

---

istihza.com

# **Python Kılavuzu**

*Sürüm 2.x*

**Fırat Özgöl**

17/04/2012



---

# İÇİNDEKİLER

---

1	Temel Bilgiler	2
1.1	Python Hakkında	2
1.2	Python Nasıl Okunur?	3
1.3	Python Nereden İndirilir?	3
1.4	Python Nasıl Çalıştırılır?	5
1.5	Python'dan Nasıl Çıkılır?	7
1.6	Bölüm Soruları	8
2	Python'a Giriş	9
2.1	print Komutu	9
2.2	Karakter Dizileri	11
2.3	Python'da Sayılar ve Matematik İşlemleri	16
2.4	Değişkenler	19
2.5	Kaçış Dizileri	20
2.6	Bölüm Soruları	25
3	Python Programlarını Kaydetmek	27
3.1	GNU/Linux Sistemi	27
3.2	Windows Sistemi	31
3.3	Türkçe Karakter Kullanımı	38
3.4	MS-DOS'ta Türkçe Karakter Problemi	41
3.5	Bölüm Soruları	42
4	Kullanıcıyla İletişim: Veri Alış-Verişi	43
4.1	raw_input() fonksiyonu	44
4.2	input() fonksiyonu	46
4.3	Güvenlik Açısından input() ve raw_input()	49
4.4	Bölüm Soruları	50

5	Python’da Koşula Bağlı Durumlar	51
5.1	if Deyimi	51
5.2	elif Deyimi	53
5.3	else Deyimi	56
5.4	Python’da Girintileme Sistemi	57
5.5	Python Kodlarına Yorum Ekleme	59
5.6	Bölüm Soruları	61
6	Bazı Önemli Ayrıntılar	62
6.1	İşleçler (Operators)	62
6.2	Bool Değerler	66
6.3	Dönüştürme İşlemleri	71
6.4	Bölüm Soruları	74
7	Python’da Döngüler	76
7.1	while Döngüsü	76
7.2	for Döngüsü	80
7.3	range() fonksiyonu	82
7.4	len() fonksiyonu	82
7.5	break deyimi	83
7.6	continue deyimi	84
7.7	Bölüm Soruları	85
8	Hata Yakalama	86
8.1	try... except...	87
8.2	pass Deyimi	90
8.3	Bölüm Soruları	91
9	Listeler	92
9.1	Liste Oluşturmak	92
9.2	Liste Öğelerine Erişmek	93
9.3	Liste Metotları	96
9.4	sum() Fonksiyonu	104
9.5	enumerate() Fonksiyonu	105
9.6	in Deyimi	107
9.7	min() ve max() Fonksiyonları	108
9.8	Bölüm Soruları	108
10	Demetler	110
10.1	Demetlerin Metotları	112
10.2	Bölüm Soruları	113
11	Sözlükler	114
11.1	Sözlük Oluşturmak	114
11.2	Sözlük Öğelerine Erişmek	115
11.3	Sözlük Öğelerini Değiştirmek	119
11.4	Sözlük Öğelerini Silmek	119

