımız `santtan_faha()` adlı fonksiyon da ekrana herhangi bir değer basmıyor. Bu fonksiyon sadece "fah" adlı değişkenin değerini "döndürüyor". Fonksiyonumuzun döndürdüğü bu değerle ne yapacağımız tamamen bize kalmış. Biz bu değeri istersek ekrana basarız, istersek bununla başka işlemler yaparız. Bu durumu biraz açıklayalım. Şu örneğe bakın:

```
def santtan_faha(sant):
    fah = sant * (9.0/5.0) + 32
    return fah

print(santtan_faha(22)-1)
```

Bu fonksiyon bize şu çıktıyı verir:

```
70.6
```

Burada `santtan_faha()` fonksiyonunun döndürdüğü değerden 1 çıkardık... Aynı işlemi, fonksiyonu şu şekilde tanımlayarak yapmaya çalışalım:

```
def santtan_faha(sant):
    fah = sant * (9.0/5.0) + 32
    print(fah)

print(santtan_faha(22)-1)
```

Gördüğünüz gibi bu defa Python bize bir hata mesajı veriyor:

```
TypeError: unsupported operand type(s) for -: 'NoneType' and 'int'
```

Yani:

```
TipHatası: - işaretinin desteklemediği işleyen türü/türleri: "NoneType" ve "int"
```

Bu hata mesajı bize, birbiriyle çıkarma işlemi yapmanın mümkün olmadığı öğeler kullandığımızı söylüyor... Peki bu nasıl oluyor?

Hatırlarsanız, Python'da bütün fonksiyonların bir değer döndürdüğünü, eğer biz özel olarak herhangi bir değer döndürmezsek, döndürülecek değerin "None" olacağını söylemiş ve bu durumu da şu şekilde teyit etmiştik:

```
def bas(kelime):
    print(kelime)

a = bas("www.istihza.com")
print(a)
```

Burada bas() fonksiyonu "www.istihza.com" karakter dizisini ekrana basıyor, ama "None" değerini "döndürüyor". Yukarıda hata verdiğini gördüğümüz santtan_faha() fonksiyonunda da özel olarak herhangi bir değer döndürmediğimiz için bu fonksiyon otomatik olarak "None" değerini döndürüyor. Dikkat ederseniz, bu defa "fah" değişkenini döndürmek yerine doğrudan print()'i kullanarak ekrana bastık. Yani burada değeri döndürmedik, ama ekrana basmakla yetindik. Bu yüzden santtan_faha() fonksiyonunun dönüş değeri "None" oldu. Dolayısıyla biz *print(santtan_faha(22)-1)* komutuyla sanki şöyle bir işlem yapıyormuş gibi olduk:

```
print(None - 1)
```

Python'un etkileşimli kabuğunda "None - 1" komutunu verdiğinizde şu hatayı alacaksınız:

```
TypeError: unsupported operand type(s) for -: 'NoneType' and 'int'
```

Gördüğünüz gibi, biraz önceki fonksiyonumuzun verdiği hatayla aynı hatayı aldık... Çünkü orada olan şeyle burada olan şey tamamen aynıdır. "None" değerinden "1" değerini çıkaramayacağımız için Python bize böyle bir hata veriyor. Buradan return deyiminin ne işe yaradığını az çok anlamış olmalıyız. Python'da fonksiyon tanımlarken doğrudan fonksiyon içinde herhangi bir değeri print() ile ekrana basmak her zaman iyi bir yol olmayabilir. Çünkü daha önce de dediğimiz gibi, fonksiyonlar şablonlara benzer. O yüzden fonksiyonları ne kadar genel tutarsak o kadar işe yarar. Bir fonksiyonun görevi sadece bir değeri ekrana basmak olmamalı. Tanımladığımız fonksiyonlarda herhangi bir değeri doğrudan ekrana basmak yerine o değeri "döndürmeyi" (return) tercih edersek, fonksiyondan dönen değer ile istediğimiz işlemi yapabiliriz. Artık dönen bu değeri sonradan ekrana basmak veya bu değerle başka işlemler yapmak tamamen size kalmış bir şey olacaktır...

Mesela yukarıdaki fonksiyonla şöyle bir deneme yapalım:

```
def santtan_faha(sant):
    fah = sant * (9.0/5.0) + 32
    return fah

santigrat = 20
print("{0} santigrat {1} fahrenhayt eder".format(santigrat,
                                                 santtan_faha(santigrat)))
```

Bu fonksiyonu çalıştırdığımızda istediğimiz çıktıyı alabiliriz. Ama eğer fonksiyonu şöyle tanımlarsak sıkıntı yaşarız:

```
def santtan_faha(sant):
    fah = sant * (9.0/5.0) + 32
    print(fah)

santigrat = 20
print("{0} santigrat {1} fahrenhayt eder".format(santigrat,
                                                 santtan_faha(santigrat)))
```

Buradan alacağımız çıktı şöyle olacaktır:

```
68.0
20 santigrat None fahrenhayt eder
```

Gördüğünüz gibi, fonksiyonumuzun döndürdüğü "None" değeri, karakter dizisinin içine yerleşti...

Sonuç olarak, return deyimi bir fonksiyonu genel amaçlı bir hale getirmeye yardımcı olan bir araçtır. Bu deyim yardımıyla, tanımladığımız fonksiyonların bir değer döndürmesini sağlıyoruz. Döndürülen bu değeri daha sonra istediğimiz gibi kullanma özgürlüğüne sahibiz...

10.11 Fonksiyonların Belgelendirilmesi

Bu bölümde kodlarımızın çalışmasını değil ama okunaklılığını etkileyecek bir özellikten söz edeceğiz. Bahsedeceğimiz özellik fonksiyonların belgelendirilmesi... "Fonksiyonların belgelendirilmesi de ne demek oluyor?" diye düşünmüş olabilirsiniz. İsterseniz size şöyle bir örnek göstererek konuya ısınmanızı sağlayalım:

Sıkça değindiğimiz fonksiyonlardan biri olan `sum()` fonksiyonunu ele alalım ve daha önce öğrendiğimiz `dir()` fonksiyonu yardımıyla bu fonksiyonun içine bakalım:

```
>>> dir(sum)

['__call__', '__class__', '__delattr__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__',
'__lt__', '__module__', '__name__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__self__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__']
```

Bu şekilde `sum()` fonksiyonunun niteliklerini ekrana dökmüş olduk. Bunlar içinde bizi ilgilendiren, dördüncü sıradaki `__doc__` niteliğidir. Şimdi bu `__doc__` niteliğini nasıl kullanacağımıza ve bu niteliğin ne işe yaradığına bakalım. Bu komutu etkileşimli kabukta veriyoruz:

```
>>> print(sum.__doc__)

sum(iterable[, start]) -> value

Returns the sum of an iterable of numbers (NOT strings) plus the value
of parameter 'start' (which defaults to 0).  When the iterable is
empty, returns start.
```

Gördüğünüz gibi, "*print(sum.__doc__)*" komutu bize `sum()` fonksiyonu hakkında kısa bir bilgi veriyor. Tabii eğer İngilizce bilmiyorsanız bu kısa açıklamaların size pek faydası olmayacaktır. Ama hiç endişe etmeyin! istihza.com bugünler için var!...

Burada `sum()` fonksiyonunun `__doc__` niteliğini kullanarak, bu fonksiyon hakkında kısa da olsa bilgi edinebiliyoruz. Bu `__doc__` niteliği bütün fonksiyonlarda vardır. Hatta kendi tanımladığımız fonksiyonlar dahi bu niteliğe sahiptirler. Hemen bir fonksiyon tanımlayarak bu durumu teyit edelim:

```
def boş_fonksiyon():
    pass
```

Bu arada, bu fonksiyonu isterseniz etkileşimli kabukta tanımlayın. Böylece fonksiyonu daha rahat test edebilirsiniz. Fonksiyonları etkileşimli kabukta tanımlamak için şöyle bir yol izliyoruz:

```
>>>def boş_fonksiyon():
...     pass
...
>>>
```

Gördüğünüz gibi `def` ile başlayan satırı yazıp enter'e bastığımızda etkileşimli kabuğun ">>>" işareti "..." işaretine dönüşüyor. Burada dört kez boşluk tuşuna basıp girinti veriyoruz ve "pass" satırını yazıyoruz. Ardından enter'e bastığımızda işaretin yine "..." olduğunu görüyoruz. Etkileşimli kabuk burada bizden başka kodlar bekliyor, ama bu fonksiyonda bizim yazacağımız başka kod yok. O yüzden tekrar enter'e basarak ">>>" işaretinin olduğu başlangıç konumuna geri dönüyoruz. Böylece etkileşimli kabukta fonksiyonumuzu tanımlamış olduk. Şimdi yine etkileşimli kabukta şu komutu vererek, yeni oluşturduğumuz bu "boş_fonksiyon" adlı fonksiyonun içine, yani niteliklerine bakabiliriz:

```
>>> dir(boş_fonksiyon)

['__annotations__', '__call__', '__class__', '__closure__', '__code__',
'__defaults__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__',
'__ge__', '__get__', '__getattribute__', '__globals__', '__gt__', '__hash__',
'__init__', '__kwdefaults__', '__le__', '__lt__', '__module__', '__name__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__']
```

Gördüğünüz gibi, bizim oluşturduğumuz fonksiyonun da `__doc__` adlı bir niteliği var. Gelin isterseniz biraz önce `sum()` fonksiyonuna yaptığımız gibi, "boş_fonksiyon"a da bu `__doc__` niteliğini uygulayalım:

```
>>> print(boş_fonksiyon)

None
```

Gördüğünüz gibi, fonksiyonumuz "None" değeri döndürdü. Çünkü fonksiyonumuzu henüz belgelendirmedik. O yüzden bu `__doc__` niteliği boş görünüyor. Şimdi bu fonksiyonu nasıl belgelendireceğimize bakalım:

```
def boş_fonksiyon():
    """Kullanımı: boş_fonksiyon() Hiç bir işe yaramayan boş bir fonksiyondur.
    None değerini döndürür."""
    pass

print(boş_fonksiyon.__doc__)
```

Bu kodları çalıştırdığımızda ekrana şu satırlar dökülecektir:

```
Kullanımı: boş_fonksiyon() Hiç bir işe yaramayan boş bir fonksiyondur.
None değerini döndürür.
```

Böylece fonksiyonumuzun `__doc__` niteliğini doldurmuş, yani fonksiyonumuzu belgelendirmiş olduk. Artık bu fonksiyon hakkında bilgi almak isteyenler bu fonksiyonun `__doc__` niteliğini kullanarak fonksiyonun ne yaptığı hakkında kısa bir bilgi sahibi olabilecekler. Burada fonksiyonumuzu nasıl belgelendirdiğimize dikkat edin. Bunun için "belgelendirme dizileri"nden (docstring) yararlandık. Fonksiyonu `def` parçacığı yardımıyla tanımladıktan hemen sonra getirdiğimiz açıklayıcı bilgilere "belgelendirme dizisi" adı veriyoruz. Python'da genel kural olarak belgelendirme dizileri üç tırnak içinde gösterilir. Belgelendirme dizilerini çift veya tek tırnakla da göstermek mümkün olsa da üç tırnak kullanmak adettendir. Ayrıca üç tırnak kullandığımızda uzun metinleri daha kolay yazabiliriz. Çünkü bildiğiniz gibi, üç tırnak kullanıldığında "enter" tuşuna basarak cümleleri rahatlıkla kaydırabiliyoruz...

Bu arada belgelendirme dizilerini yazarken girintilere dikkat ediyoruz. `def` parçacığı ile fonksiyonu tanımlayıp enter tuşuna bastıktan sonra belgelendirme dizisini girintili olarak yazmaya başlamamız gerekiyor. Yani şöyle bir şey yazamıyoruz:

```
def boş_fonksiyon():
"""belgelendirme dizisi..."""
```

Fonksiyonumuzu şöyle yazmamız gerekiyor:

```
def boş_fonksiyon():
    """belgelendirme dizisi..."""
```

Dediğimiz gibi, bir fonksiyonun belgelendirme dizisine ulaşmak için `__doc__` niteliğini kullanıyoruz. Alternatif olarak bu dizilere ulaşmak için `help()` adlı özel bir fonksiyondan da yararlan-

abiliriz:

```
>>> print(help(enumerate))
```

En başta da söylediğimiz gibi, belgelendirme dizilerinin kodlarımızın çalışması üzerinde hiç bir etkisi yoktur. Ayrıca bunları yazmak mecburiyetinde de değiliz. Bu özellik kodlarımızı daha anlaşılır ve okunaklı kılmaya yarar. Bu sayede, yazdığımız fonksiyonu okuyanlar veya kullanacak olanlar fonksiyonun nasıl kullanılacağı hakkında bilgilenme imkanı edinmiş olurlar. Elbette yazdığımız belgelendirme dizilerinin faydası, bu dizilerin ne kadar düzgün yazıldığına bağlıdır...

Ayrıca belgelendirme dizilerinin sadece fonksiyonlara has bir özellik olmadığını da belirtelim. Bir sonraki bölümde de göreceğiniz gibi, belgelendirme dizileri modüllerin de sahip olduğu bir özelliktir. Dolayısıyla modülleri belgelendirmek için yine bu belgelendirme dizilerinden yararlanacağız...

BÖLÜM 11

Modüller

Bu bölümde yine çok önemli bir konu olan “modüller”den bahsedeceğiz. Peki nedir modül? “Modül” en basit tanımıyla, ”.py” uzantısına sahip bir dosyadır. Bu tanıma göre, Python’la yazdığınız bütün program dosyaları birer modüldür. Dolayısıyla bir program yazıp bunu “cımbız.py” adıyla kaydettiğinizi varsayarsak, işte bu “cımbız.py” adlı dosya bir modüldür... Buradan yola çıkarak şöyle bir tanım yapabiliriz o halde:

> “Fonksiyonlar, deyimler ve ifadelerden oluşan, bu öğeleri içinde barındıran Python dosyalarına modül adı verilir.”

Peki modüller ne işe yarar?

Modüller Python’daki en yararlı öğelerden bir tanesidir. Bir modül içinde tanımladığımız fonksiyonları başka bir programa aktarabilir, bu fonksiyonları bir daha yazmak zorunda kalmadan tekrar tekrar kullanabiliriz.

Modüller Python’a güç ve esneklik kazandıran öğelerdir. Etrafta, emrimize amade bir şekilde bekleyen binlerce hazır modül bulunur. Bu modüllerden bir kısmı Python’un içindedir, bir kısmı ise internetten indirilebilmektedir. Hazır modülleri kullanarak Python’da inanılmaz işler başarabilirsiniz. Örneğin `Tkinter` adlı bir modül sayesinde Python’la arayüzlü programlar tasarlayabilirsiniz... Eğer matematiğe meraklıysanız `math` modülünü kullanarak karmaşık işlemleri son derece kolay bir şekilde halledebilirsiniz... Etrafta bunlar gibi bir yığın modül bulunur.

Python’un içinde yer alan modüllerin listesine http://www.python.org/doc/3.0.1/modindex.html adresinden ulaşabilirsiniz.

Yukarıda söylediğimiz sözler kulağa biraz anlaşılmaz geliyor olabilir. Ama hiç dert etmeyin. İlerleyen sayfalarda modül konusunu ayrıntılı olarak inceleyeceğiz. O halde lafı daha fazla dolandırmadan işe koyulalım.

11.1 Modülleri İçe Aktarmak (importing modules)

Python’da bir modülü kullanabilmek için, o modülü öncelikle “içe aktarmamız” gerekiyor. Burada “içe aktarmak”tan kasıt, söz konusu modüle ait özellikleri başka bir dosyanın içine taşımaktır. Bir modül içe aktarıldığında, o modüle ait özellikler, modülün aktarıldığı dosya içinden de kullanılabilecektir. Bu söylediklerimiz kafanızı karıştırmasın. Şimdi bu anlattıklarımızın ne demek olduğunu çok basit örneklerle açıklayacağız.

Python'da os adlı bir modül bulunur. Bu modül Python'un en temel ve en önemli modüllerinden bir tanesidir. Şimdi bir modülün ne olup ne olmadığını anlayabilmek için etkileşimli kabukta aşağıdaki komutu verelim. Bu komut yardımıyla bu os adlı modülün içine bakacağız:

```
>>> dir(os)
```

Ne oldu? Python bize bir hata mesajı gösterdi, değil mi? Bu gayet normal. Çünkü biraz önce de söylediğimiz gibi, bir modülü kullanabilmek için öncelikle o modülü içe aktarmamız (import) gerekiyor. Hemen bu işlemi nasıl yapacağımıza bakalım:

```
>>> import os
```

Bu komut yardımıyla Python'a şu emri vermiş oluyoruz: "*Ey Python! 'os' adlı modülü içe aktar, ki ben bu modülün bana sunduğu nimetlerden faydalanabileyim...*"

Bu emrimizi alan Python sessizce alt satıra geçecektir. Buraya kadar her şey normal. Bu yaptığımız işleme "içe aktarma" adı veriliyor. Yani biz bu işlemle, os adlı modülü içe aktarmış olduk. Artık bu modülün bütün özelliklerinden yararlanabiliriz. Öncelikle bu modülün içine bakalım neler varmış:

```
>>> dir(os)

['F_OK', 'MutableMapping', 'O_APPEND', 'O_BINARY', 'O_CREAT', 'O_EXCL', 'O_NOINH
ERIT', 'O_RANDOM', 'O_RDONLY', 'O_RDWR', 'O_SEQUENTIAL', 'O_SHORT_LIVED', 'O_TEM
PORARY', 'O_TEXT', 'O_TRUNC', 'O_WRONLY', 'P_DETACH', 'P_NOWAIT', 'P_NOWAITO', '
P_OVERLAY', 'P_WAIT', 'R_OK', 'SEEK_CUR', 'SEEK_END', 'SEEK_SET', 'TMP_MAX', 'W_
OK', 'X_OK', '_Environ', '__all__', '__builtins__', '__doc__', '__file__', '__na
me__', '__package__', '_copyreg', '_execvpe', '_exists', '_exit', '_get_exports_
list', '_keymap', '_make_stat_result', '_make_statvfs_result', '_pickle_stat_res
ult', '_pickle_statvfs_result', '_putenv', '_unsetenv', '_wrap_close', 'abort',
'access', 'altsep', 'chdir', 'chmod', 'close', 'closerange', 'curdir', 'defpath'
, 'device_encoding', 'devnull', 'dup', 'dup2', 'environ', 'errno', 'error', 'exe
cl', 'execle', 'execlp', 'execlpe', 'execv', 'execve', 'execvp', 'execvpe', 'ext
sep', 'fdopen', 'fstat', 'fsync', 'getcwd', 'getcwdb', 'getenv', 'getpid', 'isat
ty', 'linesep', 'listdir', 'lseek', 'lstat', 'makedirs', 'mkdir', 'name', 'open'
, 'pardir', 'path', 'pathsep', 'pipe', 'popen', 'putenv', 'read', 'remove', 'rem
ovedirs', 'rename', 'renames', 'rmdir', 'sep', 'spawnl', 'spawnle', 'spawnv', 's
pawnve', 'startfile', 'stat', 'stat_float_times', 'stat_result', 'statvfs_result
', 'strerror', 'sys', 'system', 'times', 'umask', 'unlink', 'urandom', 'utime',
'waitpid', 'walk', 'write']
```

Gördüğünüz gibi, os modülü içinde bir hayli araç bulunuyor. Peki bu os modülü ne işe yarar? İngilizce bilen arkadaşlarım şu komutu kullanarak bu modülün belgelendirme dizisine ulaşabilir (Belgelendirme dizilerinin ne olduğunu fonksiyonlar bölümünde işlemiştik...):

```
>>> print(os.__doc__)
```

İngilizce bilmeyen arkadaşlarım için bu modülün ne işe yaradığını biz söyleyelim:

os modülü Python içindeki en önemli modüllerden bir tanesidir. Bu modül, kullandığımız işletim sistemine ilişkin bazı önemli işlemler yapmamızı sağlar. Ayrıca bu modülden yararlanmak, birden fazla platform üzerinde çalışan programlar yazmanızı da kolaylaştıracaktır. Bu modülü ayrı bir başlık altında daha detaylı bir şekilde inceleyeceğiz. Ama şimdilik isterseniz bu modülle ilgili birkaç örnek yaparak modüller konusuna biraz olsun ısınmanızı kolaylaştırmaya çalışalım:

En başta da dediğimiz gibi, bir modülü kullanabilmek için öncelikle o modülü içe aktarmamız gerekiyor. Bunu sık sık tekrar ediyorum, çünkü Python'u öğrenenlerin en çok yaptığı hatalardan biri de bir modülü içe aktarmadan kullanmaya çalışmaktır... Şu halde hemen os modülünü içe aktaralım:

```
>>> import os
```

Eğer etkileşimli kabuk üzerinde çalışıyorsanız, bu modülü her oturumda bir kez içe aktarmak yeterli olacaktır.