11.5	Sözlüklerin Metotları . . . . .	120
11.6	Bölüm Soruları . . . . .	124
12	Kümeler . . . . .	126
12.1	Küme Oluşturmak . . . . .	126
12.2	Kümelerin Metotları . . . . .	128
12.3	Dondurulmuş Kümeler (Frozenset) . . . . .	138
12.4	Bölüm Soruları . . . . .	139
13	Fonksiyonlar . . . . .	140
13.1	Fonksiyonları Tanımlamak . . . . .	141
13.2	Fonksiyonlarda Parametre Kullanımı . . . . .	144
13.3	İsimli ve Sıralı Argümanlar . . . . .	148
13.4	Varsayılan Değerli Argümanlar . . . . .	152
13.5	İstenen Sayıda Sıralı Argüman Kullanımı . . . . .	155
13.6	İstenen Sayıda İsimli Argüman Kullanımı . . . . .	159
13.7	Gömülü Fonksiyonlar (Built-in Functions) . . . . .	162
13.8	global Deyimi . . . . .	163
13.9	return Deyimi . . . . .	166
13.10	Fonksiyonlarda pass Deyimi . . . . .	168
13.11	Fonksiyonların Belgelendirilmesi . . . . .	169
13.12	Bölüm Soruları . . . . .	171
14	Modüller . . . . .	173
14.1	Modüllerin Çeşitleri . . . . .	173
14.2	Kendi Yazdığınız Modüller . . . . .	174
14.3	Modülleri İçe Aktarmak . . . . .	175
14.4	Modülleri İçe Aktarma Yöntemleri . . . . .	178
14.5	Geliştiricilerin Yazdığı Modüller . . . . .	180
14.6	Üçüncü Şahısların Yazdığı Modüller . . . . .	192
14.7	Modüllerin Yolu . . . . .	199
14.8	Bölüm Soruları . . . . .	200
15	Dosya İşlemleri . . . . .	201
15.1	Dosya Oluşturmak . . . . .	201
15.2	Dosyaya Yazmak . . . . .	205
15.3	Dosyayı Okumak . . . . .	207
15.4	Dosya Silmek . . . . .	209
15.5	Dosyaya Rastgele Satır Ekleme . . . . .	210
15.6	Dosyadan Rastgele Satır Silmek . . . . .	212
15.7	os Modülünde Dosya-Dizin İşlemleri . . . . .	213
15.8	Kümeler ve Dosyalar . . . . .	215
15.9	with Deyimi . . . . .	217
15.10	Bölüm Soruları . . . . .	218
16	Karakter Dizileri . . . . .	219
16.1	Karakter Dizisi Nedir? . . . . .	219

16.2	Tırnak Tipleri . . . . .	221
16.3	Karakter Dizilerini Birleştirmek . . . . .	223
16.4	Karakter Dizilerini Dilimlemek . . . . .	226
16.5	Karakter Dizilerinin Metotları . . . . .	229
16.6	Bölüm Soruları . . . . .	256
17	Biçim Düzenleyiciler . . . . .	259
17.1	Biçim Düzenlemede Kullanılan Karakterler . . . . .	260
17.2	İleri Düzeyde Karakter Dizisi Biçimlendirme . . . . .	264
17.3	Bölüm Soruları . . . . .	269
18	ascii, unicode ve Python . . . . .	270
18.1	Giriş . . . . .	270
18.2	ascii . . . . .	271
18.3	unicode . . . . .	275
18.4	Python'da unicode Desteği . . . . .	279
19	Düzenli İfadeler (Regular Expressions) . . . . .	289
19.1	Düzenli İfadelerin Metotları . . . . .	290
19.2	Metakarakterler . . . . .	296
19.3	Eşleşme Nesnelerinin Metotları . . . . .	312
19.4	Düzenli İfadelerin Derlenmesi . . . . .	316
19.5	compile() ile Derleme Seçenekleri . . . . .	317
19.6	Düzenli İfadelerle Metin/Karakter Dizisi Değiştirme İşlemleri . . . . .	319
19.7	Sonuç . . . . .	323
20	Nesne Tabanlı Programlama – OOP (NTP) . . . . .	325
20.1	Neden Nesne Tabanlı Programlama? . . . . .	325
20.2	Sınıflar . . . . .	326
20.3	Örneklemeye (Instantiation) . . . . .	328
20.4	Çöp Toplama (Garbage Collection) . . . . .	329
20.5	Niteliklere Değınme (Attribute References) . . . . .	330
20.6	__init__ Nedir? . . . . .	332
20.7	self Nedir? . . . . .	334
20.8	Miras Alma (Inheritance) . . . . .	340
20.9	Eski ve Yeni Sınıflar . . . . .	346
20.10	Sonuç . . . . .	348
21	Sqlite ile Veritabanı Programlama . . . . .	349
21.1	Giriş . . . . .	349
21.2	Neden Sqlite? . . . . .	350
21.3	Sqlite'in Yapısı . . . . .	350
21.4	Veritabanıyla Bağlantı Kurmak . . . . .	351
21.5	Veri Girişi . . . . .	352
21.6	Veri İşleme - commit() Metodu . . . . .	354
21.7	Veritabanından Veri Almak . . . . .	355
21.8	Veritabanı Güvenliği - SQL Injection . . . . .	357

22	sys Modülü	362
22.1	sys Modülünün İçeriği	362
22.2	argv Niteliği	363
22.3	exit() Fonksiyonu	366
22.4	getdefaultencoding() Fonksiyonu	367
22.5	path Niteliği	367
22.6	platform Niteliği	368
22.7	stdout Niteliği	369
22.8	version_info Niteliği	373
23	math Modülü	374
23.1	Üslü İfadeler Fonksiyonu (pow)	374
23.2	Pi Niteliği (pi)	375
23.3	Karekök Fonksiyonu (sqrt)	375
23.4	Euler Sabiti (e)	375
23.5	exp Fonksiyonu	376
23.6	Logaritma (log) Fonksiyonu	376
23.7	Logaritma (log10) Fonksiyonu	376
23.8	Degrees Fonksiyonu	376
23.9	Radians Fonksiyonu	377
23.10	Kosinüs Fonksiyonu (cos)	377
23.11	Sinüs Fonksiyonu (sin)	378
23.12	Tanjant (tan) Fonksiyonu	378
23.13	Faktoriyel (factorial)	379
23.14	fmod Fonksiyonu	380
23.15	ceil Fonksiyonu	380
23.16	floor Fonksiyonu	381
23.17	fabs Fonksiyonu	381
23.18	frexp Fonksiyonu	381
23.19	ldexp Fonksiyonu	381
23.20	modf Fonksiyonu	382
23.21	trunc Fonksiyonu	382
23.22	hypot Fonksiyonu	382
23.23	Hiperbolik Fonksiyonlar	383
24	datetime Modülü	385
24.1	Bugünün Tarihini Bulmak	385
24.2	Bir Tarihin Hangi Güne Geldiğini Bulmak	386
24.3	Tarihleri Biçimlendirmek	388
24.4	Tarihlerle Aritmetik İşlem Yapmak	390
25	time Modülü	391
25.1	sleep() Fonksiyonu	391
25.2	strftime() Fonksiyonu	393
25.3	localtime() Fonksiyonu	395
25.4	gmtime(), time() ve ctime() Fonksiyonları	396

26	len() Fonksiyonu ve ascii'nin Laneti	398
27	Python'da id() Fonksiyonu, is İşleci ve Önbellekleme Mekanizması	404
28	Windows'ta Python'ı YOL'a (PATH) Ekleme	409
29	PEP 3000	416
29.1	Özet	416
29.2	Adlandırma	416
29.3	PEP'lerin Numaralandırılması	417
29.4	Takvim	417
29.5	Uyumluluk ve Geçiş	417
29.6	Gerçekleme Dili (Implementation)	418
29.7	Üst-Katkılar	418
29.8	Kaynaklar	418
29.9	Telif Hakkı	419
30	Python ve OpenOffice	420
30.1	PyUno'nun Kurulumu	421
30.2	OpenOffice'yi Dinleme Kipinde Açmak (listening mode)	422
30.3	OpenOffice'ye Bağlanmak	424
30.4	Karakter Biçimlendirme	427
30.5	Karakterleri Renklendirme	430
30.6	PyUno, Python ve OpenOffice Hakkında Bilgi Veren Kaynaklar	432
31	Python'da Paket Kurulumu - Kullanımı	433
32	reStructuredText	436
32.1	Giriş	436
32.2	RestructuredText Dosyaları ve rst2html Betiği	438
32.3	Paragraf Stilleri	439
32.4	Listeler	441
32.5	Köprüler/Bağlantılar	444
32.6	Değişkenler	445
32.7	Kod Alanları	446
32.8	Dipnotları	447
32.9	Resimler	447
32.10	Yorumlar	448
32.11	Alıntılar	448
32.12	Tablolar	448
33	Sphinx	450
33.1	Sphinx Hakkında	450
33.2	Sphinx Nasıl Kurulur?	451
33.3	Sphinx Nasıl Kullanılır?	453
33.4	conf.py Dosyası	456
33.5	index Dosyası	458



33.6 Özel Yönergeler (Directives) . . . . .	460
Dizin	465



**Uyarı:** Aşağıdaki bilgiler Python'un 2.x sürümleri içindir. Eğer kullandığınız sürüm Python'un 3.x sürümlerinden biriyse [şuradaki](#) belgeleri inceleyebilirsiniz.