Şimdi şu kodlara bakalım:

```
>>> import os

>>> if os.name == "posix":
...     print("Merhaba GNU/Linux kullanıcısı... Stallman'a benden selam söyle!")

>>> elif os.name == "nt":
...     print("Merhaba Windows kullanıcısı... Gates'e benden selam söyle!")
```

Eğer kullandığınız sistem GNU/Linux ise yukarıdaki kodları çalıştırdığınızda "Merhaba GNU/Linux kullanıcısı..." çıktısı verilecek, eğer Windows kullanıyorsanız "Merhaba Windows kullanıcısı..." çıktısı verilecektir. İsterseniz bu kodları biraz inceleyelim:

Öncelikle os modülünü içe aktarıyoruz. Çünkü bu modülü ve içindeki nitelikleri kullanabilmek için ilk yapmamız gereken şey bu modülü içe aktarmaktır. Modülü bir kez içe aktardıktan sonra "dir(os)" çıktısında görünen bütün niteliklerden, metotlardan ve fonksiyonlardan faydalanabiliriz. os modülü içinde, işletim sistemlerine ilişkin bazı işlemleri yerine getirmemizi sağlayacak araçlar bulunur. Eğer "os(dir)" çıktısına bakacak olursanız, orada name adlı bir ifade görürsünüz. "name" kelimesinin anlamı "isim"dir. Dolayısıyla bu ifade bize kullandığımız işletim sisteminin ismini söyleyecektir. os modülü içinde şu isimler tanımlanmıştır:

```
'posix', 'nt', 'mac', 'os2', 'ce', 'java', 'riscos'
```

Şimdi etkileşimli kabukta os modülünü içe aktardıktan sonra şu komutu verin:

```
>>> print(os.name)
```

Bu komut, kullandığınız işletim sistemine göre farklı çıktılar verecektir. Eğer GNU/Linux dağıtımlarından birini kullanıyorsanız yukarıdaki komuttan alacağınız çıktı "posix" olacaktır. Eğer Windows kullanıyorsanız bu komut size "nt" cevabını verir. Yok eğer siz bir Macintosh kullanıcısıysanız bu komuttan alacağınız cevap "mac" olacaktır...

Bu ifadeler arasında görünen "os2", OS/2 adlı işletim sistemini, "ce" Windows CE işletim sistemini, "java" JavaOS işletim sistemini, "riscos" ise RISC OS işletim sistemini temsil eder...

Burada os modülü içindeki bir niteliği nasıl kullandığımıza dikkat edin: os + "." + name. Burada "modül_adı" + "nokta" + "istenen_nitelik" yolunu izliyoruz.

Biz burada "print(os.name)" komutuyla aslında Python'a şöyle bir emir vermiş oluyoruz: "*os adlı modülünün "name" adlı niteliği aracılığıyla bana kullandığım işletim sisteminin adını söyle!*"

Gelin isterseniz yine os modülü içinde yer alan araçlardan biriyle ilgili bir örnek daha yapalım:

```
>>> print(os.getcwd())
```

Bu komut bize, o anda hangi dizin içinde çalışıyorsak o dizinin adını verecektir. getcwd() adlı fonksiyonu da "dir(os)" çıktısı içinde görebilirsiniz. Bu arada, getcwd'nin bir fonksiyon olduğuna dikkat edin. Bu araç bir fonksiyon olduğu için bunu parantezlerle kullanmayı unutmuyoruz. Biraz önce gördüğümüz name niteliği ise bir fonksiyon değil, os modülü içinde yer alan değişkenlerden biridir.

Burada da "print(os.getcwd())" komutuyla Python'a şu emri veriyoruz: "*os adlı modülün "getcwd()" adlı niteliği aracılığıyla bana şu anda hangi dizinde çalıştığımı göster.*"

Bu arada, eğer os modülünün içeriğini merak ediyorsanız, bu dosyayı inceleyebilirsiniz. Bu modülün sisteminizde nerede bulunduğunu öğrenmek için şöyle bir komuttan yararlanabilirsiniz:

```
>>> os.__file__
```

Bu komut, size os modülünün hangi klasör içinde olduğunu söyleyecektir...

Dediğimiz gibi, ilerleyen sayfalarda bu os modülünü daha detaylı olarak inceleyeceğiz. Burada amacımız sadece modül konusuna sizleri ısındırmak...

Dilerseniz şimdi baştan beri söz ettiğimiz "içe aktarma" mevzuunu biraz deşelim...

Dedik ki, bir modülü herhangi bir şekilde kullanabilmek için öncelikle o modülü içe aktarmamız gerekiyor. Yabancılar bu kavrama "import" diyorlar... Python'da bir modülü birkaç farklı şekilde içe aktarabiliyoruz:

1. import *modül*
2. from *modül* import *nitelik_adı*
3. from *modül* import *
4. import *modül* as *takma_ad*

Peki bu biçimler arasındaki farklar nedir? Tek tek bakalım:

```
import modül
```

Python'da bir modülü bu şekilde içe aktardığımız zaman o modülün ismi dahil bütün nitelikleri kullanılabilir duruma gelir. Bir örnek verelim:

```
>>> import os
>>> print(os.name)
```

Burada os adlı modülü "import os" biçimini kullanarak içe aktardığımız için, bu modül içinde bulunan niteliklerden biri olan name'yi kullanabiliyoruz. os modülü içindeki nitelikleri kullanabilmek için "os.nitelik" yapısını kullanmamız gerekiyor.

Bir modülü bu şekilde içe aktardığımızda, o modülün kendi ismi de içe aktarılmış olur. Bu ne demek? Şu örneğe bakalım:

```
>>> dir(os)
```

Modülümüzü "import os" diyerek içe aktardığımız için "dir(os)" komutu çalışıyor. Bunun sebebi os isminin de içe aktarılmış olmasıdır. Bunu biraz sonra daha iyi anlayacağız.

11.1.1 from modül import nitelik_adı

Modüllerimizi "from *modül* import *nitelik_adı*" biçimini kullanarak da içe aktarabiliyoruz. Bir modülü bu şekilde içe aktardığımız zaman ortaya bazı kullanım farkları çıkacaktır. Şu örneğe bakalım:

```
>>> from os import getcwd
```

Burada yaptığımız şey, os modülünün getcwd adlı niteliğini içe aktarmaktan ibarettir. Burada modülümüzün tek bir niteliğini içe aktarmış olduk. Ayrıca kullanımda da şöyle bir fark ortaya çıktı:

```
>>> print(getcwd())

'/home/istihza/Desktop/'
```

Gördüğünüz gibi, bu defa getcwd()'nin başına "os." ifadesini koymadık. Artık "os.getcwd()" demek yerine sadece "getcwd()" komutunu kullanabiliyoruz. Ancak modülümüzü bu biçimde içe aktardığımızda sadece belirtilen nitelik kullanılabilir durumda olacaktır. Yani bu durumda os modülünün `getcwd()` niteliği dışındaki hiç bir niteliğini kullanamıyoruz. Mesela burada `name` niteliğinden yararlanamayacağız:

```
>>> print(name)

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

Eğer `getcwd()` ile birlikte `name` niteliğini de kullanmak istersek kodumuzu şöyle yazmamız gerekir:

```
>>> from os import getcwd, name
```

Artık şu komutlar çalışacaktır:

```
>>> print(getcwd())

>>> print(name)
```

Gördüğünüz gibi, kullanmak istediğimiz her niteliği ayrı ayrı içe aktarıyoruz. Eğer yazacağınız programda bir modülün sadece tek bir niteliğini kullanacaksanız koca bir modülün tamamını içe aktarmak yerine, bu biçimi tercih edebilirsiniz...

Biraz önce, "*bir modülü bu import modül_adı biçimiyle içe aktardığımızda, o modülün kendi ismi de içe aktarılmış olur,*" demiştik. Şimdi bunun ne demek olduğunu göreceğiz:

```
>>> dir(os)

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

Gördüğünüz gibi, modülümüzü from *modül* import *modül_adı* biçiminde içe aktardığımızda os adını kullanamıyoruz. Çünkü, dediğimiz gibi, bu şekilde içe aktarmalarda modül adı aktarım listesine dahil olmayacaktır.

11.1.2 from modül import *

Modülümüzü içe aktarmak için kullanabileceğimiz bir başka yol da "from *modül* import * biçimini tercih etmektir. Eğer bir modülü bu şekilde içe aktaracak olursak, o modül içindeki, ismi "_" ile başlayanlar hariç bütün nitelikler kullanılabilir hale gelecektir. Örnek verelim:

```
>>> from os import *

>>> print(name)

>>> print(getcwd())
```

Gördüğünüz gibi, "from os import *" yapısını kullandığımızda, modül ismini kullanmadan (yani "os." kısmını atarak), modüle ait nitelikleri kullanabiliyoruz. Ancak bu yapının çok önemli bir riski vardır. Bir modülü bu şekilde içe aktardığımızda, o modüle ait bütün nitelikler doğrudan o anda içinde çalıştığımız isim alanına dahil olacaktır. "isim alanı" kavramını önceki derslerimizden hatırlıyorsunuz. İsterseniz bu riski şöyle bir örnekle somutlaştıralım.

Diyelim ki şöyle bir kodumuz var:

```
>>> name = "istihza.com"
```

Eğer biz bu kodun bulunduğu bir programda "from os import *" komutunu verecek olursak programımızı darmadağın ederiz... Nasıl mı? Şuna bakın şimdi:

```
>>> name = "istihza.com"
>>> print(name)

istihza.com
```

Şimdi os modülünü içe aktarıyoruz:

```
>>> from os import *
>>> print(name)

posix
```

Gördüğünüz gibi, os modülünü içe aktarmamızla birlikte önceden tanımlamış olduğumuz name adlı değişken siliniverdi... Muhtemelen yazacağınız programlarda böyle bir şeyin başınıza gelmesini istemezsiniz... Bu tür durumları engellemenin en iyi yolu modülü "import modül_adı" şeklinde içe aktarmaktır:

```
>>> name = "istihza.com"
>>> print(name)

istihza.com

>>> import os
>>> print(os.name)

posix

>>> print(name)

istihza.com
```

Bu defa, os modülüne ait name niteliğini kullanabilmek için "os." ifadesini de eklememiz gerektiğinden, daha önce tanımladığımız "name" adlı değişkeni kaybetmemiş oluyoruz...

11.1.3 import modül as takma_ad

Python'da bazı modül isimleri uzundur. Örneğin `multiprocessing` adlı modülü şu şekilde içe aktardığımızı varsayalım:

```
>>> import multiprocessing
```

Bu modülü bu şekilde içe aktardığımızda modülün niteliklerine erişmek için her defasında modül adını da belirtmemiz gerekecek:

```
>>> multiprocessing.nitelik_adı
```

Eğer her defasında "multiprocessing" yazmak zor geliyorsa modülü şu şekilde içe aktarmayı tercih edebiliriz:

```
>>> import multiprocessing as mp
```

Böylece normalde “multiprocessing” yazmamız gereken yerlere sadece “mp” yazabileceğiz:

```
mp.nitelik_adı
```

Bu yöntemi ayrıca modül adının sizin programınızdaki kodlarla çakışmaya neden olacağını düşündüğünüz durumlarda da kullanmayı tercih edebilirsiniz.

Gördüğünüz gibi Python’da bir modülü içe aktarmanın pek çok yolu var. Peki bunlardan hangisini tercih etmeliyiz? Bunlar içinde en sağlıklı olanı “import *modül_adı*”, en pratik olanı ise “from *modül_adı* import *” şeklinde gösterdiğimiz içe aktarma biçimidir. Kullandığımız modülün “from *modül_adı* import *” şeklinde içe aktarılmasının sakıncalı olmadığından emin değilsek “import *modül_adı*” biçimini tercih etmeliyiz. Yani çoğunlukla tercih edeceğimiz biçim “import *modül_adı*” olacaktır... Bazı modülleri “from modül_adı import *” şeklinde içe aktarmamızda sakınca yoktur. Mesela grafik arayüzlü programlar yapmak için kullanacağımız `tkinter` adlı modül bunlardan biridir...

11.2 os ve sys Modülleri

Bu bölümde Python içindeki modüllerden en önemli iki tanesi olan `os` ve `sys` adlı modülleri inceleyeceğiz. Bu iki modül Python’un en temel modülleridir ve mutlaka öğrenilmeleri gerekir.

O halde ilk modülümüz olan `os`‘tan başlayalım...

11.2.1 os Modülü

`os` Modülü Python’un en temel modüllerinden bir tanesidir. Bu modüle daha önce değinmiştik. Burada bu modülü, biraz daha ayrıntısına inerek incelemeye çalışacağız.

Bildiğiniz gibi, bir modülü kullanabilmek için öncelikle o modülü içe aktarmamız gerekiyor:

```
>>> import os
```

Yine bildiğiniz gibi, modüllerimizi birkaç farklı şekilde içe aktarabiliyoruz. Ama mevcut yöntemler içinde en sağlıklısı “import os” biçimidir.

İçe aktardığımız bu modülün içeriğini görmek istersek, daha önce pek çok kez kullandığımız `dir()` fonksiyonundan yararlanabiliriz:

```
>>> dir(os)
```

`dir()` fonksiyonuyla `os` modülünün içine baktığımızda bu modülün epey bir niteliği olduğunu görüyoruz...

“os”, İngilizce’deki “operating system” ifadesinin kısaltmasıdır. Bu ifade “işletim sistemi” anlamına gelir. Dolayısıyla `os` modülü bize işletim sistemleriyle ilgili işlemler yapma imkanı verir. Eğer bu modülle ilgili yardım almak isterseniz şu komutları kullanabilirsiniz:

```
>>> help(os)
```

...veya...

```
>>> print(os.__doc__)
```

Eğer İngilizce biliyorsanız, yukarıdaki komutların çıktılarına bakarak `os` modülünün ne işe yaradığına dair bazı bilgiler edinebilirsiniz. Eğer İngilizce bilmiyorsanız ziyanı yok. Biz size yardımcı olmaya çalışacağız.

Gelin isterseniz şimdi “dir(os)” komutu ile karşımıza gelen listedeki niteliklerin en önemlilerini incelemeye koyulalım.

name Niteliği

“name” kelime olarak “isim” anlamına gelir. os modülüne ait bir nitelik olan name, bize kullandığımız işletim sisteminin adını verir. Bununla ilgili şöyle bir örnek verebiliriz:

```
>>> print(os.name)
```

Kullandığımız işletim sistemine göre bu komuttan şu çıktılardan birini alırız:

```
'posix', 'nt', 'mac', 'os2', 'ce', 'java', 'riscos'
```

Bunların ne anlama geldiğini daha önce söylemiştik.

os modülünün name niteliğini kullanarak, yazdığınız kodların birkaç sisteme birden duyarlı olmasını sağlayabilirsiniz. Mesela şöyle bir örnek verelim:

```
>>> if os.name == "posix": #Eğer işletim sistemi GNU/Linux ise...
...     print("Sisteme hoşgeldiniz!")

>>> elif os.name == "nt": #Eğer işletim sistemi Windows ise...
...     print("Kullandığınız işletim sistemi desteklenmemektedir.")
```

Burada kullanıcının sahip olduğu işletim sistemine göre kullanıcıyı kabul ediyoruz veya etmiyoruz... Bu arada, yukarıdaki kodları çalıştırabilmek için öncelikle os modülünü içe aktarmış olmanız gerektiğini biliyorsunuz...

os modülünün name niteliği üzerine söylenecek fazla bir şey yok. Ne işe yaradığı zaten adından da belli oluyor. O yüzden isterseniz hemen başka bir niteliği tanımaya girişelim.

getcwd Fonksiyonu

getcwd os modülü içinde bulunan bir fonksiyondur. Biz bu fonksiyona daha önce de değinmiştik. Orada da dediğimiz gibi, bu fonksiyon o anda bulunduğumuz dizinin adını verir:

```
>>> print(os.getcwd())

'/home/istihza'
```

Demek ki ben şu anda “/home/istihza” dizininin altında çalışıyormuşum. Elbette işletim sisteminize ve o anda bulunduğunuz dizine bağlı olarak bu komutun çıktısı sizde farklı olacaktır. Örneğin bu fonksiyonu Windows altında çalıştırmak istersek...

```
>>> print("Şu anda bulunduğunuz dizinin adı:\n", os.getcwd())

'Şu anda bulunduğunuz dizinin adı:
C:\\Documents and Settings\\ISTIHZA'
```

os modülünün bu getcwd() adlı fonksiyonu, yazdığımız pek çok programda işimizi kolaylaştırabilecek bir araçtır.

listdir Fonksiyonu

"listdir" kelimesi "list directory" (dizini listele) ifadesinin kısaltmasıdır. Bu fonksiyonu kullanarak, istediğimiz bir dizinin içeriğini gösterebiliriz:

```
>>> os.listdir("/home/istihza/")
```

Burada ben "/home/istihza" dizininin içeriğini listeledim. Eğer Windows üzerindeyseniz şöyle bir şey de yapabilirsiniz tabii ki:

```
>>> os.listdir("C:\\Documents and Settings")
```

Bu komut "C:\Documents and Settings" altında bulunan bütün dosyaları bir liste halinde verecektir. Eğer o anda içinde bulunduğunuz dizinin içeriğini listelemek isterseniz şöyle bir şey yazabilirsiniz:

```
>>> mevcut_dizin = os.getcwd()
>>> os.listdir(mevcut_dizin)
```

Bu komut da o anda bulunduğunuz dizin içinde ne var ne yoksa listeleyecektir.

Eğer mesela bir dizin içindeki, "f" harfiyle başlayan dosyaları listelemek gibi bir niyetiniz varsa şöyle bir kod yazabilirsiniz:

```
>>> dizin = "/home/istihza/Desktop"

>>> for i in os.listdir(dizin):
...     if i[0] == "f":
...         print(i)
```

`print()` fonksiyonunun gücünü kullanarak şöyle bir çıktı da elde edebilirsiniz:

```
>>> mevcut_dizin = os.getcwd()
>>> print(*os.listdir(mevcut_dizin), sep=", ")
```

Böylece hem mevcut dizindeki dosyaları listeledik, hem de her bir öğeyi virgülle ayırdık...

chdir Fonksiyonu

"chdir" kelimesi, "change directory" (dizin değiştir) ifadesinin kısaltmasıdır. Adından da anlaşılacağı gibi bu fonksiyon o anda içinde bulunduğumuz dizinden çıkıp başka bir dizine geçmemizi sağlar. Mesela:

```
>>> os.chdir("/usr/share/")
```

Yukarıdaki komut yardımıyla, o anda bulunduğum dizin içinden çıkıp "/usr/share/" adlı dizinin içine girdim. Bu durumu şu şekilde teyit edebiliriz:

```
>>> mev_diz = os.getcwd()
>>> os.listdir(mev_diz)
```

Şu anda "/usr/share/" dizini altında bulunduğum için yukarıdaki kodların çıktısında bu dizinin içeriğini görüyorum. Şimdi bulunduğum dizini tekrar değiştireyim:

```
>>> os.chdir("/etc/X11/")
>>> os.listdir(os.getcwd())
```

O anda içinde bulunduğum dizini terkedip "/etc/X11/" dizinine geldiğim için, os.listdir(os.getcwd()) komutu bana "/etc/X11/" dizininin içeriğini listeliyor.

curdir Niteliği

"curdir" İngilizce'de "current directory" (mevcut dizin) ifadesinin kısaltmasıdır. Bu nitelik daha önce öğrendiğimiz `getcwd()` fonksiyonuna çok benzer. Bu ikisi arasında anlayış farkı vardır...

Öncelikle `curdir` niteliğinin ne çıktı verdiğine bakalım:

```
>>> os.curdir

'.'
```

Gördüğünüz gibi, bu komuttan bir "nokta" çıktısı alıyoruz... Peki bu ne işe yarıyor? İşletim sistemlerinde mevcut dizini ifade etmek için kullanılan basit karakter dizileri vardır. İşte bu nokta işareti de bunlardan biridir. Mesela bu karakter dizisini şu şekilde kullanabiliriz:

```
>>> os.listdir(".")
```

Bu komut bize "os.listdir(os.getcwd())" ile aynı çıktıyı verecektir. Genellikle bütün işletim sistemlerinde mevcut dizini temsil eden karakter dizisi nokta işaretidir. Ama programınızın birden fazla işletim sistemi üzerinde çalışmasını temin etmek için şöyle bir kod yazmayı tercih edebilirsiniz:

```
>>> os.listdir(os.curdir)
```

Böylece, programınızın çalıştırıldığı işletim sistemi neyse, "os.curdir" yerine otomatik olarak mevcut dizini temsil eden karakter dizisi yerleştirilecek ve böylece mevcut dizinin içeriği otomatik olarak ekrana dökülecektir...

environ Niteliği

`os` modülü içinde bulunan bu `environ` adlı nitelik esasında bir sözlüktür. Bu sözlük sisteminizdeki çevre değişkenlerine ilişkin bilgileri tutar. Mesela `environ` niteliğini kullanarak sistemdeki kullanıcı adını öğrenebiliriz:

```
>>> os.environ["USER"] #Windows'ta "USERNAME"
```

Kullanıcı dosyalarının bulunduğu dizini elde etmek için ise şöyle bir şey yazabiliriz:

```
>>> os.environ["HOME"] #Windows'ta "HOMEPATH"
```

Bilgisayar adı:

```
>>> os.environ["COMPUTERNAME"] #GNU/Linux'ta buna yakın olarak "LOGNAME" bulunur...
```

İşlemciye ilişkin bilgiler:

```
>>> os.environ["PROCESSOR_IDENTIFIER"] #GNU/Linux'ta bulunmaz.
```

Dediğimiz gibi, `environ` esasında bir sözlüktür. Dolayısıyla bu sözlüğün içindeki bütün değişkenleri görmek için şöyle bir yol izleyebilirsiniz:

```
>>> for k, v in os.environ.items():
...     print("{0} = {1}".format(k, v)
```

Mesela kullanıcının masaüstü dizinine gitmek, ardından da masaüstündeki dosyaları listelemek için `environ`'dan yararlanabilirsiniz:

```
>>> os.chdir(os.environ["HOME"] + "/Desktop")

>>> os.listdir(os.curdir)
```

sep Niteliği

İşletim sistemleri, dizinleri birbirinden ayırmak için farklı işaretler kullanır. Örneğin `os.getcwd()` komutunu verdiğimizde, dizin ayraçları açısından GNU/Linux ve Windows'ta farklı sonuçlar alırız:

```
>>> os.getcwd()

/home/istihza/Desktop #GNU/Linux

C:\\Documents and Settings\\ISTIHZA\\Desktop #Windows
```

Gördüğünüz gibi, dizin ayraçları birbirinden farklı. Eğer yazdığımız programlarda bu farklılığı göz önünde bulundurmazsak, programlarımız farklı işletim sistemlerinde düzgün çalışmayacaktır. Python bu riske karşı bize os modülü içinde sep adlı bir nitelik sunar. Bu komut farklı sistemlerde farklı çıktılar verecektir:

```
>>> os.sep

'/' #GNU/Linux

'\\' #Windows
```

GNU/Linux ve Windows'taki dizin ayraçlarının birbirlerinden farklı olduğuna dikkat edin. Bu sep niteliği sayesinde birden fazla işletim sisteminde çalışabilecek programlar yazma ihtimalimiz artar. Bir örnek verelim:

```
>>> dizin = os.getcwd() + "\\resimler"
```

Bu yapı sadece Windows için uygundur. Çünkü biz burada kodumuzun içine doğrudan "\\" ayracını koyarak programımızın GNU/Linux'ta çalışmasını engelledik. Bu şekilde dizin ayraçlarını kodun içine gömmek yerine `os.sep`'ten faydalanabiliriz:

```
>>> dizin = os.getcwd() + os.sep + "resimler"
```

Bu şekilde Python uygun dizin ayracını otomatik olarak yerine yerleştirecektir.

mkdir ve makedirs Fonksiyonları

Bu bölümde, os modülü içinde yer alan iki önemli fonksiyondan söz edeceğiz. Bunlar `mkdir()` ve `makedirs()` fonksiyonlarıdır. İlk olarak `mkdir()` fonksiyonuyla başlayalım.

mkdir() Fonksiyonu

os modülü içinde yer alan `mkdir` adlı fonksiyon yardımıyla bilgisayarımızda yeni dizinler oluşturabiliyoruz. Örneğin:

```
>>> os.mkdir("DENEME_DİZİNİ")
```

Böylece o anda çalıştığımız dizin içinde "DENEME_DİZİNİ" adlı yeni bir dizin oluşturmuş olduk.