## TEMEL KONULAR

---

# Temel Bilgiler

---

Python 1990 yılından bu yana büyük bir topluluk tarafından geliştirilen kıvrak ve dinamik bir dildir. Söz diziminin sade olması, kolay öğrenilmesi ve program geliştirme sürecini hızlandırması ile tanınan bu dil Windows, GNU/Linux ve MacOS X gibi pek çok farklı işletim sistemi üzerinde çalışabilmektedir. Dolayısıyla tek bir platformda geliştirdiğiniz bir Python uygulaması, üzerinde hiç bir değişiklik yapmaya gerek olmadan veya küçük değişikliklerle başka platformlarda da çalışabilecektir.

Son yıllarda Türkiye’de de ilgi uyandırmaya başlayan bu dil yavaş yavaş üniversite müfredatındaki yerini alıyor. Her ne kadar ülkemizde henüz yeterli ilgiyi görmese de, bugün Google’dan NASA’ya, YouTube’dan Hewlett Packard’a kadar pek çok kurum ve şirkette Python programlama dilinden yararlanıldığını görüyoruz.

Burada, Python programlama diline merak duyan okurlara bu dilin temellerini sağlam ve hızlı bir şekilde öğretmeyi amaçlıyoruz. Bu belgelerden yararlanabilmek için önceden bir programlama dili biliyor olmanız gerekmez. Buradaki bilgiler, okurun programlama konusunda hiç bir bilgisi olmadığı varsayılarak ve herkesin anlayabileceği bir üslupla hazırlanmıştır.

## 1.1 Python Hakkında

Python’ın baş geliştiricisi Guido Van Rossum adlı Hollandalı bir programcıdır. Eğer Guido Van Rossum’un neye benzediğini merak ediyorsanız, onun <http://www.python.org/~guido/pics.html> adresindeki fotoğraflarını inceleyebilirsiniz.

Her ne kadar Python programlama dili ile ilgili çoğu görsel malzemenin üzerinde bir piton resmi görsek de, Python kelimesi aslında çoğu kişinin zannettiğinin aksine piton yılanı anlamına gelmiyor. Python programlama dili, Guido Van Rossum’un çok sevdiği, **Monty Python** adlı altı kişilik bir İngiliz komedi grubunun *Monty Python’s Flying Circus* adlı gösterisinden alıyor ismini.

Python programlama dili, C ve C++ gibi daha yerleşik ve köklü dillere kıyasla çok uzun bir geçmişe sahip değildir, ama bu programlama dillerine göre hem daha kolaydır hem de program geliştirme sürecini bir hayli kısaltır. Üstelik bu dil ayrı bir derleyiciye de ihtiyaç duymaz. Ayrıca bu dilde yazılan kodlar başka dillere göre hem daha okunaklı hem de daha temizdir...

İşte Python, bu ve buna benzer üstünlükleri sayesinde pek çok kimsenin gözdesi haline gelmiştir. Google’ın da Python’a özel bir önem ve değer verdiğini, çok iyi derecede Python bilenlere iş olanağı sunduğunu hemen söyleyelim. Mesela Python’ın geliştiricisi Guido

Van Rossum 2005 yılının Aralık ayından beri Google'da çalışıyor. (Guido Van Rossum'un özgeçmişine <http://www.python.org/~guido/Resume.html> adresinden erişebilirsiniz.)

## 1.2 Python Nasıl Okunur?

Geliştiricisi Hollandalı olsa da *Python* İngilizce bir kelimedir. Dolayısıyla bu kelimenin telaffuzunda İngilizce'nin kuralları geçerli. Ancak bu kelimeyi hakkıyla telaffuz etmek, ana dili Türkçe olanlar için pek kolay değil. Çünkü bu kelime içinde, Türkçe'de yer almayan ve okunuşu peltek s'yi andıran **th** sesi var. İngilizce bilenler bu sesi *think* [düşünmek] kelimesinden hatırlayacaklardır. Ana dili Türkçe olanlar *think* kelimesini genellikle **tink** şeklinde telaffuz eder. Dolayısıyla *Python* kelimesini de **paytın** şeklinde telaffuz edebilirsiniz...

Asıl söylenişinin dışında bu kelimeyi tamamen Türkçeleştirerek **piton** şeklinde telaffuz edenler de var. Elbette siz de dilinizin döndüğü bir telaffuzu tercih etmekte özgürsünüz.

Eğer "python" kelimesinin İngilizce telaffuzunu dinlemek istiyorsanız <http://www.howjsay.com/> adresini ziyaret edebilir, Guido Van Rossum'un bu kelimeyi nasıl telaffuz ettiğini merak ediyorsanız da <http://video.google.com/videoplay?docid=-6459339159268485356#> adresindeki tanıtım videosunu izleyebilirsiniz.

## 1.3 Python Nereden İndirilir?

Python'ı kullanabilmek için, bu programlama dilinin sistemimizde kurulu olması gerekiyor. İşte biz de bu bölümde Python'ı nereden indirip sistemimize nasıl kuracağımızı öğreneceğiz.

Python Windows ve GNU/Linux işletim sistemlerine kurulma açısından farklılıklar gösterir. Biz burada Python'ın hem GNU/Linux'a hem de Windows'a nasıl kurulacağını ayrı ayrı inceleyeceğiz. Ancak her ne kadar GNU/Linux ve Windows bölümlerini ayırmış da olsak, hangi işletim sistemini kullanıyor olursanız olun, ben size her iki bölümü de okumanızı tavsiye ederim. Çünkü GNU/Linux bölümünde Windows kullanıcılarının, Windows bölümünde ise GNU/Linux kullanıcılarının ilgisini çekebilecek kısımlar olabilir. Ayrıca yazdığınız bir programın sadece tek bir işletim sistemine bağımlı olmaması önemli bir özelliktir. O yüzden hem Windows hem de GNU/Linux hakkında bilgi sahibi olmanın zararını değil, faydasını göreceksiniz.

Öncelikle GNU/Linux'tan başlayalım:

### 1.3.1 GNU/Linux'ta Python'ı Kurmak

Python hemen hemen bütün GNU/Linux dağıtımlarında kurulu geliyor. Mesela Pardus ve Ubuntu'da Python'ın kurulu olduğunu biliyoruz, o yüzden Pardus veya Ubuntu kullanıyorsanız Python'ı kurmanıza gerek yok.

Eğer Python'ı kurmanız gerekirse <http://www.python.org/download> adresinden Python'ın kaynak dosyalarını indirebilirsiniz. Ancak Python GNU/Linux dağıtımlarında çok önemli bazı parçalarla etkileşim halinde olduğu için kaynaktan derleme pek tavsiye edilmez. Hele ki Pardus gibi, sistemin belkemiğini Python'ın oluşturduğu bir dağıtımda Python'ı kaynaktan derlemeye çalışmak, eğer dikkatsiz davranırsanız epeyce başınızı ağrıtabilir. Sözün özü, GNU/Linux sistemlerinde en kestirme yol dağıtımın kendi Python paketlerini kullanmaktır.

Ancak GNU/Linux sistemlerinde kurulu olarak gelen Python çoğunlukla en yeni sürüm değildir. Eğer mutlaka ama mutlaka Python'ın daha yeni bir sürümüne ihtiyacınız yoksa eski de olsa dağıtımınızla birlikte gelen Python sürümünü kullanmanızı öneririm. Ama eğer, "*Ben illa ki en*

*yeni sürümü kullanacağım!"* dersiniz, kullandığınız GNU/Linux dağıtımına, sisteminizdekinden daha yeni (veya daha eski) bir Python sürümü kurmanız da elbette mümkündür. Bunun için;

- Öncelikle şu adresi ziyaret ediyoruz: <http://www.python.org/download>
- Bu adreste, üzerinde *Python 2.x.x compressed source tarball (for Linux, Unix or OS X)* yazan bağlantıya tıklayarak ilgili .tgz dosyasını bilgisayarımıza indiriyoruz.
- Daha sonra bu sıkıştırılmış dosyayı açıyoruz ve açılan dosyanın içine girip, orada sırasıyla aşağıdaki komutları veriyoruz:

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

Ancak bir noktaya dikkatinizi çekmek isterim: Python'ın düzgün kurulabilmesi için `make altinstall` komutunu yetkili kullanıcı veya *root* olarak çalıştırmalısınız.