`mkdir()` fonksiyonu sadece tek bir dizin oluşturmamıza izin verir. Yani bu fonksiyonu kullanarak birden fazla veya iç içe geçmiş dizinler oluşturamayız. Yani şöyle bir şey yapamayız:

```
>>> os.mkdir("deneme/test")
```

Ancak şöyle bir şey yapabiliriz:

```
>>> os.mkdir("deneme")
>>> os.mkdir("deneme/test")
```

Yani bu fonksiyonu kullanarak bir dizin oluşturduktan sonra o dizin içinde başka dizinler de oluşturabiliriz. Ayrıca bu fonksiyon yardımıyla aynı anda birden fazla dizin oluşturmak istersek elbette `for` döngülerinden de yararlanabiliriz:

```
>>> dizin_listesi = ["Programlar", "Belgeler", "Resimler"]
>>> for i in dizin_listesi:
...     os.mkdir(i)
```

`mkdir()` fonksiyonu GNU/Linux'ta `mode` adlı bir parametre daha alır. Bu fonksiyonu kullanarak dizin oluştururken `mode` adlı parametreye "0755", "0644" gibi sayılar vererek, oluşturduğunuz klasörün izinlerini belirleyebilirsiniz:

Örneğin şu komut yardımıyla GNU/Linux'ta oluşturduğunuz klasörün izinlerini "0755" olarak ayarlayabilirsiniz:

```
>>> os.mkdir("dizin", 0755)
```

Son olarak, eğer `os.mkdir()` fonksiyonunu kullanarak, halihazırda var olan bir dizin yaratmaya çalışırsanız, Python size bir hata mesajı gösterecektir.

Gelelim `makedirs()` fonksiyonuna...

makedirs() Fonksiyonu

`makedirs()` fonksiyonu bize iç içe geçmiş dizinler oluşturma imkanı sağlar:

```
>>> os.makedirs("dizin1/dizin2")
```

Bu komut "dizin1" adlı bir dizin, bunun içinde de "dizin2" adlı başka bir dizin oluşturacaktır. Eğer oluşturmaya çalıştığınız dizinler zaten varsa Python bir hata mesajı gösterecektir. Tıpkı `mkdir()` fonksiyonunda olduğu gibi `makedirs()` fonksiyonu da kullanılıp kullanılmaması tercihinize kalmış bir `mode` parametresi sunar:

```
>>> os.makedirs("falanca/filanca", 0644)
```

rmdir() ve removedirs() Fonksiyonları

Bir önceki bölümde `os` modülünü kullanarak nasıl dizin oluşturacağımızı öğrenmiştik. Bu bölümde ise oluşturduğumuz bu dizinleri silmeyi öğreneceğiz. Bunun için `os` modülünde yer alan iki adet fonksiyondan yararlanacağız. Bu fonksiyonlardan birinin adı `rmdir()` öbürü ise `removedirs()`. Öncelikle `rmdir()` fonksiyonundan başlayalım:

rmdir() Fonksiyonu

Bu fonksiyon yardımıyla tek bir dizini silebiliyoruz:

```
>>> os.rmdir("dizin")
```

Ancak eğer silmek istediğimiz dizinin içi doluysa Python bize bir hata mesajı gösterecektir. Dolayısıyla bu fonksiyonu kullanarak içi dolu dizinleri silemeyiz. Ama diyelim ki elimizde içi boş, ama iç içe geçmiş dizinler varsa bu fonksiyon yardımıyla bunları silebiliriz.

Diyelim ki sistemimizde "dizin1" adlı bir dizin, bunun içinde de "dizin2" adlı boş bir dizin daha var. Biz `rmdir()` fonksiyonunu kullanarak "dizin1"i silemeyiz. Çünkü "dizin1" boş değildir ("dizin1"in içinde "dizin2" var). Ama istersek "dizin2"yi silebiliriz. Çünkü bu dizinin için boş:

```
>>> os.rmdir("dizin1/dizin2")
```

Bu komut yalnızca "dizin2"yi silecek, "dizin1"e ise dokunmayacaktır. Ama eğer "dizin2"yi sildikten sonra "dizin1"in içinde başka hiç bir dosya veya klasör kalmamış, yani "dizin1"in içi boşalmışsa, bu komutu tekrar yazarak "dizin1"i de silebiliriz:

```
>>> os.rmdir("dizin1")
```

Gördüğünüz gibi, `rmdir()` fonksiyonu tek bir boş dizin üzerinde işlem yapabiliyor. Elbette bu fonksiyonu bir `for` döngüsüne sokarak birden fazla boş dizini de silebilirsiniz...

Sıra geldi `removedirs()` adlı fonksiyona...

removedirs() Fonksiyonu

Bu fonksiyon genellikle yanlış anlaşılır. Bir defa şunu bilmeliyiz ki bu fonksiyon içi dolu dizinleri silmek için kullanılmaz. Bu fonksiyon birden fazla dizini silmek için de kullanılmaz... Peki bu `removedirs()` denen fonksiyon ne işe yarar?

Diyelim ki sistemimizde "dizin1" adlı bir dizin, bunun içinde "dizin2" adlı başka bir dizin, bunun içinde de "dizin3" adlı başka bir dizin var. Şimdi şöyle bir komut verelim:

```
>>> os.removedirs("dizin1/dizin2/dizin3")
```

Bu komutu alan Python, sondan başa doğru ilerleyecek ve yoluna çıkan bütün boş dizinleri silecektir. Ama Python bu işlem sırasında karşısına çıkan ilk içi dolu dizinde silme işlemini keser... Yani eğer bu üç dizinin üçünün de içi boşsa `removedirs()` fonksiyonu bu üç dizinin üçünü de siler. Eğer "dizin1"in içinde "dizin2" harici başka dosyalar da varsa, `removedirs()` fonksiyonu "dizin2" ve "dizin3"ü siler. Eğer "dizin2"nin içinde başka dosyalar varsa, bu fonksiyon sadece "dizin3"ü siler. Eğer "dizin3"ün içinde başka dosyalar var ise hiç bir şey silmez... Eğer bu dediklerim biraz karışık geldiyse, kendi kendinize bazı denemeler yaparak `removedirs()` fonksiyonunun tam olarak ne iş yaptığını daha net olarak görebilirsiniz.

Peki `rmdir()` ve `removedirs()` fonksiyonlarından hiçbiri içi dolu dizinleri silemiyorsa, biz içi dolu dosyaları silmek istersek ne yapacağız? Hiç endişelenmeyin. Python'da bunun için kullanılan başka araçlar vardır. Onları da bir iki bölüm sonra göreceğiz.

rename Fonksiyonu

Şimdiye kadar Python'daki `os` modülünü kullanarak dizin oluşturmayı (`mkdir`/`makedirs`) ve oluşturduğumuz bu dizinleri silmeyi (`rmdir`/`removedirs`) öğrendik. Peki ya biz oluşturduğu-

muz dizinlerin adını değiştirmek istersek ne yapacağız? İşte burada devreye `rename()` adlı fonksiyon girecek.

`rename()` fonksiyonunun formülü şudur:

```
>>> os.rename("dizinin_eski_adı", "dizinin_yeni_adı")
```

Diyelim ki sistemimizde "documents" adlı bir dizin var ve biz bu dizinin adını "belgeler" olarak değiştirmek istiyoruz. İşte burada `os.rename()` fonksiyonu işimizi görecektir:

```
>>> os.rename("documents", "belgeler")
```

Bu fonksiyon için, üzerinde işlem yapacağı dizinin boş olup olmaması farketmez. `rename()` fonksiyonu hem boş hem de dolu dizinlerin isimlerini başarıyla değiştirebilir.

system Fonksiyonu

`os` modülü içinde yer alan `system` adlı fonksiyon, oldukça faydalı bir araçtır. Ayrıca bu araç, `os` modülü içindeki fonksiyon ve nitelikler arasında en sık kullanılanlardan biridir. Bu fonksiyon yardımıyla işletim sistemimize ilişkin komutları Python içinden çalıştırabiliriz:

```
if os.name == "nt":
    os.system("'C:\Program Files\Internet Explorer\iexplore.exe' www.istihza.com")
elif os.name == "posix":
    os.system("firefox www.istihza.com")
```

Eğer kullandığımız sistem Windows ise Internet Explorer programını kullanarak "istihza.com" adresine gidiyoruz. Eğer kullandığımız sistem GNU/Linux ise aynı işlemi Firefox kullanarak yapıyoruz...

Herhalde söylememize gerek yok... `system` adlı bu fonksiyon, sistem komutlarını çalıştırdığı için, bu fonksiyonu kullanırken dikkatli olmamız gerekir...

os.path Modülü

Aslında Python'da `os.path` veya `path` adlı bir modül yoktur. Bunun yerine, GNU/Linux için `posixpath` ve Windows için `nt.path` adlı iki farklı modül bulunur. `os.path` adı bu iki modül yerine kullanılan bir "takma ad"dır. Biz kullandığımız sisteme göre doğrudan bu iki modülden birini içe aktarmakla uğraşmıyoruz. Bizim yapmamız gereken tek şey `os` modülünü içe aktarıp, `path` niteliğine ise `os.path` biçimini kullanarak erişmektir. Python kullandığımız sistemi kendisi tespit edip, `ntpath` veya `posixpath` modüllerinden birini otomatik olarak içe aktaracaktır. Şimdi etkileşimli kabukta şu komutu verelim:

```
>>> os.path.__file__
```

Bu komutun çıktısında, kullandığınız işletim sistemine bağlı olarak, `os.path` isminin hangi dosyaya bağlı olduğu ve bu dosyanın sistemdeki konumu görünecektir. Şimdi bu komutun sizi götürdüğü yere gidip `ntpath.py` veya `posixpath.py` dosyalarının içeriğini inceleyin. Bu dosyaların en başında bazı açıklamalar göreceksiniz. Mesela `posixpath.py` dosyasının en başında şu açıklama var:

> *Common operations on Posix pathnames. Instead of importing this module directly, import os and refer to this module as os.path. The "os.path" name is an alias for this module on Posix systems; on other systems (e.g. Mac, Windows), os.path provides the same operations in a manner specific to that platform, and is an alias to another*

module (e.g. macpath, ntpath). Some of this can actually be useful on non-Posix systems too, e.g. for manipulation of the pathname component of URLs.

Bu paragrafta, `os.path` modülünün, GNU/Linux için yazılmış `posixpath` modülüne atıfta bulunan bir takma ad olduğundan bahsediliyor. Buradaki açıklamaya göre, `os.path` modülü başka işletim sistemlerinde de `macpath` veya `ntpath` gibi modüllere atıfta bulunan bir takma addır aynı zamanda... Buna benzer bir açıklamayı `ntpath.py` dosyasında da bulabilirsiniz. Eğer bu açıklamaya etkileşimli kabuktan erişmek isterseniz ne yapmanız gerektiğini biliyorsunuz:

```
>>> print(os.path.__doc__)
```

Bu komutun çıktısında, kullandığınız sisteme bağlı olarak farklı açıklamalar göreceksiniz.

Bu ".py" uzantılı dosyaların bulunduğu dizinde "os.py" adlı bir dosya daha göreceksiniz. İşte bu dosya, biz etkileşimli kabukta "import os" komutunu verdiğimizde içe aktarılan modüldür... En başta modülün ne demek olduğunu tanımlarken söylediğimiz gibi, Python modülleri (çoğunlukla) .py uzantılı birer dosyadır. Bu "os.py" adlı dosyayı açıp içine dikkatlice bakın. Bu bölümde bahsettiğimiz, `makedirs`, `mkdir`, `removedirs` gibi fonksiyonları ve `name` gibi nitelikleri bu "os.py" adlı modül içinde bulup, bunların nasıl tanımlandığını inceleyebilirsiniz. Bu dosya içindeki her şeyi anlamak zorunda değilsiniz. Sadece göz ucuyla inceleseniz bile Python programlarının nasıl yazılacağı konusunda epey fikir sahibi olabilirsiniz...

"os.py" dosyası içinde şöyle bir kod parçası göreceksiniz:

```
if "posix" in _names:
    import posixpath as path

elif "nt" in _names:
    import ntpath as path
```

Bu bölümün başından beri anlattığımız şeylerin "os.py" dosyası içinde nasıl tanımlandığına dikkat edin. Yukarıdaki kod parçasında, biz etkileşimli kabukta "import os" komutu verdiğimizde, kullandığımız işletim sistemine göre hangi modülün "path" adıyla çağrılacağı belirtiliyor. Buna göre, eğer GNU/Linux kullanıyorsak `posixpath` adlı modül `path` takma adıyla çağrılıyor. Aynı şekilde Windows kullanıcıları için de `ntpath` adlı modül `path` adıyla içe aktarılıyor...

Şimdi biraz pratik yapalım:

```
>>> import os
>>> dir(os.path)
```

Burada, "os.path" adlı modülün içinde neler olduğunu görüyoruz. Kullandığınız sisteme göre "posixpath.py" veya "ntpath.py" dosyalarının içeriğini tekrar kontrol ederseniz, bu listedeki nitelik ve fonksiyonların nasıl tanımlandığını inceleyebilirsiniz...

Şimdi bu modülün içindeki önemli nitelik ve fonksiyonları kısaca inceleyelim:

os.path.abspath()

Bu fonksiyon, bize bir dosyanın tam yolunu verir:

```
>>> os.path.abspath("falanca.pdf")
```

`os.path.abspath()` fonksiyonunun görevi bir dosyanın nerede olduğunu bulmak değildir. Hatta bu fonksiyon, kendisine parametre olarak girdiğiniz dosya adının gerçek olup olmadığıyla dahi ilgilenmez... Bu fonksiyonun görevi, bir dosya adını almak ve bunu o anda içinde bulunulan dizinin sonuna yapıştırmaktır... Mesela bilgisayarınızda olmayan bir dosya

adı vermeyi deneyin bu fonksiyona... Örneğin yukarıda verdiğimiz "falanca.pdf" adını kullanın. Diyelim ki o anda içinde bulunduğunuz dizin "home/kullanıcı_adı/Desktop". Şu halde os.path.abspath("falanca.pdf") komutunun çıktısı "home/kullanıcı_adı/Desktop/falanca.pdf" olacaktır. Şu anda bulunduğunuz dizinin ne olduğunu `os.getcwd()` komutuyla öğrenebileceğinizi biliyorsunuz.

os.path.basename()

Bu fonksiyon bir yol içindeki dosya adını verir bize... Örneğin:

```
>>> os.path.basename("/home/istihza/Desktop/falanca.pdf")

falanca.pdf
```

Tıpkı `os.path.abspath()` fonksiyonunda olduğu gibi, `os.path.basename()` fonksiyonu da yol adının gerçek olup olmadığıyla ilgilenmez. Yani bu fonksiyonu hayali yol adlarıyla birlikte de kullanabilirsiniz:

```
>>> os.path.basename("/dangıl/dungul/bangıl/bungul/füzel.pfg")

füzel.pfg
```

os.path.exists

Bu fonksiyon, bir dosyanın var olup olmadığını kontrol eder. Eğer dosya varsa "True", yoksa "False" çıktısı verir:

```
>>> os.path.exists("falanca.txt")
```

Eğer o anda çalışılan dizin içinde "falanca.txt" adlı bir dosya varsa "True", yoksa "False" çıktısı alırız. Bu fonksiyonu uzak konumlardaki dosyaların varlığını kontrol etmek için de kullanabiliriz:

```
>>> os.path.exists("/home/istihza/Desktop/deneme.py")
```

os.path.getsize

Bu fonksiyon yardımıyla bir dosyanın boyutunu "bayt" cinsinden elde edebiliriz:

```
>>> os.path.getsize("falanca.odt")

27135
```

Demek ki "falanca.odt" adlı dosyanın boyutu "27135" baytmış. Yani yaklaşık 27 kilobayt...

os.path.isdir ve os.path.isfile

`isdir()` fonksiyonu bir "şey"in dizin olup olmadığını kontrol eder. Eğer sorguladığımız şey bir dizin ise bu fonksiyon True değerini, değilse False değerini verir:

```
>>> os.path.isdir("falanca")
```

`isfile()` fonksiyonu ise bir şeyin dosya olup olmadığını kontrol eder:

```
>>> os.path.isfile("falanca")
```

os.path.split

İsterseniz bu fonksiyonun ne yaptığına doğrudan bir örnekle bakalım:

```
>>> os.path.split("/usr/share/harman/harman.py")

('/usr/share/harman', 'harman.py')
```

Gördüğünüz gibi, bu fonksiyon yol ve dosya adını ayıkladıktan sonra bunları bize bir demet olarak verdi...

os.path.join

`os.path` modülü içindeki sık kullanılan fonksiyonlardan biri de `join`'dir. Bu kelime Türkçe'de "birleştirmek" anlamına gelir. `join` fonksiyonu, kendisine verdiğimiz parametreleri alıp bir dizin haline getirir. Mesela şu komut o anda içinde bulunulan dizin içindeki, "özel dosyalar" ve "resimler" adlı klasörleri birleştirip, kullanılan işletim sisteminin yapısına uygun bir yol adı oluşturacaktır:

```
>>> dizin = os.path.join(os.getcwd(), "özel dosyalar", "resimler")

'C:\\Documents and Settings\\ISTIHZA\\Desktop\\özel dosyalar\\resimler'
```

...veya...

```
'/home/istihza/Desktop/özel dosyalar/resimler'
```

11.2.2 sys Modülü

Python'daki önemli modüllerden biri de `sys` adlı modüldür. Bu modül Python sistemiyle ilgili nitelikler ve fonksiyonlar barındırır. `os` modülünün aksine, `sys` adlı bu modülün Python dilinde yazılmış bir kaynak kodu yoktur. Bu modül C programlama dili ile yazılmıştır. Dolayısıyla, sistemde "sys.py" adlı bir dosya bulunmadığı için, "sys.__file__" gibi bir komut bu modül üzerinde çalışmayacaktır.

Her zamanki gibi, bu modülün içeriğini şu şekilde görüyoruz:

```
>>> import sys
>>> dir(sys)
```

Bu çıktıdaki öğeler, kullandığınız işletim sistemine göre farklılık gösterebilir. Örneğin `winver` adlı nitelik sadece Windows'ta çalışacaktır...

Bu bölümde `sys` modülünün içindeki fonksiyon ve niteliklerden en önemli olanlarını işlemeye çalışacağız. Burada işleyeceğimiz nitelik ve fonksiyonlar şunlardır:

1. sys.argv
2. sys.executable
3. sys.exit
4. sys.getdefaultencoding

5. sys.path
6. sys.platform

O halde dilerseniz vakit kaybetmeden ilk öğemizle işe koyulalım...

argv Niteliği

Bu nitelik sys modülünün en fazla kullanılan öğelerinden bir tanesidir. argv yapı olarak aslında basit bir listeden ibarettir. Python'u normal bir şekilde başlatıp etkileşimli kabukta şu komutu verecek olursak alacağımız çıktı içi boş bir karakter dizisi içeren tek öğeli bir liste olacaktır:

```
>>> sys.argv

['']

>>> len(sys.argv)

1
```

Gördüğünüz gibi, elimizdeki şey bir listeden ibarettir... Peki bu sys.argv denen şey ne işe yarar?

sys.argv bize Python'un hangi parametrelerle başlatıldığını gösterir. Dilerseniz bunu basitçe test edelim. Şimdi bir Python dosyasına şu kodları yazalım:

```
import sys
print(sys.argv)
```

Daha sonra bu dosyayı mesela "deneme.py" adıyla kaydedelim ve komut satırına gidip bu programı çalıştıralım:

```
python3 deneme.py

['deneme.py']
```

Buradan gördüğümüze göre python3 uygulaması "deneme.py" adlı parametre ile çalıştırılmış...