Eğer her şey yolunda gittiyse Python'ın farklı bir sürümü sistemimize kurulmuş oldu. Yalnız burada `make install` yerine `make altinstall` komutunu kullandığımıza dikkat edin. `make altinstall` komutu, Python kurulurken klasör ve dosyalara sürüm numarasının da eklenmesini sağlar. Böylece yeni kurduğunuz Python, sistemdeki eski Python sürümünü silip üzerine yazmamış olur ve iki farklı sürüm yan yana varolabilir. Bu önemli ayrıntıyı kesinlikle gözden kaçırmamalısınız.

Bu noktada bir uyarı yapmadan geçmeyelim: Daha önce de dediğimiz gibi, Python özellikle bazı GNU/Linux dağıtımlarında pek çok sistem aracıyla sıkı sıkıya bağlantılıdır. Yani Python, kullandığınız dağıtımın belkemiği durumunda olabilir. Bu yüzden Python'ı kaynaktan derlemek bazı riskler taşıyabilir. Eğer yukarıda anlatıldığı şekilde, kaynaktan Python derleyecekseniz, karşı karşıya olduğunuz risklerin farkında olmalısınız. Ayrıca GNU/Linux üzerinde kaynaktan program derlemek konusunda tecrübeli değilseniz ve eğer yukarıdaki açıklamalar size kafa karıştırıcı geliyorsa, kesinlikle dağıtımınızla birlikte gelen Python sürümünü kullanmalısınız. Python sürümlerini başa baş takip ettiği için, ben size Ubuntu GNU/Linux'u denemenizi önerebilirim. Ubuntu'nun depolarında Python'ın en yeni sürümlerini rahatlıkla bulabilirsiniz. Ubuntu'nun resmi sitesine <http://www.ubuntu.com> adresinden, yerel Türkiye sitesine ise <http://www.ubuntu.org.tr> adresinden ulaşabilirsiniz.

### 1.3.2 Windows'ta Python'ı Kurmak

GNU/Linux dağıtımlarının aksine, Windows işletim sisteminde Python programlama dili kurulu olarak gelmez. Dolayısıyla Python'ı Windows'ta kullanabilmek için bu programı sitesinden [<http://www.python.org>] indirmemiz gerekiyor. Resmi sitedeki indirme adresinde [<http://www.python.org/download>] programın Microsoft Windows işletim sistemiyle uyumlu sürümlerini bulabilirsiniz. Bu adresten Python'ı indirmek isteyen çoğu Windows kullanıcısı için en uygunu, üzerinde *Python 2.x.x Windows installer (Windows binary – does not include source)* yazan bağlantıya tıklamak olacaktır.

Windows kullanıcıları resmi sitedeki indirme adresinde yer alan Python kurulum betiğini bilgisayarlarına indirdikten sonra kurulum dosyasına çift tıklayarak ve ekrandaki yönergeleri takip ederek Python'ı kurabilirler. Python'ı kurmak çok kolaydır. Python'ı kurarken, öntanımlı ayarlarda herhangi bir değişiklik yapmadan, sadece *Next* tuşlarına basarak kurulumu gerçekleştirebilirsiniz. Python otomatik olarak *C:\Python2x* dizini içine kurulacaktır.

Eğer Python programlama dilinin hangi sürümünü kullanmanız gerektiği konusunda kararsızlık yaşıyorsanız, ben size 2.6 sürümünü tavsiye ederim. Aslında 2.5 ve üstü bütün sürüm-

ler kullanıma uygundur. Ancak Python'ın 2.6 sürümü şu anda mevcut sürümler arasında en olgunudur. Biz burada konuları anlatırken Python'ın 2.6 sürümünü temel alacak olsak da Python'ın başka sürümlerini kullananlar da buradaki belgelerden faydalanabilir.