Dediğimiz gibi, sys.argv bir listedir. Dolayısıyla bu listenin öğelerine rahatlıkla erişebilirsiniz:

```
print(sys.argv[0])

deneme.py
```

Peki sys.argv'nin bize verdiği bu bilgi neye yarar?

sys.argv'den elde ettiğimiz çıktıyı kullanarak programımızla ilgili bazı önemli işlevleri yerine getirebiliriz. Mesela yukarıdaki örneği şöyle çalıştırmayı deneyelim:

```
python3 deneme.py --yardım

['deneme.py', '--yardım']
```

Bu defa sys.argv çıktısı iki öğeli bir liste verdi. İşte bu listeyi kullanarak mesela şöyle bir şey yazabilirsiniz:

```
import sys

if "--yardım" in sys.argv:
    print("""
```

```
Yardım bölümüne hoşgeldiniz. Burada programın nasıl kullanılacağına
ilişkin kısa bilgiler bulacaksınız.""")
```

Eğer kullanıcımız yazdığımız programı "–yardım" parametresiyle birlikte çağırırsa, kendisine programımıza ilişkin yardım bilgileri sunuyoruz:

```
python3 deneme.py --yardım
```

Bu komut şöyle bir çıktı verecektir:

```
Yardım bölümüne hoşgeldiniz. Burada programın nasıl kullanılacağına
ilişkin kısa bilgiler bulacaksınız.
```

`sys.argv`'yi kullanarak, programınızın parametreli mi yoksa parametresiz mi çalıştırıldığını denetlemek için `sys.argv`'nin uzunluğunu kontrol edebilirsiniz:

```
import sys

if len(sys.argv) < 2:
    print("programımızı parametresiz olarak çalıştırdınız!")
else:
    print("programımızı parametreli olarak çalıştırdınız!")
```

Eğer `sys.argv` adlı listenin uzunluğu 2'den azsa program parametresiz çalıştırılmış demektir. Eğer uzunluk 2 ya da daha fazla ise kullanıcımız programımıza en az bir adet parametre eklemiş demektir...

executable, getdefaultencoding ve platform

Bu bölümde üç minik nitelikten/fonksiyondan bahsedeceğiz. Bunlar `sys.executable` ve `sys.platform" adlı nitelikler ile ''sys.getdefaultencoding()` adlı fonksiyondur. Bu üç öğe bize sistem hakkında bazı bilgiler verir. Önce `sys.executable` niteliğinin ne verdiğine bakalım:

```
>>> sys.executable
```

Bu komut bize kullandığımız sisteme bağlı olarak, çalıştırılabilir Python dosyasının adını ve konumunu verecektir. Mesela GNU/Linux'ta bu komutun çıktısı "*/usr/bin/python3*" gibi bir şey olurken, Windows'ta "*c:\python3x\python.exe*" olacaktır...

`sys.getdefaultencoding` fonksiyonu ise Python'un varsayılan olarak hangi kodlama sistemini kullandığını gösterir:

```
>>> sys.getdefaultencoding()
```

Bu komut "utf-8" çıktısı verecektir. Python'un 2.x sürümlerinde bu komutun çıktısı "ascii" idi... Python'un 3.x sürümüyle birlikte gelen "utf-8" desteği sayesinde İngiliz alfabesinde bulunmayan harflerle de işlem yapabiliyoruz...

Son olarak `sys.platform` adlı niteliğin çıktısına bakacağız:

```
>>> sys.platform
```

Bu komut Windows'ta "win32" çıktısını verirken GNU/Linux'ta "linux2" çıktısını verecektir. Bu komutun çıktısı kullanılan sistemin özelliğine göre farklılık gösterebilir...

exit Fonksiyonu

"exit" kelimesi "çıkmak" anlamına geliyor. Adından da anlaşılacağı gibi, bu fonksiyon Python programından çıkmamızı sağlar. Bunu şu şekilde kullanıyoruz:

```
sys.exit()
```

Eğer yazdığımız bir programın herhangi bir aşamasında programı terketmemiz gerekirse `sys.exit()` fonksiyonundan yararlanabiliriz:

```
import sys

sayı = "23"

while True:
    soru = input("Bir sayı tutun: ")

    if soru == sayı:
        sys.exit()
    else:
        continue
```

Burada eğer kullanıcının girdiği sayı 23'e eşitse programdan çıkılacaktır...

path Niteliği

Hatırlarsanız, önceki derslerimizde `os.path` adlı bir modülden söz etmiştik. Şimdi işleyeceğimiz `sys.path` modülünü daha önce işlediğimiz `os.path` modülüyle karıştırmamamız gerekiyor. İsim olarak birbirlerine benzeseler de bu iki öğenin görevleri birbirinden çok farklıdır.

`sys` modülünün `path` niteliği bize Python'un modülleri içe aktarma sistemiyle ilgili önemli bilgiler verir. Dilerseniz bu `path` niteliğini anlatmaya başlamadan önce biraz modül içe aktarma sisteminden söz edelim.

Python'da "import *modül_adı*" komutunu verdiğimizde, Python aradığımız modülü bulabilmek için sistem üzerinde birkaç farklı konuma bakar. Biz "import *modül_adı*" komutunu verdiğimizde Python'un ilk bakacağı yer o anda içinde bulunduğumuz dizindir. Yani Python öncelikle `os.getcwd()` ile elde edilen dizin adının içini kontrol edecektir. Eğer aradığımız modül bu dizin içinde yoksa, Python'un kendi içinde tanımlı olan özel dizinlerin içeriği kontrol edilecektir. Eğer aranan modül hiç bir yerde bulunamazsa Python bize bir hata mesajı gösterecektir. Python'un, bir modülü bulamaması durumunda verdiği hata şöyledir:

```
import falanca

ImportError: No module named falanca
```

Daha önce de sözünü ettiğimiz gibi, Python'un kendi içinde bulunan modüller haricinde bir de internet üzerinden veya başka kaynaklardan elde edilen modüller vardır. O modülleri kullanabilmek için öncelikle ilgili modülü bilgisayarımıza kurmamız gerekir. Dolayısıyla eğer istersseniz, o modülleri kullanarak yazdığınız programlarda şöyle bir kontrol kodu yazabilirsiniz:

```
try:
    import falanca

except ImportError:
    print("falanca adlı modül bilgisayarınızda kurulu değil.")
    print("Bu modülü filanca adresinden indirebilirsiniz.")
```

Böylece harici bir modüle bağımlı olan programınız, son kullanıcı için anlamsız bir hata üretmek yerine, kullanıcıya daha makul bir mesaj gösterebilecektir...

Dedik ki, eğer aradığımız modül o anda çalıştığımız dizin içinde yoksa, Python kendi içinde tanımlı olan özel dizinlerin içeriğini kontrol edecektir... Peki bu dizinler hangileridir? İşte burada `sys.path` adlı nitelik devreye giriyor. Şimdi bu niteliğin çıktısına bakalım:

```
>>> sys.path
```

Bu komutun çıktısında görünen dizinler, o bahsettiğimiz özel dizinlerdir. Eğer o anki çalışma dizininde bulunamayan modül `sys.path` listesinden elde edilen çıktıdaki dizinlerde de yoksa, yukarıda bahsettiğimiz "ImportError" adlı hata verilecektir...

Dediğimiz gibi, `sys.path` aslında bir listeden ibarettir. Dolaysıyla listelerin bütün metotlarını destekler. Mesela listelerin append adlı metodunu kullanarak, herhangi bir dizini bu listeye ekleyebilirsiniz:

```
>>> sys.path.append("/falanca/filanca/")
```

Elbette bu değişiklik kalıcı olmayacaktır. Programınızı veya etkileşimli kabuğu kapattığınızda `sys.path` listesi özgün haline geri dönecektir.

11.3 Kendi Modüllerimizi Yazmak

Şimdiye kadar hep halihazırda yazılmış olan modülleri kullanmayı öğrendik. Bu bölümde ise modüllerimizi kendimiz yazmayı ve kullanmayı öğreneceğiz.

En başta da dediğimiz gibi, sonu ".py" uzantısı ile biten bütün dosyalar aslında bir modüldür ve başka programların içine aktarılabilme potansiyeli taşırlar. Bununla ilgili çok basit bir örnek verelim. Diyelim ki elimizde şöyle bir kod var:

```
isim = "istihza"
eposta = "kistihza[at]yahoo[nokta]com"
```

Yine diyelim ki bu kodlar "bilgi.py" adlı bir dosya içinde bulunuyor. İşte bu "bilgi.py" adlı dosya bir modüldür ve bu modülün adı da `bilgi`'dir... Dolayısıyla bu modülü başka bir program içinden çağırabiliriz.

Şimdi mesela "bilgi.py"nin olduğu klasör içinde "deneme.py" adlı başka bir dosya daha oluşturup içine şunu yazalım:

```
import bilgi
```

Geçen bölümlerde `sys.path` konusunu anlatırken, Python'un bir modülü bulmak için hangi dizinlere baktığını söylemiştik. Orada da dediğimiz gibi, Python aradığımız bir modülü bulabilmek için önce o anda çalışılan dizinin içine ardından da `sys.path` içinde görünen dizinlere bakar. Eğer aradığımız modül bu dizinlerin hiçbirinde bulunamazsa, Python bize bir hata mesajı gösterir. Dolayısıyla kendi yazdığımız modülleri pratik şekilde kullanabilmek için en makul yol, modüllerimizi ve bu modülleri çağırmak için kullandığımız ana programımızı aynı klasör içine yerleştirmektir. Yani bu "bilgi.py" adlı dosyayı ve bu modülü içe aktarmak için kullandığımız "deneme.py" adlı dosyayı aynı dizin içine koyarsak, `bilgi` adlı modülü içe aktarmak kolay olacaktır...

Yukarıda gösterdiğimiz "import bilgi" komutunu verdikten sonra "bilgi.py" içindeki bütün nitelikleri "deneme.py" adlı dosyanın içine aktarmış oluyoruz. Artık `bilgi` adlı modülün bize sunduğu bütün olanaklardan yararlanabiliriz. Mesela bilgi.py içindeki `isim` adlı niteliğe erişmek için şöyle bir şey yazabiliriz:

```
import bilgi
print(bilgi.isim)
```

Bu kodların çıktısı şu olacaktır:

```
istihza
```

Bu `bilgi` adlı modülü tıpkı daha önce öğrendiğimiz `os` ve `sys` modülleri gibi kullandığımıza dikkat edin. Mesela `os` modülü içindeki `name` adlı niteliğe erişmek için şöyle bir şey yazıyorduk:

```
import os
print(os.name)
```

Aynı şekilde `bilgi` modülü içindeki `isim` adlı niteliğe erişmek için de "bilgi.isim" komutunu veriyoruz..

Elbette modülümüzün içine fonksiyon da yazabiliriz. Mesela daha önceki derslerimizin birinde yazdığımız `çarp()` adlı fonksiyonu bu modül içine yerleştirelim:

```
isim = "istihza"
eposta = "kistihza[at]yahoo[nokta]com"

def çarp(liste):
    a = 1
    for i in liste:
        a = i * a
    print(a)
```

Artık `bilgi` adlı modülümüzün, `isim` ve `eposta` adlı niteliklerinin yanısıra, `çarp` adlı bir fonksiyonu da var... Bu fonksiyona şu şekilde erişebileceğimizi biliyorsunuz:

```
import bilgi
lis = [20, 45, 55]

bilgi.çarp(lis)
```

Burada önemli bir noktaya değinmemizde yarar var. Özellikle etkileşimli kabuk üzerinde çalışırken, içe aktarma konusunda Python'un bir özelliğine dikkat etmemiz gerekir. Python aynı oturum içinde bir modülü sadece bir kez içe aktarır. Yani biz etkileşimli kabukta "import *modül_adı*" komutunu ilk verişimizde modülümüz içe aktarılır. Bundan sonra vereceğimiz "import *modül_adı*" komutları hiç bir işe yaramaz. Bunu şöyle test edebiliriz. Diyelim ki "bilgi.py" adlı dosyamız masaüstünde kayıtlı. Şimdi etkileşimli kabuğu masaüstünün olduğu dizinde açıp şu komutları verelim:

```
>>> import bilgi
>>> dir(bilgi)

['__builtins__', '__doc__', '__file__', '__name__',
 '__package__', 'eposta', 'isim', 'çarp']
```

Gördüğünüz gibi, bizim tanımladığımız `eposta`, `isim` ve `çarp` öğeleri modülümüzün içinde görünüyor. Şimdi "bilgi.py" adlı dosyayı açıp içine şöyle bir ekleme yapalım:

```
telefon = "02121231212"
```

Yani dosyamızın son hali şöyle görünecek:

```
isim = "istihza"
eposta = "kistihza[at]yahoo[nokta]com"
```

```
telefon = "02121231212"

def çarp(liste):
    a = 1
    for i in liste:
        a = i * a
    return a
```

Şimdi etkileşimli kabukta bu modülün içeriğini tekrar kontrol edelim:

```
>>> dir(bilgi)
```

Gördüğünüz gibi, modüle yeni eklediğimiz `telefon` niteliği listede görünmüyor. Çünkü eklemeler modüle otomatik olarak kaydedilmiyor. Şimdi modülümüzü tekrar içe aktarmayı deneyelim:

```
>>> import bilgi
>>>dir(bilgi)
```

Listede yine herhangi bir değişiklik olmadığına dikkat edin. Zaten `telefon` niteliğini kullanamamamızdan da bellidir bu:

```
>>> bilgi.telefon

AttributeError: 'module' object has no attribute 'telefon'
```

Bu hata mesajı şöyle çevrilebilir: "*NitelikHatası: "module" nesnesinin "telefon" adlı bir niteliği yok*"

Halbuki biz bu modüle biraz önce `telefon` adlı bir nitelik eklemiştik...

Eğer bunun gibi durumlarda modülümüzü tekrar içe aktarmak istersek Python'daki `imp` adlı bir modülün `reload` adlı fonksiyonundan yararlanmamız gerekir:

```
>>> import imp
>>> imp.reload(bilgi)
```

Artık modülümüzün `telefon` adlı niteliğini kullanabiliriz:

```
>>> print(bilgi.telefon)

02121231212
```

Dolayısıyla modüllerimizi "tazeleme" ihtiyacı duyduğumuzda yardımımıza koşacak fonksiyon, `imp` modülünün `reload` fonksiyonudur...

11.4 if __name__ == "__main__"

Python'da, kendi yazdığınız modüller de dahil, `dir()` fonksiyonunu kullanarak hangi modülün içine bakarsanız bakın orada `__name__` adlı bir nitelik görürsünüz. "name" kelimesi Türkçe'de "isim" anlamına geliyor. Bu fonksiyon bize modülün adının ne olduğunu söylemekle yükümlüdür.

Bir modülün farklı durumlara göre iki adı bulunur: "__main__" veya modülün kendi adı... Hemen bir örnek verelim... Şimdi boş bir dosyaya şu satırı yazıp "isim.py" adıyla kaydedelim:

```
print(__name__)
```

Daha sonra başka bir boş dosyaya şu satırı yazıp “test.py” adıyla kaydedelim:

```
import isim
```

Şimdi komut satırında şu komutu verelim:

```
python3 isim.py
```

Bu komutun çıktısı “__main__” olacaktır. Şimdi de şuna bakalım:

```
python3 test.py
```

Bu komut ise “isim” çıktısını verecektir...

Buradan şu sonuçları çıkarıyoruz:

1. Eğer bir modül doğrudan çalıştırılırsa `__name__` niteliği (yani ismi) “__main__” oluyor.
2. Eğer bir modül doğrudan değil de bir başka program içinden çağrılıyorsa `__name__` niteliği modülün kendi adı oluyor.

Yukarıdaki örnekte, “python3 isim.py” komutunu vererek `isim` adlı modülü doğrudan çalıştırmış olduk. Dolayısıyla `__name__` niteliğinin değeri “__main__” oldu. “python3 test.py” komutunu verdiğimizde ise `isim` adlı modülü test.py adlı program içinden çağırmış olduk. Bu yüzden de `__name__` niteliğinin değeri modülün kendi adı olan “isim” oldu...

Peki bu özellik ne işimize yarar?

Python’da `__name__` niteliğinin değerini kontrol ederek, yazdığınız modülleri test edebilirsiniz. Mesela bir önceki bölümde verdiğimiz örneği ele alalım:

```
isim = "istihza"
eposta = "kistihza[at]yahoo[nokta]com"
telefon = "02121231212"

def carp(liste):
    a = 1
    for i in liste:
        a = i * a
    return a
```

Bu modül bu haliyle doğrudan çalıştırıldığı zaman herhangi bir çıktı vermez (dosya adının “deneme.py” olduğunu varsayıyoruz):

```
python3 deneme.py
```

Şimdi bu dosyaya şöyle bir ekleme yapalım:

```
if __name__ == "__main__":
    lis = [2, 3, 4]
    print("deneme amaçlı hesap işleminin sonucu: ", carp(lis))
```

Yani programımızın en son hali şöyle görünecek:

```
isim = "istihza"
eposta = "kistihza[at]yahoo[nokta]com"
telefon = "02121231212"

def carp(liste):
```

```
    a = 1
    for i in liste:
        a = i * a
    return a

print(__name__)

if __name__ == "__main__":
    lis = [2, 3, 4]
    print("deneme amaçlı hesap işleminin sonucu: ", carp(lis))
```

Komut satırında “python3 deneme.py” komutunu verdiğimizde şu çıktıyı elde ederiz:

```
deneme amaçlı hesap işleminin sonucu: 24
```

Böylece yazdığımız modülün çalışıp çalışmadığını ufak bir kod parçasıyla test etmiş olduk. Artık bu modülü güvenle başka programlar içinden çağırabiliriz. Mesela “test.py” adlı bir dosya oluşturup şu kodları yazmayı deneyelim:

```
import deneme

lst = [20, 10, 30]
print("test.py modülünden çağrılarak yapılan hesap işleminin sonucu: ", deneme.carp(lst))
```

Komut satırında “python3 test.py” komutunu verdiğimizde ise şu sonucu elde edeceğiz:

```
test.py modülünden çağrılarak yapılan hesap işleminin sonucu:  6000
```

Burada durumu birazcık daha somutlaştırmak için modül içine şu satırı ekleyerek arka planda neler olup bittiğini görebilirsiniz:

```
print("’__name__’ niteliğinin değeri: ", __name__)
```

Şöyle:

```
isim = "istihza"
eposta = "kistihza[at]yahoo[nokta]com"
telefon = "02121231212"

def carp(liste):
    a = 1
    for i in liste:
        a = i * a
        return a

print("’__name__’ niteliğinin değeri: ", __name__)

if __name__ == "__main__":
    lis = [2, 3, 4]
    print("deneme amaçlı hesap işleminin sonucu: ", carp(lis))
```

Burada modülü ne şekilde çalıştırdığımıza bağlı olarak `__name__` niteliğinin değerinin nasıl değiştiğine dikkat edin...

BÖLÜM 12

Python'da Dosya İşlemleri

Bu bölümde Python programlama dilini kullanarak sistemimizdeki dosyaları nasıl yönete-ceğimizi öğreneceğiz. Konumuz temel olarak, bir dosyanın nasıl açılacağı, bu dosya üzerinde nasıl işlem yapılacağı ve dosya üzerindeki işlemler bittikten sonra bu dosyanın nasıl kapatıla-cağıdır.

İleride yazacağınız programlarda sıklıkla kullanacağınız bilgileri barındırdığı için, bu bölümü dikkatlice incelemenizi tavsiye ederim... O halde en temel bilgilerle başlayalım.

12.1 Varolan bir Dosyayı Okumak Üzere Açmak

Bir önceki bölümde modülleri işlerken "os" modülü içindeki "listdir()" adlı bir fonksiyon yardımıyla bilgisayarımızdaki dizinlerin içeriğini listelemeyi öğrenmiştik. Şimdiki bölümde ise, listelediğimiz bu dizinler içindeki dosyaları açmayı öğreneceğiz.

Diyelim ki bilgisayarımızda şu adda bir dosya var:

```
liste.txt
```

Bu dosya üzerinde herhangi bir işlem yapabilmek için, ilk iş olarak bu dosyayı açmamız gerekiyor. Dosyalarımızı Python ile açabilmek için open() adlı bir fonksiyondan yararlanacağız. Gelin isterseniz bu fonksiyonu kullanarak "liste.txt" adlı dosyamızı açalım:

```
>>> open("liste.txt")
```

Burada yaptığımız şey, "liste.txt" adlı dosyayı açmaktan ibaret. Yalnız biz burada, "liste.txt" adlı dosyanın o anda çalıştığımız dizin içinde olduğunu varsaydık. Yani bu dosyanın yukarıdaki komutla açılabilmesi için şu komutun çıktısında görünüyor olması lazım:

```
>>> os.listdir(os.curdir) #ya da os.listdir(os.getcwd())
```

Eğer bu dosya yukarıdaki komutun çıktısında görünmüyorsa, open() fonksiyonu içinde bu dosyanın tam yolunu belirtmemiz gerekir:

```
>>> open("/home/istihza/Desktop/liste.txt")
```

...gibi...

Yukarıdaki komutu verdiğimizde, “liste.txt” adlı dosyayı “salt-okunur” olarak açmış oluyoruz. Yani burada yaptığımız şey dosyayı sadece okumak üzere açmaktır. Dolayısıyla, dosyayı bu şekilde açtığımızda üzerinde değişiklik yapamayız. Sadece okuyabiliriz bu dosyayı...

“liste.txt” adlı dosyayı açmamızı sağlayan bu `open()` adlı fonksiyon ek olarak bir argüman daha alır. Örnek üzerinden görelim bunu:

```
>>> open("liste.txt", "r")
```

Burada “r” harfi” “read” kelimesinin kısaltmasıdır. Bu kelime Türkçe’de “okumak” anlamına gelir. Bu harfin görevi dosyayı “salt-okunur” kipte (sadece okumak üzere) açmamızı sağlamaktır... Eğer biz `open()` fonksiyonunun bu ikinci argümanını belirtmezsek, Python varsayılan olarak “r” argümanını vermişiz gibi davranacaktır. Dolayısıyla, open(“liste.txt”, “r”) yazmak ile open(“liste.txt”) yazmak arasında hiç bir fark yoktur...

“r” harfi dışında, `open()` fonksiyonuyla birlikte kullanabileceğimiz başka “kip belirleme harfleri” de bulunur. Bunları biraz sonra göreceğiz...

Python’da `open()` fonksiyonunu kullanarak bir dosyayı açmaya çalıştığımızda eğer herhangi bir hata ortaya çıkmadan imleç bir alt satıra geçiyorsa bu iyiye işarettir ve açmaya çalıştığımız dosyanın başarıyla açıldığını gösterir. Eğer Python açmaya çalıştığımız dosyayı bulamazsa bize şöyle bir hata verecektir:

```
>>> open("falanca.txt")

IOError: [Errno 2] No such file or directory: 'falanca.txt'
```

Gördüğünüz gibi, dosya bulunamadığında “IOError” adlı bir hata alıyoruz. Yazdığımız programlarda sıklıkla bu hatayı yakalamamız gerekecektir. Zira dosyalar ile uğraştığımız programlarda, bir dosyayı açarken, öncelikle o dosyanın sistemde var olup olmadığını kontrol etmek iyi bir yöntemdir:

```
>>> try:
...     open("falanca.txt")
... except IOError:
...     print("Böyle bir dosya yok!")
```

Böylece, eğer sistemde “falanca.txt” adlı bir dosya bulunmuyorsa, kullanıcılarımıza mantıklı bir hata mesajı göstermiş oluyoruz.

Python’da bir dosyayı okumak üzere nasıl açacağımızı gördük. Ama henüz dosyamızı okumadık... Peki yukarıda gösterildiği şekilde dosyaları açtıktan sonra bunların içeriğini nasıl görüntüleyeceğiz?

Okumak üzere açtığımız dosyaların içeriğini görüntülemenin en kolay yolu basit bir `for` döngüsü kurmaktır. Bunu nasıl yapacağımıza bakalım. Önce dosyamızı açıyoruz:

```
>>> dosya = open("python.txt")
```

Bu kısmı zaten biliyoruz. Şimdi bu dosya üzerinde bir `for` döngüsü kurarak dosya içeriğini görüntüleyelim:

```
>>> for satirlar in dosya:
...     print(satirlar)
```

Bu döngü, “python.txt” adlı dosyanın bütün içeriğini ekrana dökecektir.

Böylece en basit şekilde, varolan bir dosyayı nasıl açıp okuyabileceğimizi görmüş olduk... Şimdi yolumuza dosyaların başka bir yönüyle devam edebiliriz.

12.2 Varolan Bir Dosyayı Yazmak Üzere Açmak

Bir önceki bölümde, varolan bir dosyayı "okumak" üzere nasıl açacağımızı öğrenmiştik. Bu bölümde ise varolan bir dosyayı "yazmak" üzere açacağız.

Hatırlarsanız, önceki bölümde dosyayı okumak üzere açmak için "r" adlı bir "kip değiştirme harfi"nden söz etmiştik. Bu harf, dosyayı okumak üzere açtığımızı gösteriyordu. Tıpkı buna benzer bir biçimde, varolan dosyalara yazabilmek için de bir kip değiştirme harfi kullanacağız. Bu seferki kip değiştirme harfimiz "a". Bu harf İngilizce'deki "append" (eklemek, iliştirmek) kelimesinin baş harfidir. Önceki bölümde de söylediğimiz gibi, eğer dosyayı açarken herhangi bir kip değiştirici harf kullanmazsak, Python "r" yazmışız gibi davranacaktır. Dolayısıyla, eğer amacımız bir dosyayı okumak üzere değil de yazmak üzere açmaksa, ilgili kip değiştirme harfini (yani "a"yı) mutlaka belirtmemiz gerekir:

```
>>> dosya = open("falanca.txt", "a")
```

Eğer bilgisayarınızda "falanca.txt" adlı bir dosya varsa, bu komut varolan dosyayı açacaktır. Eğer böyle bir dosya yoksa, bu ada sahip yeni bir dosya oluşturulacaktır. Dolayısıyla bu komutu yeni bir dosya oluşturmak için de kullanabilirsiniz.

Şimdi bu "falanca.txt" adlı dosyanın içinde ne yazdığını kontrol edelim:

```
>>> for satirlar in dosya:
...     print(satirlar)
```

Ne oldu? Hata aldınız, değil mi? Gayet normal... Çünkü biz dosyamızı yazmak üzere açtık... Okumak üzere değil... "a" ve "r" kipleri arasındaki bu fark çok önemlidir. Bu ikisi arasındaki farkı unutmamamız gerekir: "r" dosyayı okumamızı, "a" ise dosyaya yazmamızı sağlar... Eğer dosyanın içindekileri görmek istiyorsanız dosyayı şöyle açmanız gerekiyor:

```
>>> open("falanca.txt", "r") #veya "r" olmadan...
```

Neyse... Ne diyorduk? Evet... Şimdi bu dosyaya, aşağıda "metin" adlı bir değişken içine yerleştirdiğimiz satırları ekleyelim:

```
metin = """Python, pek çok dile kıyasla öğrenmesi kolay bir programlama
dilidir. Bu yüzden, eğer daha önce hiç programlama deneyiminiz olmamışsa,
programlama maceranıza Python'la başlamayı tercih edebilirsiniz."""
```

Bu satırları eklemek için `write()` adlı bir fonksiyondan faydalanacağız:

```
>>> dosya.write(metin)
```

Gördüğünüz gibi, bu `write()` adlı metodu doğrudan dosyanın üzerine uyguluyoruz...

Bu aşamada önemli bir noktadan bahsetmemiz gerekiyor. Python'da dosya işlemleri yaparken temel olarak şu üç aşamayı takip etmemiz gerekir:

1. Dosyayı açmak
2. Açtığımız dosya üzerinde işlem yapmak
3. Son olarak da dosyayı kapatmak

Biz şimdiye kadar bu listedeki ilk iki işlemi yerine getirdik. Yani şimdiye kadar Python'la bir dosyanın nasıl açılacağını ve açılan bu dosya üzerinde nasıl işlem yapılacağını inceledik. Şimdi öğrenmemiz gereken şey dosyayı kapatmak olacaktır. Python'da açtığımız bir dosyayı kapatmak için `close()` adlı başka bir metottan yararlanacağız:

```
>>> dosya.close()
```

Yukarıdaki örneği basitleştirerek tekrar görelim:

```
metin = """Python, pek çok dile kıyasla öğrenmesi kolay bir programlama
dilidir. Bu yüzden, eğer daha önce hiç programlama deneyiminiz olmamışsa,
programlama maceranıza Python'la başlamayı tercih edebilirsiniz."""
```

Şimdi bu değişkeni kullanarak dosya işlemlerimizi gerçekleştirelim:

```
>>> dosya = open("falanca.txt", "a")

>>> dosya.write(metin)

>>> dosya.close()
```

Burada `open()` adlı fonksiyonu kullanarak öncelikle dosyamızı açtık. Daha sonra `write()` metodu yardımıyla dosya üzerinde bir işlem yaptık. En son da `close()` metodunu kullanarak dosyamızı kapattık. Eğer yazdığımız uygulamalarda, açtığımız bir dosyayı kapatmayı ihmal edersek, dosyamız üzerinde yaptığımız son değişiklikler dosyada görünmeyebilir. Ayrıca, kullandığımız işletim sisteminin yöntemlerini kullanarak (mesela dosyayı seçip "delete" tuşuna basarak) açık olan bir dosyayı silmeye çalışmak bizi hüsrana uğratacaktır...

Gelin isterseniz bu noktada bir nefes alıp, şimdiye kadar öğrendiğimiz şeyleri çok kısaca özetleyelim:

1. Python'da bir dosyayı açmak için `open()` adlı bir fonksiyondan yararlanıyoruz.
2. Eğer dosyamızı okumak üzere açacaksak kip değiştirme harfi olarak "r"yi kullanıyoruz.
3. Eğer niyetimiz dosyaya yazmak ise kullanmamız gereken harf "a" olacaktır.
4. Bir dosyaya ekleme yapmak için `write()` adlı bir fonksiyondan yararlanıyoruz.
5. Üzerinde işlem yaptığımız bir dosyayı kapatmak için ise `close()` fonksiyonunu kullanıyoruz.

Yukarıdaki listeyi örneklerle anlatmamız gerekirse...

1:

```
>>> dosya = open("falanca.txt", "r") #dosyayı okuma kipinde açtık
```

2:

```
>>> dosya = open("falanca.txt", "a") #dosyayı yazma kipinde açtık
```

3:

```
>>> dosya.write("Merhaba Zalim Dünya!") #dosyaya ekleme yaptık
```

4:

```
>>> dosya.close() #dosyayla işimizi bitirip dosyamızı kapattık
```

Bir dosyayı okuma kipinde açtığımızda, basit bir `for` döngüsü yardımıyla dosyanın içeriğini görüntüleyebiliriz:

```
>>> for i in dosya:
...     print(i)
```

Bir dosyayı nasıl açacağımızı, üzerinde işlem yapacağımızı ve bu dosyayı nasıl kapatacağımızı öğrendiğimize göre, "Python'da dosya işlemleri" konusunun başka bir yönünü incelemeye geçebiliriz...

12.3 Yeni bir Dosya Oluşturmak

Şimdiye kadar hep varolan dosyalar üzerinde çalıştık. Gerçi bir önceki bölümde öğrendiğimiz "a" adlı kip değiştirme harfi yardımıyla yeni bir dosya da oluşturabiliyoruz, ama bu bölümde tek görevi yeni bir dosya oluşturmak olan bir kip değiştirme harfinden söz edeceğiz... Bu harfin adı "w"dir. İngilizce "write" (yazmak) kelimesinin baş harfi olan "w", bilgisayarımızda yeni bir dosya açmamızı sağlar. Hemen bir örnek verelim:

```
>>> dosya = open("yenidosya.txt", "w")
```

Bu komut, bilgisayarımızda "yenidosya.txt" adlı bir dosya oluşturacaktır. Yalnız bu harfi çok dikkatli kullanmamız gerekiyor. Eğer bilgisayarımızda "yenidosya.txt" adlı bir dosya halihazırda mevcutsa, yukarıdaki komut bu dosyanın içinde ne var ne yoksa silecektir... Daha doğrusu, bu komut aynı adlı eski dosyayı silip aynı adda başka bir dosya oluşturacaktır. Gördüğünüz gibi, "w" harfi saç-baş yoldurma potansiyeli taşıyan bir araç. O yüzden dikkatlice kullanılması gerekiyor.

12.4 Dosya Silmek

Artık Python'da dosya açmayı, dosya oluşturmayı ve dosya kapatmayı öğrendik. Peki ya biz bu dosyaları silmek istersek ne olacak? Bunun çok basit bir cevabı var. Python'daki hemen hemen her şeyde olduğu gibi, bu iş için de özel bir fonksiyondan yararlanacağız. Aslında bu fonksiyon daha önce öğrendiğimiz os adlı modülün metotlarından bir tanesidir. Dolayısıyla bu metodu kullanabilmek için öncelikle os modülünü içe aktarmamız gerekir. Dilerseniz bununla ilgili bir örnek verelim:

```
>>> import os
>>> os.remove("/home/istihza/Desktop/falanca")
```

Yukarıdaki komut, "/home/istihza/Desktop/" dizini içindeki "falanca" adlı dosyayı uçuracaktır. Ayrıca bu komutun, dosya silerken "emin misiniz?" tarzı bir soru sormadığına dikkatinizi çekmek isterim. Dolayısıyla bu komutu vermeden önce iki kez düşünmekte fayda var...

12.5 seek() ve tell() Metotları

Hatırlarsanız, okumak üzere açtığımız bir dosyanın içinde ne olduğunu görebilmek için `for` döngülerinden faydalanabileceğimizi söylemiştik. Yani bir dosyayı şu komutlar yardımıyla okuyabiliyorduk:

```
>>> dosya = open("falanca.txt")
>>> for i in dosya:
...     print(i)
```

Dosyayı bu şekilde başarıyla okuyabildik. Şimdi bu dosyayı tekrar okumayı deneyelim:

```
>>> for i in dosya:
...     print(i)
```

Ne oldu? Boş bir çıktı aldık, değil mi? Bu gayet normal. Çünkü bir önceki adımda dosyayı bir kez sonuna kadar okuduğumuz için, aynı dosyayı ikinci kez okumaya çalıştığımızda, zaten dosyanın sonuna varmış olmamızdan ötürü, Python bize boş bir çıktı verdi... Bunu, sonuna geldiğimiz bir kaseti tekrar dinlemeye çalışmaya benzetebiliriz ("kaset" diye bir şey vardı, hatırlıyorsunuz değil mi??).

Eğer sonuna geldiğimiz bir kaseti tekrar dinlemek istersek, öncelikle bu kaseti başa sarmamız gerekir. Aynı şekilde Python'da da bir dosyayı sonuna kadar okuduktan sonra tekrar okumak istersek bu dosyayı başa sarmamız gerekecek. Bu iş için `seek()` adlı bir metottan yararlanacağız. Şimdi dosyamızı başa saralım:

```
>>> f.seek(0)
```

Böylece dosyamızı 0. bayta, yani en başa geri almış olduk. Şimdi dosyayı tekrar okuyabiliriz:

```
>>> for i in dosya:
...     print(i)
```

`seek()` metoduna sadece "0" parametresini verip dosyanın 0. baytına gitmek mecburiyetinde değiliz. Eğer biz istersek, `seek()` metodunu kullanarak dosyanın istediğimiz baytına atlayabiliriz. Bu konunun daha somut bir şekilde anlaşılabilmesi için isterseniz dosyamızın içinde şu satırların olduğunu varsayalım:

```
birinci satır
ikinci satır
üçüncü satır
```

Şimdi bu dosyanın 10. baytına gidiyoruz:

```
>>> f.seek(10)
```

Dediğim gibi, bu komut, bizi alıp dosyanın 10. baytına götürecektir. Dolayısıyla bu komuttan sonra aşağıdaki komutu verecek olursak, Python okumaya 10. bayttan başlar:

```
>>> for i in dosya:
...     print(i)
...
tır
ikinci satır
üçüncü satır
```

Çok kaba bir şekilde ifade edecek olursak, çoğunlukla "10. bayt" demek "10. karakter" demek gibidir. Mesela yukarıdaki örnekte "seek(10)" komutu bizi 10. karakterin başına götürdü... Ama bunun her zaman böyle olmayacağını da asla aklımızdan çıkarmıyoruz. Çünkü her karakter 1 bayt değerinde olmayabilir. Bazı karakterler 1'den fazla bayttan oluşur. Dolayısıyla "1 bayt = 1 karakter" şeklindeki bir düşünce bizi yanlış sonuçlara götürür.

Dediğimiz gibi, `seek()` fonksiyonunu kullanarak bir metnin herhangi bir baytına gidebiliriz. Peki o anda bir metnin kaçıncı baytında olduğumuzu nasıl öğreniriz? Tabii ki `tell()` adlı metodu kullanarak:

```
>>> dosya.tell()
```

Bu komut bize o anda metnin kaçıncı baytında olduğumuzu söyleyecektir. Henüz bir dosyayı parça parça okumamaızı sağlayacak fonksiyonları görmediğimiz için, `seek()` metodunu kullanarak bir dosyanın belirli bir baytına gitmediğimiz sürece, `tell()` komutu bize ancak bir

dosyanın 0. veya sonuncu baytında olduğumuzu söyleyebilecektir. Ama hiç merak etmeyin. Bu bahsettiğimiz fonksiyonları şimdi göreceğiz...

12.6 read(), readline(), readlines() Metotları

Python'da dosya işlemlerine ilişkin temel bazı bilgileri edindiğimize göre, bu konuya ilişkin önemli bazı fonksiyon ve metotları incelemeye geçebiliriz. Python'da dosyaları işlerken kullanabileceğimiz çok önemli üç adet metot bulunur. Bunlar `read()`, `readline()` ve `readlines()` adlı metotlardır. Bu metotlar birbirine benzese de aralarında önemli bazı farklar bulunur.

Şimdi şu komutları çalıştıralım:

```
>>> f = open("deneme.txt", "w")

>>> f.write("birinci satır\n")
>>> f.write("ikinci satır\n")
>>> f.write("üçüncü satır\n")

>>> f.close()
```

Yukarıdaki komutlar yardımıyla önce "deneme.txt" adlı bir dosya oluşturduk. Burada "w" harfini kullandığımıza dikkat edin. Eğer bilgisayarımızda "deneme.txt" adlı bir dosya yoksa bu "w" harfi sayesinde "deneme.txt" adlı bir dosya oluşturulacaktır. Eğer zaten böyle bir dosya varsa, bu dosya silinip yerine aynı adla başka bir dosya gelecektir...

Daha sonra art arda üç satır ekliyoruz "deneme.txt" adlı dosyamıza. Bunu yaparken `write()` fonksiyonundan yararlandığımıza dikkat edin. Son olarak da `close()` metodunu kullanarak dosyamızı kapatmayı ihmal etmiyoruz.

Bu arada her bir `write()` komutundan sonra etkileşimli kabukta "14", "13", "13" gibi rakamlar görmüş olmalısınız. Bu rakamlar sizin `write()` metodunu kullanarak dosyaya eklediğiniz karakter dizisinin kaç bayt olduğunu gösteriyor...

Şimdi dosyamızı okuma kipinde tekrar açalım:

```
>>> f = open("deneme.txt")
```

Burada bu komutu istersek şöyle yazabileceğimizi de biliyoruz:

```
>>> f = open("deneme.txt", "r")
```

Bildiğiniz gibi, biz "r" harfini kullansak da kullanmasak da Python o harf oraya yazılmış gibi davranacaktır...

Şimdi bu dosyamızın içeriğine bakalım:

```
>>> for i in f:
...     print(i)
...
birinci satır
ikinci satır
üçüncü satır
```

Daha önce de dediğimiz gibi, dosyaları okumak için for döngüleri basit ve pratik bir araçtır. Ama şimdi göstereceğimiz yöntem bize bu iş için daha da pratik bir yol sunar. Bu yöntemin adı `read()` metodudur... Hemen deneyelim:

```
>>> print(f.read())
```

Eğer bu komutun çıktısı boş ise, ya açmaya çalıştığınız dosya boştur, ya da `seek(0)` metodunu kullanarak dosyanın en başına sarmamışsınızdır. Eğer dosyanın boş olmadığından eminseniz şu komutu vererek dosyayı en başa sarın:

```
>>> seek(0)
```

Şimdi komutumuzu tekrar verelim:

```
>>> print(f.read())

birinci satır
ikinci satır
üçüncü satır
```

Bu read() metodunu kullanarak dosyadan istediğimiz sayıda bayt da okuyabiliriz:

```
>>> print(f.read(7))

birinci
```

Yukarıdaki komut ile, "f" adlı dosyadan 7 baytlık bir kısmı okumuş olduk... Aynı komutu tekrar verelim:

```
>>> print(f.read(7))

ikinci
```

Gördüğünüz gibi, o anda dosyanın hangi noktasında olduğumuza bağlı olarak dosyanın farklı kısımları okunuyor... O anda dosyanın kaçıncı baytında olduğumuzu öğrenmek için `tell()` adlı metottan yararlanmamız gerektiğini biliyoruz:

```
>>> f.tell()
```

Şimdiye kadar dosyalarımızı hep bir bütün olarak okuduk. Ama eğer istersek, `readline()` adlı bir metot yardımıyla dosyamızı satır satır da okuyabiliriz:

```
>>> print(f.readline())

birinci satır
```

Gördüğünüz gibi bu komutla dosyanın bütününü bir anda değil, sadece ilk satırı okuduk. Şimdi aynı komutu tekrar verelim:

```
>>> print(f.readline())

ikinci satır
```

Bu defa dosyanın ikinci satırını elde ettik. Devam edelim:

```
>>> print(f.readline())

üçüncü satır
```

Tıpkı `read()` metodunda olduğu gibi, `readline()` metoduna isterseniz parametre olarak bir sayı verip, o verdiğiniz sayı kadar bayt okuyabilirsiniz dosyadan:

```
>>> print(f.readline(7))

birinci
```

Bütün bu örneklerden anladığımız gibi, `read()` metodu bir dosyanın tamamını bir anda okuturken, `readline()` metodu aynı dosyayı satır satır okumamıza imkan tanıyor. Peki başlıkta gördüğümüz `readlines()` metodu ne işe yarıyor? Hemen görelim:

```
>>> print(f.readlines())

['birinci satır\n', 'ikinci satır\n', 'üçüncü satır\n']
```

Bu metodun çıktısı bir listedir. Bu listede, dosya içindeki bütün satırlar tek tek yer alır. Ayrıca dikkat ederseniz satır sonlarına işaret eden "n" kaçış dizisi de çıktıda belirgindir. Elde ettiğimiz çıktı bir liste olduğu için, dosyadaki satırlara tek tek erişebiliriz:

```
>>> f.seek(0) #dosyamızı başa saralım...

>>> satırlar = f.readlines() #kolay erişim için f.readlines() komutunu değişkene atadık

>>> satırlar[1] #şimdi de tek tek satırlara erişelim...

'ikinci satır\n'

>>> satırlar[2]

'üçüncü satır\n'

>>> satırlar[0]

'birinci satır\n'

>>> f.close() #eğer işimiz bittiyse dosyamızı kapatalım
```

Öteki iki metotta olduğu gibi, `readlines()` metoduna da istenirse parametre olarak bir sayı verilebilir. Ancak `readlines()` metoduna verilen bu sayı biraz farklı bir işlev görecektir. Bakalım:

```
>>> print(f.readlines(4))

['birinci satır\n']
```

Burada "4" sayısını vermemize rağmen sadece 4 bayt okunmadığına dikkat edin. Şimdi şu örneğe bakalım. Bu örnek durumu biraz aydınlatacaktır:

```
>>> print(f.readlines(15))

['birinci satır\n', 'ikinci satır\n']
```

Gördüğünüz gibi, `readlines()` metodu her koşulda satır sonuna ulaşıyor. Eğer parametre olarak verdiğimiz sayı bir satırdan az sayıda bayt içeriyorsa sadece bir satır çıktı olarak veriliyor. Eğer parametredeki sayı bir satırdaki bayt sayısını aşıyorsa, bu bayt sayısı ikinci satırın tamamını karşılamasa bile, birinci satırla birlikte ikinci satırın tamamı da ekrana dökülüyor... Bu özelliği, bir satırındaki bayt sayısını tam olarak bilmediğiniz, ama dosyanın büyüklüğünden ötürü tamamını da okumak istemediğiniz dosyalar üzerinde kullanabilirsiniz...