Yeri gelmişken önemli bir uyarıda bulunalım: Python'ın 2.x numaralı sürümleri ile 3.x numaralı sürümleri birbirinden farklıdır. Eğer Python'ın 3.x sürümlerinden birini kullanmak istiyorsanız [http://www.istihza.com/py3/icindekiler\\_python.html](http://www.istihza.com/py3/icindekiler_python.html) adresindeki belgelerle çalışabilirsiniz.

## 1.4 Python Nasıl Çalıştırılır?

Bu bölümde hem GNU/Linux, hem de Windows kullanıcılarının Python'ı nasıl çalıştırması gerektiğini tartışacağız. Öncelikle GNU/Linux kullanıcılarından başlayalım.

### 1.4.1 GNU/Linux'ta Python'ı Çalıştırmak

Eğer GNU/Linux işletim sistemi üzerinde KDE kullanıyorsak Python programını çalıştırmak için önce ALT+F2 tuşlarına basıp, çıkan ekranda şu komutu vererek bir konsol ekranı açıyoruz:

```
konsole
```

Eğer kullandığımız masaüstü GNOME ise ALT+F2 tuşlarına bastıktan sonra vermemiz gereken komut şudur:

```
gnome-terminal
```

Bu şekilde komut satırına ulaştığımızda;

```
python
```

yazıp ENTER tuşuna basarak Python programlama dilini başlatıyoruz. Karşımıza şuna benzer bir ekran gelmeli:

```
Python 2.6.5 (r265:79063, Apr 3 2010, 01:57:29)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license"
for more information.
>>>
```

Bu ekranda kullandığımız Python sürümünün 2.6.5 olduğunu görüyoruz. Buradaki >>> işareti Python'ın bizden komut almaya hazır olduğunu gösteriyor. Komutlarımızı bu işareten hemen sonra, **boşluk bırakmadan** yazacağız. Bunun dışında, istersek Python kodlarını bir metin dosyasına da kaydedebilir, bu kaydettiğimiz metin dosyasını konsoldan çalıştırabiliriz. Bu işlemin nasıl yapılacağını daha sonra konuşacağız.

Eğer python komutunu verdiğinizde yukarıdaki ekran yerine bir hata mesajıyla karşılaşıyorsanız iki ihtimal var:

Birincisi, "python" kelimesini yanlış yazmış olabilirsiniz. Mesela yanlışlıkla "pyhton", "pyton", "phyton" veya "Python" yazmış olabilirsiniz. Doğru kelimenin tamamen küçük harflerden oluştuğuna özellikle dikkat etmemiz gerekiyor. Windows açısından "python" kelimesini büyük veya küçük harflerle yazmanızın bir önemi olmayabilir, ama GNU/Linux açısından büyük-küçük harf ayrımı son derece önemlidir.

İkincisi, eğer ilk maddede söylenenlerin geçerli olmadığından eminseniz, çok düşük bir ihtimal olmakla birlikte, Python sisteminizde kurulu değil demektir. Yalnız GNU/Linux sistemlerinde Python'ın kurulu olmama ihtimalinin sıfıra yakın olduğunu söyleyeyim. O yüzden sisteminizde

Python'ın kurulu olmadığına kesinkes karar vermeden önce, durumunuzun birinci madde kapsamına girmediğinden emin olmalısınız.

Eğer kullandığınız GNU/Linux dağıtımında Python'ın kurulu olmadığına eminseniz, önünüzde iki seçenek var:

Birincisi, Python o anda sisteminizde kurulu olmasa bile, kullandığınız dağıtımın paket depolarında olabilir. O yüzden öncelikle dağıtımınızın paket depolarında "python" kelimesini kullanarak bir arama yapın ve kullandığınız dağıtıma uygun bir şekilde Python paketini kurun.

İkincisi, Python kullandığınız dağıtımın paket depolarında bulunsa bile, siz Python'ı kaynaktan derlemek istiyor olabilirsiniz. Eğer öyleyse <http://www.python.org/download> adresinden "Python 2.x.x compressed source tarball (for Linux, Unix or OS X)" bağlantısına tıklayarak, .tgz dosyasını bilgisayarınıza indirin ve klasörü açıp orada sırasıyla ./configure, make ve make install komutlarını verin. Burada farklı olarak make altinstall yerine make install komutunu kullandığımıza dikkat edin. Çünkü sizin sisteminizde Python'ın hiç bir sürümü kurulu olmadığı için, elle kuracağınız yeni sürümün eski bir sürümle çakışma riski de yok. O yüzden make altinstall yerine doğrudan make install komutunu kullanabilirsiniz. Ancak daha önce de dediğimiz gibi, biz size dağıtımınızın depolarında bulunan Python sürümünü kullanmanızı şiddetle tavsiye ediyoruz.