12.7 Dosyalarda Karakter Kodlaması (encoding)

Bazen internette dolaşırken kimi Türkçe sayfalardaki harflerin bozuk olduğunu görürüz. Yani hemen hemen herkes hayatı boyunca en az bir kez şuna benzer bir görüntüyle karşılaşmıştır:

> *Python nesne y?nelimli, yorumlanabilen, birimsel(mod?ler) ve etkile?imli bir programlama dilidir. Girintilere dayal? basit s?zdizimi, dilin Ã?renilmesini ve ak?lda kalmas?n? kolayla?t?racakt?r. Bu da ona s? diziminin ayr?nt?lar? ile vakit yitirmeden programlama yap?lmaya ba?lanabilen bir dil olma ?zelli?i kazand?r?r.*

Buradaki sorun, sayfanın karakter kodlamasının düzgün yapılmamış olmasıdır. Biz bu durumu internet tarayıcılarımızın "dil kodlaması" (veya "kodlama") ayarlarının yapıldığı bölümden uygun bir dili veya kodlamayı seçerek düzeltebiliriz. Bu ayarlarda genellikle tercih edeceğimiz kodlama "Utf-8" olacaktır. Python programlama dili de varsayılan olarak "utf-8" kodlamasını kullanır. Bu kodlama yardımıyla dünya üzerindeki pek çok dile ait özel karakterleri rahatlıkla görüntüleyebiliyoruz. Konuya işletim sistemleri çerçevesinden bakacak olursak, GNU/Linux'ta utf-8 kodlamasını kullanarak sorunsuz bir şekilde çalışabiliyoruz. Ancak Windows üzerinde bazen utf-8 kodlaması görüntüleme hatalarına yol açabilir. Windows sistemlerinde yaygın olan karakter kodlaması (Türkçe için) cp1254'tür. Bu yüzden yazdığımız bir Python programında Türkçe karakterler kullanacağımız zaman, dosyamızın karakter kodlamasını açık bir şekilde belirtmemiz gerekebilir.

Dediğim gibi, Python varsayılan olarak utf-8 kodlamasını kullanır. Bu kodlama biçimi, GNU/Linux sistemlerinde Türkçe harfleri rahatlıkla göstermemizi sağlarken, bazı Windows sistemlerinde sorun yaratabilir. Eğer yazdığınız bir programda kullandığınız Türkçe harfler düzgün görünmüyorsa veya Python bu programın çalışması sırasında hata veriyorsa, betiğinizin en başına şu satırı eklemeniz gerekebilir:

```
#-*-coding:cp1254-*-
```

Buna benzer bir durum dosyalarla çalışırken de karşımıza çıkabilir. Bazı dosyaları açmaya çalışırken şuna benzer bir hata mesajıyla karşılaşabiliriz:

```
UnicodeEncodeError: 'charmap' codec can't encode characters
in position x-y: character maps to <undefined>
```

Bu durumu engellemek için, dosya açarken mutlaka `encoding` seçeneğini de belirtmemiz gerekir:

```
>>> dosya = open("deneme.txt", "r", encoding="utf-8")
```

Gördüğünüz gibi, `open()` fonksiyonu toplam üç parametre alıyor. İlk iki parametreyi daha önce görmüştük. Üçüncü parametre olan `encoding` dosyanın karakter kodlamasını belirlememizi sağlıyor. Eğer biz açtığımız bir dosyada `encoding` parametresini belirtmezsek, Python otomatik olarak varsayılan karakter kodlamasını kullanacaktır. `encoding` parametresini belirtmeden açtığınız bir dosyanın karakter kodlamasını iki şekilde öğrenebilirsiniz:

1:

```
>>> dosya.encoding

utf-8
```

Bu komut GNU/Linux üzerinde verildiği için çıktımız "utf-8". Eğer aynı komutu Windows üzerinde verirseniz çıktınız muhtemelen "cp1254" olacaktır.

2:

```
>>> import locale
>>> locale.getpreferredencoding()

utf-8
```

Bu komut da tıpkı bir önceki komut gibi GNU/Linux üzerinde verilmiştir. Bu yüzden buradaki çıktı utf-8. Eğer bu komutu Windows'ta verirseniz çıktınız büyük ihtimalle "cp1254" olacaktır. Gördüğünüz gibi, ikinci komutta `locale` adlı modülün `getpreferredencoding()` adlı fonksiyonundan yararlandık... Eğer açtığınız bir dosyada özel olarak bir karakter kodlaması belirtmediyseniz, `encoding` niteliğinin çıktısı ile `locale.getpreferredencoding()` fonksiyonunun çıktısı aynı olacaktır.

12.8 İkili Dosyalar (Binary Files)

Şimdiye kadar hep metin dosyaları ile çalıştık. Şu ana dek öğrendiğimiz bilgilerle, içeriğinde bir metin barındıran dosyaları okuyabilir, bu dosyalara yazabilir ve bu dosyaları güvenli bir şekilde kapatabiliriz. Çok kaba bir şekilde ifade etmemiz gerekirse, eğer bir dosyayı herhangi bir metin düzenleyici ile açabiliyorsak, bu dosyalara şimdiye kadar öğrendiğimiz bilgileri uygulayabiliriz... Ancak karşımıza her zaman sadece metin içeren dosyalar çıkmayabilir. Bazen de "ikili" (binary) biçimdeki dosyalarla uğraşmamız gerekebilir. Mesela resim dosyaları birer ikili dosyadır ve bunları şimdiye kadar öğrendiğimiz bilgilerle açıp okuyamayız veya bunlara herhangi bir şey yazamayız. Python'da ikili dosyalarla çalışabilmek için özel bir kip değiştirme harfinden yararlanacağız.

Bildiğiniz gibi, şimdiye dek Python'da üç adet kip değiştirme harfi gördük. Bunlar, "r" (okuma), "a" (yazma) ve "w" (oluşturma) harfleri idi. Bu bölümde bu harflere bir yenisini daha ekleyeceğiz. Burada göreceğimiz harf "binary" kelimesinin baş harfi olan "b"dir...

Bu "b" harfini tek başına kullanmıyoruz. Bunu daha önce öğrendiğimiz o üç harfle birlikte kullanacağız. Mesela ikili düzendeki bir dosyayı okumak üzere açmak için şu komutu yazacağız:

```
>>> dosya = open("falanca.bin", "rb")
```

Hatırlarsanız, metin dosyalarını okumak üzere açarken "r" harfini belirtmesek de oluyordu. Ama ikili dosyaları okumak üzere açarken, bu dosyanın kipini mutlaka belirtmemiz gerekir.

İkili dosyaları yazmak üzere açarken ise şöyle bir komut kullanacağız:

```
>>> dosya = open("falanca.bin", "ab")
```

Gördüğünüz gibi, yaptığımız şey önceden bildiğimiz "a" harfinin sağına bir adet "b" harfi eklemekten ibarettir... Yeni bir ikili dosya oluşturabilmek için de yine benzer bir yöntem kullanıyoruz:

```
>>> dosya = open("falanca.bin", "wb")
```

Metin dosyalarını okurken karakter dizilerine bakıyoruz. İkili dosyalarda baktığımız şey ise baytlardır. Yani metin dosyaları bize karakter karakter gelirken, ikili dosyalar bayt bayt gelecektir:

```
>>> ikili = open("logo.jpeg", "rb")

>>> oku = ikili.read()

>>> len(oku)
```

```
27429
```

Demek ki benim elimdeki “logo.jpeg” dosyası 27429 bayt uzunluğundaymış. Eğer dosyanızın özelliklerine bakacak olursanız, dosyanın boyut kısmındaki rakamın `len()` fonksiyonundan aldığımız rakamla aynı olduğunu göreceksiniz.

Bu arada, eğer bu ikili dosyayı daha önce öğrendiğimiz şekilde açıp okumaya çalışırsanız Python size hata mesajı gösterecektir:

```
>>> dosya = open("logo.jpeg")

>>> oku = dosya.read()

UnicodeDecodeError: ’charmap’ codec can’t decode byte
0x8e in position 1254: character maps to <undefined>
```

Gördüğünüz gibi, Python bu şekilde ikili dosya içindeki baytları doğal olarak tanıyamıyor... Dolayısıyla ikili dosyaları açarken, kullanacağımız kip değiştirme harfine dikkat etmemiz gerekiyor.

Karakter Dizilerinin Metotları

Bu bölümde, önceki derslerde öğrenmiş olduğumuz bir konu olan "Karakter Dizileri"nden söz edeceğiz. Dediğim gibi, aslında biz karakter dizisinin ne olduğunu biliyoruz. Çok kaba bir şekilde ifade etmek gerekirse, karakter dizileri, adından da anlaşılacağı gibi, karakterlerin bir araya gelmesiyle oluşan bir dizidir. Karakter dizileri, tırnak (tek, çift veya üç tırnak) içinde gösterilen ve öteki veri tiplerinden bu tırnaklar aracılığıyla ayırt edilen özel bir veri tipidir. Mesela şu örneklerin birer karakter dizisi olduğunu biliyoruz:

```
>>> kardiz1 = "çift tırnaklı karakter dizisi"
>>> kardiz2 = 'tek tırnaklı karakter dizisi'
>>> kardiz3 = """üç tırnaklı karakter dizisi"""
```

Biz karakter dizilerini bundan önceki bölümlerde de bolca kullanmıştık. Dolayısıyla bu veri tipinin ne olduğu konusunda bir sıkıntımız yok. Bu bölümde, şimdiye kadar karakter dizileri ile ilgili öğrendiğimiz şeylere ek olarak, karakter dizilerin metotlarından söz edeceğiz. Aslında biz "*metot*" kavramının da ne olduğunu biliyoruz. Metotlar Python'da bir karakter dizisinin, bir sayının, bir listenin veya sözlüğün niteliklerini kolaylıkla değiştirmemizi veya bu veri tiplerine yeni özellikler katmamızı sağlayan küçük parçacıklardır. Örneğin listeler konusunu işlerken sözünü ettiğimiz append(), listelerin bir metodudur. Daha önce listeleri, sözlükleri, demetleri ve kümeleri işlerken nasıl bu veri tiplerinin bazı metotları olduğundan bahsettiysek, bu bölümde de karakter dizilerinin metotlarından bahsedeceğiz. Ancak bu bölümde bahsedeceğimiz tek şey karakter dizilerinin metotları olmayacak. Bu bölümde aynı zamanda karakter dizilerinin yapısı ve özelliklerine ilişkin söyleyeceklerimiz de olacak...

İlk iş olarak isterseniz karakter dizilerinin hangi metotları olduğunu görelim:

```
>>> dir("")
...
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
 '__gt__', '__hash__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
 '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__', '_formatter_field_name_split', '_formatter_parser',
 'capitalize', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
 'format', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier',
 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join',
 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind',
 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Gördüğünüz gibi karakter dizilerinin çok sayıda metodu var. Zaten daha önce de dediğimiz gibi, karakter dizileri Python programlama dilinin en güçlü olduğu veri tiplerinden bir tanesidir.

Biz bu bölümde, yukarıdaki liste içinde görünen metotlar arasında, içinde "_" işareti barındırmayanları inceleyeceğiz. Yani şunları:

```
>>> for i in dir(""):
...     if "_" not in i:
...         print(i)
...
capitalize
center
count
encode
endswith
expandtabs
find
format
index
isalnum
isalpha
isdecimal
isdigit
isidentifier
islower
isnumeric
isprintable
isspace
istitle
isupper
join
ljust
lower
lstrip
maketrans
partition
replace
rfind
rindex
rjust
rpartition
rsplit
rstrip
split
splitlines
startswith
strip
swapcase
title
translate
upper
zfill
```

Bu metotları karışık olarak inceleyeceğiz ilerleyen sayfalarda... Gelin isterseniz sözü daha fazla uzatmadan ilk metodumuzla başlayalım işe...

13.1 startswith Metodu

Bu bölümde inceleyeceğimiz ilk metodumuz `startswith()`. Bu metot, bize bir karakter dizisinin belli bir karakterle başlayıp başlamadığını söyler. Hemen bir örnek verelim:

```
>>> kardiz = "elveda zalim dünya!"
>>> kardiz.startswith("e")

True
```

Burada "elveda zalim dünya!" karakter dizisi "e" harfiyle başladığı için `startswith()` metodu *True* çıktısı veriyor. Bir de şuna bakalım:

```
>>> kardiz.startswith("i")

False
```

"elveda zalim dünya!" karakter dizisi "i" harfiyle başlamadığı için metodumuz bize *False* çıktısı verdi. Eğer `startswith()` diye bir metot olmasaydı yukarıdaki örnekleri şöyle yazmamız gerekecekti:

```
>>> kardiz[0] == "e"

True

>>> kardiz[0] == "i"

False
```

Bu örnekten de anladığımıza göre, tıpkı listelerde olduğu gibi karakter dizilerinde de, öğelerin sırasını belirterek bu öğelere tek tek erişebiliyoruz. Örneğin:

```
>>> kardiz = "python"

>>> kardiz[0] #karakter dizisinin ilk öğesini alıyoruz...

'p'

>>> kardiz[-1] #karakter dizisinin son öğesini alıyoruz...

'n'

>>> len(kardiz) #karakter dizisinin uzunluğunu ölçelim...

6
```

13.2 endswith Metodu

Sıradaki metodumuz `endswith()`. Bu metot yukarıda incelediğimiz `startswith()` metodunun yaptığı işin tam tersini yapar. O metot bir karakter dizisinin belli bir harfle başlayıp başlamadığını söylüyordu, bu metot ise bize bir karakter dizisinin belli bir karakterle bitip bitmediğini söyler:

```
>>> kardiz = "elveda zalim dünya!"
>>> kardiz.endswith("!")
```

```
True
```

Gördüğünüz gibi, karakter dizimiz "!" karakteriyle bittiği için yukarıdaki koddan *True* çıktısı aldık. Eğer karakter dizimiz "!" karakteriyle bitmeseydi *False* çıktısı alacaktık.

`endswith()` diye bir metot olmasaydı, aynı işlevi şöyle de yerine getirebilirdik:

```
>>> kardiz = "elveda zalim dünya!"
>>> kardiz[-1] == ("!")

True
```

13.3 islower Metodu

Bu metot bir karakter dizisinin tamamen küçük harflerden oluşup olmadığını denetler. Mesela:

```
>>> kardiz = "elif dedim be dedim"
>>> kardiz.islower()

True
```

Burada bütün karakterler küçük harfli olduğu için `islower()` metodu *True* çıktısı veriyor. Eğer karakter dizimizin içinde tek bir büyük harf dahi olsa bu metot *False* çıktısı verecektir:

```
>>> kardiz = "Elif dedim be dedim"
>>> kardiz.islower()

False
```

13.4 isupper Metodu

Bu metot ise, `islower()` metodunun yaptığı işin tam tersini yapar. Bu metot ile bir karakter dizisinin tamamen büyük harflerden oluşup oluşmadığını denetleyebiliriz:

```
>>> kardiz = "ELİF DEDİM BE DEDİM"
>>> kardiz.isupper()

True
```

Bu metot da, tıpkı `islower()` metodunda olduğu gibi, eğer karakter dizisinin içinde tek bir küçük harf varsa dahi `False` çıktısı verecektir...

13.5 replace Metodu

Bu metot, karakter dizisi metotları içinde en yararlı ve en çok kullanılan metotlardan biridir. Normal şartlar altında zorlukla halledebileceğimiz işlerin altından bu metot yardımıyla çok kolay bir şekilde kalkabiliriz. `replace()` metodu bir karakter dizisi içindeki karakterleri başka karakterlerle değiştirmemize olanak tanır:

```
>>> kardiz = "elma"
>>> kardiz.replace("e", "a")

'alma'
```

Gördüğünüz gibi, "elma" karakter dizisi içindeki "e" harfini "a" harfine dönüştürdük. Ancak burada önemli bir ayrıntıya dikkatinizi çekmek isterim. Hatırlarsanız karakter dizilerinden ilk kez bahsettiğimiz bölümde, karakter dizilerinin "değiştirilemeyen" (immutable) veri tipleri olduğunu söylemiştik. Bu özellik burada da karşımıza çıkıyor. Mesela yukarıdaki örnekte "elma" karakter dizisi üzerinde değişiklik yapmış gibi görünsek de, aslında karakter dizisinde herhangi bir değişiklik olmadı. Bunu şu şekilde test edebiliriz:

```
>>> print(kardiz)

'elma'
```

Gördüğünüz gibi, karakter dizimiz, `replace()` metoduyla "e" harfini "a" harfine dönüştürmüş olmamıza rağmen olduğu gibi duruyor. İşte bunun nedeni karakter dizilerinin değiştirilemeyen bir yapıda olmasıdır. Elbette bu kısıtlamayı da aşmanın bir yolu var:

```
>>> kardiz = kardiz.replace("e", "a")
>>> print(kardiz)

'alma'
```

Burada yaptığımız şey, yeni verileri "kardiz" adlı karakter dizisinin üzerine yazmaktan ibarettir... Yani biz burada "kardiz"in değiştirilmiş halini tekrar "kardiz"in kendisine atıyoruz. Böylece sonuç olarak karakter dizimiz üzerinde değişiklik yapmış oluyoruz. Karakter dizilerinin değiştirilemez oluşu oldukça önemli bir konudur. Yazdığımız kodların beklediğimiz şekilde çalışabilmesi için karakter dizilerinin bu özelliğini mutlaka göz önünde bulundurmamız gerekir.

Sizlere kolaylık olması açısından bu `replace()` metodunun formülünü de verelim:

```
karakter_dizisi.replace(eski_karakter, yeni_karakter)
```

Bu formüle göre, parantez içinde ilk sırada kendisini değiştireceğimiz karakteri belirtiyoruz. İkinci sırada ise değiştireceğimiz karakterin yerine gelecek yeni karakteri yazıyoruz.

`replace()` metodu yukarıdakilerin dışında üçüncü bir argüman daha alır. Bunu bir örnek üzerinden görelim:

```
>>> kardiz = "katar"
>>> kardiz.replace("a", "ı", 2)

'kıtır'
```

`replace()` metoduna verdiğimiz üçüncü argüman, belirtilen karakterin kaç yerde değiştirileceğini gösteriyor. Yukarıdaki örneğe göre biz "katar" adlı karakter dizisi içinde geçen 2 adet "a" harfini "ı" harfiyle değiştirmek istiyoruz. Eğer bu üçüncü argümanı belirtmezsek, `replace()` metodu bulduğu bütün "a" harflerini değiştirecektir:

```
>>> kardiz = "dal kalkar kartal sarkar, kartal kalkar dal sarkar"
>>> kardiz.replace("a", "@")

'd@l k@lk@r k@rt@l s@rk@r, k@rt@l s@rk@r d@l k@lk@r'
```

13.6 join Metodu

Tıpkı `replace()` metodu gibi, `join()` metodu da en çok kullanılan ve en işe yarar karakter dizisi metotlarından biridir. "join" kelimesi Türkçe'de birleştirmek anlamına gelir. Bu metodun görevi birden fazla karakter dizisini tek bir karakter dizisi haline getirmektir. Bu metodu, örneğin, karakter dizilerinden oluşmuş bir listeyi tek bir karakter dizisi haline getirmek için kullanabiliriz:

```
>>> liste = ["bugün", "günlerden", "pazar"]
>>> " ".join(liste)

'bugün günlerden pazar'
```

Gördüğünüz gibi, bu metodun kullanımı öteki metotlardan biraz farklı gibi görünüyor. Ama aslında ötekilerden pek bir farkı yoktur. Metodu nasıl kullandığımıza dikkat edin. Sanırım şöyle bir örnek daha verirsek bazı noktaları birazcık aydınlatmış oluruz:

```
>>> ", ".join(liste)

'bugün, günlerden, pazar'
```

Bu örnekten de anladığımız gibi, `join()` metodu listedeki bütün karakter dizilerini alıyor, her bir karakter dizisinin arasına "," işaretini yerleştiriyor ve bu karakter dizilerini tek bir karakter dizisi olarak birleştiriyor.

Hatırlarsanız *Genel Tekrar* bölümünde bu `join()` metoduyla ilgili olarak şöyle bir örnek vermiştik:

```
>>> liste = ["elma", "armut", "kebap"]
>>> print(", ".join(liste))

elma, armut, kebap
```

Artık bu örneğin nasıl çalıştığını anlayabiliyoruz. Burada listemizin bütün öğelerini alıp her bir öğeyi tek bir karakter dizisi halinde birleştirdik. Bu öğeleri birleştirirken de her bir öğenin arasına bir "," işareti koyduk... Elbette eğer istersek biz bu öğeleri istediğimiz başka bir işaretle de birleştirebiliriz. Mesela:

```
>>> liste = ["elma", "armut", "kebap"]
>>> print(" -- ".join(liste))

elma -- armut -- kebap
```

Bu arada, bu `join()` metodunun sadece karakter dizileri üzerinde işlediğini unutmuyoruz. Dolayısıyla, içinde karakter dizisi dışında bir veri tipi barındıran listeleri bu metotla birleştirmeye kalkarsak hata alırız:

```
>>> liste = [1, 2, "elma"]
>>> " -- ".join(liste)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sequence item 0: expected str instance, int found
```

13.7 split Metodu

Bu metot, temel olarak `join()` metodunun tam tersi bir vazife görür. `split()` metodunu kullanarak, bir karakter dizisinin içindeki öğeleri liste haline getirebiliriz:

```
>>> kardiz = "güzel günler göreceğiz"
>>> kardiz.split()

['güzel', 'günler', 'göreceğiz']
```

Eğer karakter dizisinin tamamını tek bir liste halinde almak isterseniz şöyle bir kod kullanabilirsiniz:

```
>>> kardiz.split(",")

['güzel günler göreceğiz']
```

Elbette, `split()` metoduna argüman olarak yalnızca virgül (",") vermek zorunda değiliz:

```
>>> kardiz.split("g")

['', 'üzel ', 'ünler ', 'öreceğiz']
```

Şöyle bir örnek daha vererek bu metot hakkındaki bilgimizi pekiştirelim:

```
>>> site = "www.istihza.com"
>>> site.split(".")

['www', 'istihza', 'com']
```

Burada "site" adlı karakter dizimizi, içindeki nokta işaretlerinden böldük.

`split()` metodu ek olarak bir parametre daha alır. Bu parametre bir sayıdır ve karakter dizisi üzerine kaç kez bölme işlemi uygulanacağını gösterir:

```
>>> site.split(".", 1)

['www', 'istihza.com']
```

Burada metodumuz, "site" adlı karakter dizisi üzerine sadece 1 kez bölme işlemi uyguladı. Buna göre, "www" kısmını böldükten sonra geri kalan kısmı tek bir öğe olarak liste içine dahil etti.

13.8 rsplit Metodu

Bu metot yukarıda anlattığımız `split()` metoduna çok benzer. Hatta tamamen aynı işi yapar. Tek bir farkla: `split()` metodu karakter dizilerini soldan sağa doğru okurken; `rsplit()` metodu sağdan sola doğru okur. Önce şöyle bir örnek verip bu iki metodun birbirine ne kadar benzediğini görelim:

```
>>> site = "www.istihza.com"
>>> site.split(".")

['www', 'istihza', 'com']

>>> site.rsplit(".")
```

```
['www', 'istihza', 'com']
```

Bu örnekte ikisi arasındaki fark pek belli olmasa da, aslında `split()` metodu karakter dizisini soldan sağa doğru okurken, `rsplit()` metodu sağdan sola doğru okuyor. Şimdi daha açık bir örnek verelim:

```
>>> site.split(".", 1)

['www', 'istihza.com']
```

Yukarıdaki örnek, `split()` metodunun davranışını gösteriyor. Şimdi şuna bakalım:

```
>>> site.rsplit(".", 1)

['www.istihza', 'com']
```

Gördüğünüz gibi, `rstrip()` metodu karakter dizisini sağdan sola doğru okuyup, tek bir bölme işlemi yaparak, sitenin "com" kısmını tek başına, "www.istihza" kısmını ise birlikte tek bir öğe olarak liste haline getirdi...

13.9 strip Metodu

`strip()`, karakter dizilerinin metotları arasında en ilginçlerinden biridir... Bu metodun, parametreli veya parametresiz verilişine bağlı olarak farklı görevleri vardır.

Bu metot parametresiz olarak kullanıldığında, bir karakter dizisinin başındaki ve sonundaki boşluk karakterlerini ve kaçış dizilerini kırpar. Örnek üzerinde görelim:

```
>>> kardiz = "\nVicdansız Sabuha\t"
>>> kardiz.strip()

'Vicdansız Sabuha'
```

Gördüğünüz gibi, bu metot yukarıdaki karakter dizisinin başındaki ve sonundaki "\n" ve "\t" kaçış dizilerini kırptı. Bu metot aynı zamanda baştaki ve sondaki boşlukları da siler:

```
>>> kardiz = " Vicdansız Sabuha "
>>> kardiz.strip()

'Vicdansız Sabuha'
```

Eğer bu metodu bazı parametrelerle kullanırsak daha farklı bir sonuç elde ederiz:

```
>>> kardiz = "Vicdansız Sabuha"
>>> kardiz.strip("ha")

'Vicdansız Sabu'
```

Burada `strip()` metodu, karakter dizimizin en sonunda yer alan "ha" harflerini sildi. Eğer bu karakter dizisinin başında da "ha" harfleri olsaydı onlar da gidecekti...

```
>>> kardiz = "hayırsız Sabuha"
>>> kardiz.strip("ha")

'yırsız Sabu'
```

Bu metot, kendisine parametre olarak verilen karakterlerin sırasını önemsemez. Yani şöyle bir şey de yazsanız baştaki ve sondaki “ha” harfleri uçacaktır:

```
>>> kardiz.strip("ah")

'yırsız Sabu'
```

Hatta şöyle bir şey dahi mümkündür:

```
>>> kardiz.strip("ahyı")

'rsız Sabu'
```

Burada `strip()` metodu karakter dizisinin başında ve sonunda gördüğü “a”, “h”, “y” ve “ı” harflerini kırptı.

Bu metotla ilgili şöyle bir kullanım örneği verebiliriz:

```
>>> liste = ["çilek", "şeker", "üzüm", "ısırgan", "ödül", "kitap", "defter",
... "soru"]
>>> for i in liste:
...     i.strip("çşöüı")
...
'ilek'
'eker'
'züm'
'sırgan'
'dül'
'kitap'
'defter'
'soru'
```

Burada, bir liste içinde geçen kelimelerin başındaki ve sonundaki Türkçe harfleri kırptık. Başında ve sonunda Türkçe harfler içermeyen kelimeler bu kodlardan etkilenmedi...

13.10 lstrip Metodu

`strip()` metodunu anlatırken bu metodun bir karakter dizisinin başına ve sonuna etki ettiğini görmüştük. Eğer istediğimiz şey karakter dizisinin sadece başında yer alan bazı karakterleri kırpmaksa `lstrip()` metodundan yararlanacağız.

Hatırlarsanız `strip()` metodunda şöyle bir örnek vermiştik:

```
>>> kardiz = "hayırsız Sabuha"
>>> kardiz.strip("ha")

'yırsız Sabu'
```

Bu örnekte doğal olarak `strip()` metodu karakter dizisinin her iki tarafını da kırpıyordu. Şimdi aynı örneği `lstrip()` ile yapalım:

```
>>> kardiz == "hayırsız Sabuha"
>>> kardiz.lstrip("ha")

'yırsız Sabuha'
```

Gördüğünüz gibi, `lstrip()` metodu karakter dizisinin sadece baş tarafıyla ilgilendi, son kısma hiç dokunmadı.

13.11 rstrip Metodu

`rstrip()` metodu yukarıda gördüğümüz `lstrip()` metodunun tersidir. Bu metot, bir karakter dizisinin sadece son tarafıyla ilgilenir:

```
>>> kardiz = "hayırsız Sabuha"
>>> kardiz.rstrip("ha")

'hayırsız Sabu'
```

Gördüğünüz gibi, `lstrip()` metodunun aksine `rstrip()` metodu karakter dizisinin sadece son tarafıyla ilgilendi, baş tarafa hiç ilişmedi.

13.12 upper Metodu

Sırada `upper()` adlı bir metot var. Bu metot yardımıyla bir karakter dizisi içindeki bütün harfleri büyütebileceğiz:

```
>>> kardiz = "merhaba dünya!"
>>> kardiz.upper()

'MERHABA DÜNYA'
```

13.13 lower Metodu

`lower()` metodu `upper()` metodunun yaptığı işin tam tersini yapar. Yani büyük olan harfleri küçültür:

```
>>> kardiz = "MERHABA DÜNYA!"
>>> kardiz.lower()

'merhaba dünya!'
```

13.14 capitalize Metodu

Yeni metodumuz `capitalize()`. Bu metodun görevi bir karakter dizisinin ilk harfini büyütmektir. Bunu şöyle kullanıyoruz:

```
>>> kardiz = "lahanayı yerken kıtır kıtır, sapına gelince meee!"
>>> kardiz.capitalize()

'Lahanayı yerken kıtır kıtır, sapına gelince meee!'
```

Gördüğünüz gibi, karakter dizimizin yalnızca ilk harfini büyüttü bu `capitalize()` adlı metot... Bu metodu (ve tabii ki öteki bütün metotları) istersek doğrudan karakter dizisinin üzerine de uygulayabiliriz:

```
>>> "çilek".capitalize()

'Çilek'
```

Bu arada, gördüğünüz gibi bu metot Türkçe harfleri büyütmekte zorlanmadı. Ancak bunun bir istisnası vardır. Bu metodun (ve birkaç başka metodun) anlaşamadığı tek bir Türkçe harf bulunur: "i" harfi. Sorunu kendi gözlerinizle görün:

```
>>> a = "işletim sistemi"
>>> a.capitalize()

'Işletim sistemi'
```

Buradaki sorun, "i" harfinin büyütülürken noktasını kaybetmesi... Ne yazık ki doğrudan Python'dan kaynaklanmayan bir sorun nedeniyle "i" harfini "İ" şeklinde büyütemiyoruz. Ancak endişe etmeyin. Bu "i" harfini doğrudan büyütemeyecek olsanız da, karşı karşıya olduğunuz duruma göre özel çözümler üreterek yolunuza devam edebilirsiniz. Basit bir örnek vereyim:

```
def tr_capitalize(kardiz):
    """Bir karakter dizisinin sadece ilk harfini büyütür."""
    if kardiz.startswith("i"):
        kardiz = kardiz[0].replace("i", "İ") + kardiz[1:]

    return kardiz.capitalize()
```

Aynen yukarıda olduğu gibi, eğer içinde Türkçe karakterler barındıran karakter dizileriyle uğraşacaksanız, `capitalize()` metodunu doğrudan kullanmak yerine, `if` deyimlerinden de yararlanarak kendinize özel çözümler üretebilirsiniz. Yukarıdaki `tr_capitalize()` adlı fonksiyon her türlü karakter dizisinin ilk harfini büyütecektir.

13.15 title Metodu

Şimdi de `title()` metoduna bakalım... Bu metot bir önceki bölümde incelediğimiz `capitalize()` metoduna benzer. O metot, bir karakter dizisinin sadece ilk harfini büyütüyordu, bu metot ise bir karakter dizisi içinde geçen bütün kelimelerin ilk harflerini büyütür. Hemen bununla ilgili bir örnek vererek metodun ne işe yaradığını anlamaya çalışalım:

```
>>> kardiz = "python programlama dili"
>>> kardiz.title()

'Python Programlama Dili'
```

Bu metot, gördüğünüz gibi, `capitalize` metodunun aksine karakter dizisinin sadece ilk harfini büyütmekle yetinmedi... Elbette bu metodu istersek doğrudan karakter dizisi üzerine de uygulayabiliriz:

```
>>> "python programlama dili".title()

'Python Programlama Dili'
```

Bu metot da Türkçe karakterlerle karşılaştığında genellikle bir sorun çıkarmaz:

```
>>> a = "çayır çömlek patladı"
>>> a.title()

'Çayır Çömlek Patladı'
```

Anca.. highlight:: py3 .. _kardiz-metot-upper:k tıpkı `capitalize()` metodunda olduğu gibi, burada da "i" harfi bize biraz sorun çıkaracaktır:

```
>>> a = "iyi incir"
>>> a.title()

'Iyi Incir"
```

Tabii yine endişelenmenize gerek yok. Eğer "i" harfini büyütmeniz gereken bir durumla karşı karşıya kalırsanız, kendi çözümlerinizi kendiniz de üretebilirsiniz. Basit bir örnek verelim yine:

```
def tr_title(kardiz):
    """Bir karakter dizisi içindeki her bir kelimenin ilk harfini büyütür."""
    depo = []
    for elem in kardiz.split():
        if not "i" in elem[0]:
            depo.append(elem.title())

        else:
            elem = elem[0].replace("i", "İ") + elem[1:]
            depo.append(elem)

    return " ".join(depo)
```

13.16 swapcase Metodu

Bu metot ile bir karakter dizisinin sahip olduğu büyük-küçük harf yapısını tersine çeviriyoruz. Şu örneğe bakalım:

```
>>> a = "python programlama"
>>> a.swapcase()

'PYTHON PROGRAMLAMA'
```

Gördüğünüz gibi, tamamı küçük olan harfler, bu metot yardımıyla tamamı büyük olacak şekilde değişti. Bir de şuna bakalım:

```
>>> a = "PYTHON PROGRAMLAMA"
>>> a.swapcase()

'python programlama'
```

Bu defa tamamı büyük olan harfler, bu metot yardımıyla tamamı küçük olacak şekilde değişti... Demek ki bu metot büyük harfleri küçük harfe, küçük harfleri ise büyük harfe dönüştürmeye yarıyor. Bir de şu örneğe bakalım:

```
>>> a = "Python programlama"
>>> a.swapcase()

'pYTHON PROGRAMLAMA'
```

`swapcase()` metodu normal olarak görevini yerine getirdi ve büyük harfleri küçük harfe, küçük harfleri de büyük harfe çevirdi...

Bu metodun da "i" harfiyle sorunu olduğunu tahmin etmişsinizdir. Burada da kendi çözümlerinizi kendiniz üretebilirsiniz:

```
def tr_swapcase(kardiz):
    """Bir karakter dizisi içindeki büyük-küçük harfleri değiştirir."""
```

```
    depo = []
    for elem in kardiz:
        if "i" not in elem:
            depo.append(elem.swapcase())

        elif "i" in elem and elem.islower():
            elem = elem.replace("i", "İ")
            depo.append(elem.upper())

        else:
            depo.append(elem.lower())

    return "".join(depo)
```

13.17 istitle Metodu

Hatırlarsanız `title()` adlı metodu kullanarak bir karakter dizisi içindeki bütün kelimelerin ilk harflerini büyütebiliyorduk. Bu `istitle()` metoduyla da bir karakter dizisi içindeki bütün kelimelerin ilk harflerinin büyük olup olmadığını sorguluyoruz:

```
>>> kardiz = "Python Programlama Dili"
>>> kardiz.istitle()

True

>>> kardiz = "Python programlama dili"
>>> kardiz.istitle()

False
```

Gördüğünüz gibi, eğer karakter dizisi içindeki bütün kelimelerin ilk harfleri büyükse `istitle()` metodu *True* çıktısı veriyor. Aksi bir durumda ise *False* çıktısı alıyoruz...

13.18 ljust Metodu

`ljust()` metodu bize özellikle karakter dizilerinin hizalama işlemlerinde yardımcı olacak. Bu metot yardımıyla karakter dizilerimizi sola yaslayıp, sağ tarafına da istediğimiz karakterleri yerleştirebileceğiz. Hemen bir örnek verelim:

```
>>> kardiz = "tel no"
>>> kardiz.ljust(10, ".")

'tel no....'
```

Burada olan şey şu: `ljust()` metodu, kendisine verilen "10" parametresinin etkisiyle 10 karakterlik bir alan oluşturuyor. Bu 10 karakterlik alanın içine önce 6 karakterlik yer kaplayan "tel no" ifadesini, geri kalan 4 karakterlik boşluğa ise "." karakterini yerleştiriyor. Eğer `ljust()` metoduna verilen sayı karakter dizisinin uzunluğundan az yer tutarsa, karakter dizisinin görünüşünde herhangi bir değişiklik olmayacaktır. Örneğin yukarıdaki örnekte karakter dizimizin uzunluğu "6". Dolayısıyla kodumuzu şu şekilde yazarsak bir sonuç elde edemeyiz:

```
>>> kardiz.ljust(5, ".")

'tel no'
```

Gördüğünüz gibi, karakter dizisinde herhangi bir değişiklik olmadı. `ljust()` metoduna verdiğimiz "." karakterini görebilmemiz için, verdiğimiz sayı cinsli parametrenin en az karakter dizisinin boyunun bir fazlası olması gerekir:

```
>>> kardiz.ljust(7, ".")

'tel no.'
```

`ljust()` metoduyla ilgili basit bir örnek daha verelim:

```
>>> liste = ["elma", "armut", "patlıcan"]
>>> for i in liste:
...     i.ljust(10, ".")
...
'elma......'
'armut.....'
'patlıcan..'
```

Gördüğünüz gibi, bu metot karakter dizilerini şık bir biçimde sola hizalamamıza yardımcı oluyor...

13.19 rjust Metodu

Bu metot, `ljust()` metodunun yaptığı işin tam tersini yapar. Yani karakter dizilerini sola değil sağa yaslar:

```
>>> liste = ["elma", "armut", "patlıcan"]
>>> for i in liste:
...     i.rjust(10, ".")
...
'      elma'
'     armut'
'  patlıcan'
```

13.20 center Metodu

Bu metot ise karakter dizilerini ne sola ne de sağa yaslar. `center()` metodunun görevi karakter dizilerini ortalamaktır:

```
>>> kardiz = "elma"
>>> kardiz.center(10, "-")

'---elma---'
```

13.21 count Metodu

Bu metot ile bir karakter dizisi içindeki harflerin, o karakter dizisi içinde kaç kez geçtiğini öğreneceğiz:

```
>>> kardiz = "karakter dizisi"
>>> kardiz.count("a")

2
```

`count()` metodu ilave olarak bir parametre daha alır. Örnek üzerinden açıklamaya çalışalım:

```
>>> kardiz.count("a", 0)

2
```

Burada "0" parametresi, "a" harfini ararken metodumuzun karakter dizisinin kaçıncı sırasından başlaması gerektiğini gösteriyor. Yukarıdaki koda göre, karakter dizisinin 0. sırasından başlayarak, diziyi sonuna kadar tarayacağız ve "a" harfinin kaç kez geçtiğini raporlayacağız. Bir de şu örneğe bakalım:

```
>>> kardiz.count("a", 2)

1
```

Bu defa "1" çıktısı almamızın nedeni, "a" harfini karakter dizisinin soldan sağa 2. sırasından itibaren aramamızdır. Gördüğünüz gibi, "karakter dizisi" ifadesi içinde "a" harfi 1 ve 3. sıralarda yer alıyor... Eğer şöyle bir kod yazarsak "0" çıktısı alırız:

```
>>> kardiz.count("a", 5)

0
```

Burada "0" çıktısı almamızın nedeni, karakter dizisinin 5. sırasından sonra herhangi bir "a" harfinin bulunmamasıdır.

13.22 find Metodu

Bu metot bir karakter dizisi içindeki karakterin sıra numarasını söyler bize:

```
>>> kardiz = "karakter dizisi"
>>> kardiz.find("t")

5
```

Buna göre karakter dizimiz içinde "t" harfi 5. sırada bulunuyormuş. Bunu teyit edelim:

```
>>> kardiz[5]

't'
```

`find()` metodundan aldığınız bilgiyi pek çok farklı yerde kullanabilirsiniz. Mesela bir karakter dizisinin belli bir harften sonraki kısmını almak için şöyle bir kod yazabilirsiniz:

```
>>> kardiz[kardiz.find("t"):]

'ter dizisi'
```

Bu metot, aradığımız harfin sadece ilk geçtiği sırayı verecektir. Örneğin yukarıdaki karakter dizisinde "a" harfi iki kez geçiyor. Ancak `find()` metodu yalnızca ilk "a" harfinin sırasını söyler:

```
>>> kardiz.find("a")

1
```

Eğer ikinci “a” harfinin sırasını öğrenmek isterseniz `find()` metodunu şu şekilde kullanabilirsiniz:

```
>>> kardiz.find("a", 2)

3
```

Gördüğünüz gibi, `find()` metoduna sayı değerli bir parametre vererek, bir harfin karakter dizisi içindeki kaçıncı tekrarını almak istediğimizi belirtebiliyoruz.

Eğer `find()` metodunu kullanarak, bir harfin geçtiği bütün sıra numaralarını almak isterseniz şöyle bir şey yazabilirsiniz:

```
>>> for i in range(1, kardiz.count("a")+1):
...     print(kardiz.find("a", i))
...
1
3
```

Eğer `find()` metodu aradığını karakteri bulamazsa “-1” çıktısı verecektir:

```
>>> kardiz.find("j")

-1
```

13.23 rfind Metodu

Bu metot, `find()` metoduyla aynı işi yapar. Ama `rfind()` metodu `find()` metodunun aksine, bir karakter dizisini sağdan sola doğru tarar:

```
>>> kardiz = "karakter dizisi"
>>> kardiz.find("a")

3
```

Gördüğünüz gibi, bu metot ilk “a” harfini değil, ikinci “a” harfini verdi.

Bu metot da tıpkı `find()` metodu gibi, eğer aradığınız karakteri bulamazsa “-1” çıktısı verecektir...

13.24 index Metodu

`index()` metodu `find()` metodu ile tamamen aynı işi yapar. Bu iki metot arasındaki tek fark, aranan karakter bulunamadığında verilen çıktıdır. Hatırlarsanız, `find()` metodunun, aradığımız karakteri bulamaması halinde “-1” çıktısı verdiğini söylemiştik. `index()` metodu ise, aradığımız karakteri bulamadığında doğrudan bir hata mesajı verecektir:

```
>>> kardiz.index("g")

ValueError: substring not found
```

13.25 rindex Metodu

Bu metot da tıpkı `rfind()` metodu gibidir. Burada da tek fark, karakter dizisinin bulunamadığı durumda verilen cevaptır:

```
>>> kardiz.rindex("g")

ValueError: substring not found
```

13.26 splitlines Metodu

Bu metot, birkaç satırdan oluşan karakter dizilerini satır satır böler:

```
>>> kardiz = "birinci satır.\nikinci satır.\nüçüncü satır.\ndördüncü satır"
>>> kardiz.splitlines()

['birinci satır.', 'ikinci satır.', 'üçüncü satır.', 'dördüncü satır']
```

Eğer bu metoda herhangi bir sayı değerli parametre verecek olursanız, şöyle bir çıktı alırsınız:

```
>>> kardiz.splitlines(1)

['birinci satır.\n', 'ikinci satır.\n', 'üçüncü satır.\n', 'dördüncü satır']
```

Burada hangi sayıyı verdiğinizin önemi yoktur. Parantez içinde herhangi bir sayı olması durumunda `splitlines()` metodu satır sonu işaretlerini de (\n) çıktıya dahil edecektir.

13.27 isalpha Metodu

Bu metot yardımıyla bir karakter dizisinin "alfabetik" olup olmadığını denetleyeceğiz. Peki "alfabetik" ne demektir? Eğer bir karakter dizisi içinde yalnızca alfabe harfleri (a, b, c gibi...) varsa o karakter dizisi için "alfabetik" diyoruz. Bir örnekle bunu doğrulayalım:

```
>>> a = "kezban"
>>> a.isalpha()

True
```

Ama:

```
>>> b = "k3zb6n"
>>> b.isalpha()

False
```

13.28 isdigit Metodu

Bu metot da isalpha metoduna benzer. Bunun yardımıyla bir karakter dizisinin "sayısal" olup olmadığını denetleyebiliriz. Sayılardan oluşan karakter dizilerine "sayı karakter dizileri" adı verilir. Örneğin şu bir "sayı karakter dizisi"dir:

```
>>> a = "12345"
```

Metodumuz yardımıyla bunu doğrulayabiliriz:

```
>>> a.isdigit()

True
```

Ama şu karakter dizisi sayısal değildir:

```
>>> b = "123445b"
```

Hemen kontrol edelim:

```
>>> b.isdigit()

False
```

13.29 isalnum Metodu

Bu metot, bir karakter dizisinin “alfanümerik” olup olmadığını denetlememizi sağlar. Peki “alfanümerik” nedir?

Daha önce bahsettiğimiz metotlardan hatırlayacaksınız:

“Alfabetik” karakter dizileri, alfabe harflerinden oluşan karakter dizileridir.

“Sayısal” karakter dizileri, sayılardan oluşan karakter dizileridir.

“Alfanümerik” karakter dizileri ise bunun birleşimidir. Yani sayı ve harflerden oluşan karakter dizilerine alfanümerik karakter dizileri adı verilir. Örneğin şu karakter dizisi alfanümerik bir karakter dizisidir:

```
>>> a = "123abc"
```

İsterseniz hemen bu yeni metodumuz yardımıyla bunu doğrulayalım:

```
>>> a.isalnum()

True
```

Eğer denetleme sonucunda “True” alıyorsak, o karakter dizisi alfanümeriktir. Bir de şuna bakalım:

```
>>> b = "123abc>"
>>> b.isalnum()

False
```

b değişkeninin tuttuğu karakter dizisinde alfanümerik karakterlerin yanısıra (“123abc”), alfanümerik olmayan bir karakter dizisi de bulunduğu için (“>”), b.isalnum() şeklinde gösterdiğimiz denetlemenin sonucu “False” (yanlış) olarak görünecektir.

Dolayısıyla, bir karakter dizisi içinde en az bir adet alfanümerik olmayan bir karakter dizisi bulunursa (bizim örneğimizde “<”), o karakter dizisi alfanümerik olmayacaktır.

13.30 isdecimal Metodu

Bu metot yardımıyla bir karakter dizisinin ondalık sayı cinsinden olup olmadığını denetliyoruz. Mesela aşağıdaki örnek ondalık sayı cinsinden bir karakter dizisidir:

```
>>> a = "123"
>>> a.isdecimal()

True
```

Ama şu ise "kayan noktalı" (floating-point) sayı cinsinden bir karakter dizisidir:

```
>>> a = "123.3"
>>> a.isdecimal()

False
```

Dolayısıyla `a.isdecimal()` komutu *False* çıktısı verir...

13.31 isidentifier Metodu

"identifier" kelimesi Türkçe'de "tanımlayıcı" anlamına gelir. Python'da değişkenler, fonksiyon ve modül adlarına "tanımlayıcı" denir. İşte başlıkta gördüğümüz `isidentifier()` metodu, neyin tanımlayıcı olup neyin tanımlayıcı olamayacağını denetlememizi sağlar. Hatırlarsanız değişkenler konusundan bahsederken, değişken adı belirlemenin bazı kuralları olduğunu söylemiştik. Buna göre, örneğin, değişken adları bir sayı ile başlayamıyordu. Dolayısıyla şöyle bir değişken adı belirleyemiyoruz:

```
>>> 1a = 12
```

Dediğimiz gibi, değişkenler birer tanımlayıcıdır. Dolayısıyla bir değişken adının geçerli olup olmadığını `isidentifier()` metodu yardımıyla denetleyebiliriz:

```
>>> "1a".isidentifier()

False
```

Demek ki "1a" ifadesini herhangi bir tanımlayıcı adı olarak kullanamıyoruz. Yani bu ada sahip bir değişken, fonksiyon adı veya modül adı oluşturamıyoruz. Ama mesela "liste1" ifadesi geçerli bir tanımlayıcıdır. Hemen denetleyelim:

```
>>> "liste1".isidentifier()

True
```

13.32 isnumeric Metodu

Bu metot bir karakter dizisinin "nümerik" olup olmadığını denetler. Yani bu metot yardımıyla bir karakter dizisinin sayı değerli olup olmadığını denetleyebiliriz:

```
>>> "12".isnumeric()

True
```

```
>>> "dasd".isnumeric()

False
```

13.33 isprintable Metodu

Eğer bir karakter ekrana basılabiliyorsa bu metot *True* çıktısı verecektir. Aksi halde bu metottan "False* yanıtını alırız. Temel olarak bütün harfler, sayılar ve boşluk işareti basılabilir karakterlerdir. Ama örneğin kaçış dizileri basılabilir karakterler değildir:

```
>>> "a".isprintable()

True

>>> "123".isprintable()

True

>>> "".isprintable()

True

>>> "\n".isprintable()

False
```

13.34 isspace Metodu

Bu metot yardımıyla bir karakter dizisinin tamamen boşluklardan oluşup oluşmadığını denetleyebiliriz. Eğer karakter dizimiz boşluklardan oluşuyorsa bu metot *True* çıktısı verecek, ama eğer karakter dizimizin içinde bir tane bile boşluk harici karakter varsa bu metot *False"* çıktısı verecektir:

```
>>> a = " "
>>> a.isspace()

True

>>> a = "               "
>>> a.isspace()

True

>>> a = "" #karakter dizimiz tamamen boş. İçinde boşluk karakteri bile yok...
>>> a.isspace()

False

>>> a = "fd"
>>> a.isspace()

False
```

13.35 zfill Metodu

Bu metot kimi yerlerde işimizi epey kolaylaştırabilir. `zfill()` metodu yardımıyla karakter dizlerinin sol tarafına istediğimiz sayıda sıfır ekleyebiliriz:

```
>>> a = "12"
>>> a.zfill(3)

'012'
```

Bu metodu şöyle bir iş için kullanabilirsiniz:

```
>>> for i in range(11):
...     print(str(i).zfill(2))
00
01
02
03
04
05
06
07
08
09
10
```

Burada `str()` metodunu kullanarak `range()` fonksiyonundan elde ettiğimiz sayıları birer karakter dizisine çevirdiğimize dikkat edin. Çünkü `zfill()` karakter dizilerinin bir metodudur... Sayıların değil...

13.36 encode Metodu

Bu metot yardımıyla karakter dizilerimizi istediğimiz kodlama sistemine göre kodlayabiliriz. Python 3.x'te varsayılan karakter kodlaması "utf-8"'dir. Eğer istersek şu karakter dizisini "utf-8" yerine "cp1254" ile kodlayabiliriz:

```
>>> "çilek".encode("cp1254")
```

13.37 expandtabs Metodu

Bu metot yardımıyla bir karakter dizisi içindeki sekme boşluklarını genişletebiliyoruz. Örneğin:

```
    >>> a = "elma\tbir\tmeyvedir"
    >>> a.expandtabs(10)
```

    'elma bir meyvedir'

13.38 partition Metodu

Bu metot yardımıyla bir karakter dizisini belli bir ölçüte göre üçe bölüyoruz. Örneğin:

```
>>> a = "istanbul"
>>> a.partition("an")
```

('ist', 'an', 'bul')

Eğer `partition()` metoduna parantez içinde verdiğimiz ölçüt karakter dizisi içinde bulunmuyorsa şu sonuçla karşılaşırız:

```
>>> a = "istanbul"
>>> a.partition("h")
```

('istanbul', '', '')

13.39 rpartition Metodu

Bu metot da `partition()` metodu ile aynı işi yapar, ama yöntemi biraz farklıdır. `partition()` metodu karakter dizilerini soldan sağa doğru okur. `rpartition()` metodu ise sağdan sola doğru.. Peki bu durumun ne gibi bir sonucu vardır? Hemen görelim:

```
>>> b = "istihza"
>>> b.partition("i")

('', 'i', 'stihza')
```

Gördüğünüz gibi, `partition()` metodu karakter dizisini ilk "i" harfinden böldü. Şimdi aynı işlemi `rpartition()` metodu ile yapalım:

```
>>> b.rpartition("i")

('ist', 'i', 'hza')
```

`rpartition()` metodu ise, karakter dizisini sağdan sola doğru okuduğu için ilk "i" harfinden değil, son "i" harfinden böldü karakter dizisini...

`partition()` ve `rpartition()` metotları, ölçütün karakter dizisi içinde bulunmadığı durumlarda da farklı tepkiler verir:

```
>>> b.partition("g")

('istihza', '', '')

>>> b.rpartition("g")

('', '', 'istihza')
```

Gördüğünüz gibi, `partition()` metodu boş karakter dizilerini sağa doğru yaslarken, `rpartition()` metodu sola doğru yasladı.

13.40 str.maketrans ve translate Metotları

Bu iki metot birbiriyle bağlantılı olduğu için, bunları bir arada göreceğiz.

Diyelim ki elimizde bir karakter dizisi var ve biz bu karakter dizisini şifrelemek istiyoruz. Bunun için şöyle bir kod yazdığımızı varsayalım:

```
sözlük = {"a":"0",
          "b":"1",
          "c":"2",
          "ç":"3",
          "d":"4",
          "e":"5",
          "f":"6",
          "g":"7",
          "ğ":"8",
          "h":"9",
          "ı":"a",
          "i":"b",
          "j":"c",
          "k":"ç",
          "l":"d",
          "m":"e",
          "n":"f",
          "o":"g",
          "ö":"h",
          "p":"ı",
          "r":"i",
          "s":"j",
          "ş":"k",
          "t":"l",
          "u":"m",
          "ü":"n",
          "v":"o",
          "y":"ö",
          "z":"p",
          " ":" "}

kardiz = "python programlama dili"

şifreli = ""
for i in kardiz:
        şifreli = şifreli + sözlük[i]

print(şifreli)
```

Bu kodu çalıştırdığımız zaman şöyle bir çıktı alırız:

```
ıöl9gf ıig7i0ed0e0 4bdb
```

Aslında bu çıktıda “python programlama dili” yazıyor. Biz yukarıdaki kodlar yardımıyla karakter dizimizi şifreledik. Python, yazdığımız sözlüğü temel alarak “p” harfi gördüğü yere “ı” harfini; “y” harfi gördüğü yere “ö” harfini, “t” gördüğü yere “l” harfini... yerleştiriyor. Böylece karakter dizimizin çıplak gözle anlaşılmasını engelleyecek bir çıktı veriyor bize. İsterseniz yukarıdaki kodları bir fonksiyon olarak tanımlayıp çok daha verimli bir hale getirebiliriz:

```
sözlük = {"a":"0",
          "b":"1",
          "c":"2",
          "ç":"3",
          "d":"4",
          "e":"5",
          "f":"6",
          "g":"7",
          "ğ":"8",
```

```
            "h":"9",
            "ı":"a",
            "i":"b",
            "j":"c",
            "k":"ç",
            "l":"d",
            "m":"e",
            "n":"f",
            "o":"g",
            "ö":"ğ",
            "p":"h",
            "r":"ı",
            "s":"i",
            "ş":"j",
            "t":"k",
            "u":"l",
            "ü":"m",
            "v":"n",
            "y":"o",
            "z":"ö",
            " ":" "}

def sifrele(kardiz):
        şifreli = ""
        for i in kardiz:
                şifreli = şifreli + sözlük[i]

        return şifreli

print(sifrele("python programlama dili"))
```

Fonksiyonumuzu bir kez bu şekilde tanımladıktan sonra sadece `sifrele()` fonksiyonunu kullanarak karakter dizilerimizi şifreleyebiliriz:

```
>>> print(sifrele("merhaba dünya"))

e5i9010 4nfö0
```

Burada bazı sorunlar olduğu açık. Mesela sözlük içinde tanımlanmamış karakter kullanıldığında programımız hata verecektir. Ayrıca açıkçası bu kodlar biraz "kalabalık" görünüyor göze...

İsterseniz yukarıdaki işlemi daha sade ve performanslı bir biçimde yapmanın bir yolu vardır Python'da...

Bunun için `str.maketrans()` ve `translate()` metotlarından yararlanacağız. Şu kodlara bir bakalım:

```
kaynak_dizi = "abcçdefgğhıijklmnoöprsştuüvyz"
hedef_dizi =  "0123456789abcçdefgğhıijklmnoö"
çeviri_nesnesi = str.maketrans(kaynak_dizi, hedef_dizi)

kardiz = "python programlama dili"

sonuç = kardiz.translate(çeviri_nesnesi)

print(sonuç)
```

Burada yaptığımız şey çok basit. Öncelikle "kaynak_dizi" adında bir karakter dizisi oluşturduk. Amacımız bir karakter dizisi içindeki karakterleri başka karakterlere dönüştürmek. İşte bu "kay-

nak_dizi" adlı karakter dizisi özgün kaynak dizimizin karakterlerini temsil ediyor. Alt satırdaki "hedef_dizi" adlı karakter dizisi ise, kaynak dizi içindeki karakterlerin dönüştürüleceği karakterleri gösteriyor. Bu iki karakter dizisini, bir üstteki kodlarda sözlük halinde yazmıştık. Buradaki "kaynak_dizi" adlı karakter dizisi bir önceki kodlardaki sözlüğün "anahtar"larının (keys), "hedef_dizi" adlı karakter dizisi ise sözlüğün "değer"lerinin (values) yerini tutuyor.

Karakter dizilerimizi tanımladıktan sonra `str.maketrans()` metodunu kullanarak bir "çeviri nesnesi" oluşturacağız. Oluşturacağımız bu çeviri nesnesi, biraz önce tanımladığımız "kaynak_dizi" ve "hedef_dizi" adlı karakter dizilerinin öğelerini birbiriyle eşleştirme vazifesi görecek. Zaten "*str.maketrans(kaynak_dizi, hedef_dizi)*" satırı bu vazifeyi gözler önüne seriyor... Bu arada `maketrans()` metodunu, öteki metotlardan farklı olarak `str.maketrans()` şeklinde kullandığımıza özellikle dikkat edin.

Hemen ardından çeviri işlemine tabi tutacağımız karakter dizimizi yazdık. Bu karakter dizimiz "python programlama dili".

Daha sonra da "sonuç" adlı bir değişken içinde `translate()` metodundan faydalanarak yukarıda oluşturduğumuz "çeviri_nesnesi"ni bu metoda argüman olarak veriyoruz. Bu metot, "kardiz" adlı karakter dizisini alıp, yukarıda "çeviri_nesnesi" içinde belirlediğimiz formüle göre çeviri işlemine tabi tutuyor ve ortaya şifrelenmiş bir karakter dizisi çıkarıyor...

13.41 format Metodu

`format()`, karakter dizilerinin en önemli metotlarından biridir. Bu metot Python'un 3.x sürümü ile birlikte dile dahil oldu. Aslında biz bu metodu daha önce *Karakter Dizilerini Biçimlendirme* konusunu işlerken de anlatmıştık. Burada ise bu metodu biraz daha ayrıntılı bir şekilde inceleyeceğiz.

Dediğimiz gibi, `format()` metodu Python'a 3.x sürümüyle birlikte dahil oldu. 3.x'ten önce `format()` metodunun yaptığı işi şu şekilde yerine getiriyorduk:

```
>>> "%s %s'yi seviyor!"%("Ali", "Ayşe")

'Ali Ayşe'yi seviyor!'
```

Esasında bu yöntem Python'un 3.x sürümünde de geçerlidir. Yani yukarıdaki yöntemi kullanarak hâlâ karakter dizilerini biçimlendirebiliriz. Ancak bu yöntem artık "eskimiş" olarak kabul ediliyor ve yerini yavaş yavaş `format()` metoduna bırakıyor. Dolayısıyla yeni yazdığınız kodlarda "%" işareti yerine `format()` metodunu kullanmanız şiddetle tavsiye edilir...

Gelelim bu metodun ayrıntılarına...

Temel olarak `format()` metodunu nasıl kullanacağımızı biliyoruz. Basit bir örnek verelim:

```
>>> zaman = "Cumartesi"
>>> "Seninle {0} günü buluşalım".format(zaman)

'Seninle Cumartesi günü buluşalım'
```

Burada {0} biçiminde belirttiğimiz kısma, `format()` metodu içinde belirttiğimiz değişken adı otomatik olarak yerleştiriliyor. Buradaki "0" sayısı, parantez içinde gösterilecek öğelerin sırasına işaret ediyor:

```
>>> zaman = "Pazartesi"
>>> saat = "12:30"
>>> "Seninle {0} günü saat {1}'da buluşalım".format(zaman, saat)
```

```
"Seninle Pazartesi günü saat 12:30'da buluşalım"
```

Python'un 3.0 sürümünde, yukarıdaki kodlarda görünen {} alanları içine mutlaka sıra belirten bir sayı yazmamız gerekiyordu. Ancak Python'un 3.1 sürümüyle birlikte bu zorunluluk ortadan kaldırıldı. Yani Python 3.1'den itibaren artık yukarıdaki kodları şöyle de yazma imkanına sahibiz:

```
>>> "Senin {} günü saat {}'da buluşalım".format(zaman, saat)
```

Eğer istersek yine {} içinde sayı belirtmekte özgürüz. Ama artık bu bir zorunluluk değil... Ama `format()` metodu içinde belirtilen parametrelerin doğal sırasını bozacaksak, sayıyı belirtmemiz gerekir:

```
>>> "{1} {0}'yi seviyor!".format("Ali", "Ayşe")

'Ayşe Ali'yi seviyor!'
```

`format()` metodu en basit haliyle yukarıdaki gibidir. Ancak bu metot bize çok daha gelişmiş özellikler de sunar. Şu örneğe bir bakalım:

```
>>> liste = ["elma", "armut", "kebap"]
>>> "listenin ilk öğesi şudur: {0[0]}".format(liste)
>>> "listenin ikinci öğesi şudur: {0[1]}".format(liste)
>>> "listenin üçüncü öğesi şudur: {0[2]}".format(liste)
```

Gördüğünüz gibi, "{}" alanı içinde liste öğelerini dilimleme imkanına da sahibiz... Yukarıdaki kodları şöyle de yazabiliriz:

```
>>> "listenin ilk öğesi şudur :", liste[0]
>>> "listenin ikinci öğesi şudur :", liste[1]
>>> "listenin üçüncü öğesi şudur :", liste[2]
```

Elbette `format()` metodu yukarıdaki yönteme göre çok daha esnek ve pratiktir...

`format()` metodunu ayrıca şu şekilde de kullanabiliriz:

```
>>> "Son güncelleme: {gun} {ay} {yil}".format(gun="27", ay="05", yil="2009")
```

Burada "{}" alanı içinde isim kullanarak kullanımı daha kolay bir yapı oluşturduğumuza dikkat edin.