Gelelim Microsoft Windows kullanıcılarına...

## 1.4.2 Windows'ta Python'ı Çalıştırmak

Python'ı yukarıda verdiğimiz indirme adresinden indirip bilgisayarlarına kurmuş olan Windows kullanıcıları, *Başlat/Programlar/Python 2.x/Python (Command Line)* yolunu takip ederek Python'ın komut satırına ulaşabilirler.

Ayrıca alternatif olarak, *Başlat/Çalıştır* yolunu takip ederek, cmd komutuyla ulaştığınız MS-DOS ekranında şu komutu verdiğinizde de karşınıza Python'ın komut satırı gelecektir:

```
c:/python26/python
```

Eğer yukarıda yaptığımız gibi uzun uzun konum belirtmek yerine sadece python komutunu kullanmak isterseniz Python'ı YOL'a (PATH) eklemeniz gerekir. Peki Python'ı nasıl YOL'a ekleyeceğiz? Şöyle:

- Denetim Masası içinde "Sistem" simgesine çift tıklayın. (Eğer klasik görünümde değilseniz Sistem simgesini bulmak için "Performans ve Bakım" kategorisinin içine bakın veya Denetim Masası açıkken adres çubuğuna doğrudan "sistem" yazıp ENTER tuşuna basın.)
- "Gelişmiş" sekmesine girin ve "Ortam Değişkenleri" düğmesine basın.
- "Sistem Değişkenleri" bölümünde "Path" öğesini bulup buna çift tıklayın.
- "Değişken Değeri" ifadesinin hemen karşısındaki kutucuğun en sonuna şu girdiyi ekleyin: ;C:\Python26.
- TAMAM'a basıp çıkın.
- Bu değişikliklerin geçerlilik kazanabilmesi için açık olan bütün MS-DOS pencerelerini kapatıp yeniden açın.

---

**Not:** Eğer bu işlemler size karışık geldiyse, resimli ve daha ayrıntılı bir anlatım için [Windows'ta Python'ı YOL'a Ekleme](#) adlı makalemizi inceleyebilirsiniz.

---



Eğer yukarıdaki işlemleri başarıyla gerçekleştirdiyseniz, *Başlat/Çalıştır* yolunu takip edip cmd komutunu vererek ulaştığınız MS-DOS ekranında;

```
python
```

yazıp ENTER tuşuna bastığınızda karşınıza şöyle bir ekran geliyor olmalı:

```
Python 2.6.5 (r265:79096, Mar 19 2010, 21:48:26)
[MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license"
for more information.
>>>
```

Bu ekranda kullandığımız Python sürümünün 2.6.5 olduğunu görüyoruz. Buradaki >>> işareti Python'ın bizden komut almaya hazır olduğunu gösteriyor. Komutlarımızı bu işaretten hemen sonra, **boşluk bırakmadan** yazacağız. Bunun dışında, istersek Python kodlarını bir metin dosyasına da kaydedebilir, bu kaydettiğimiz metin dosyasını komut satırından çalıştırabiliriz. Bu işlemin nasıl yapılacağını daha sonra konuşacağız.

Eğer python komutunu verdiğinizde yukarıdaki ekran yerine bir hata mesajıyla karşılaşıyorsanız üç ihtimal var:

1. "python" kelimesini yanlış yazmış olabilirsiniz. Mesela yanlışlıkla "pyhton", "pyton" veya "phyton" yazmış olabilirsiniz. Bu yüzden "python" kelimesini doğru yazdığınıza emin olun.
2. Python'ı YOL'a ekleyememiş olabilirsiniz. Eğer yukarıda anlattığımız YOL'a ekleme adımlarını uygulamak konusunda sıkıntı yaşıyorsanız, kistihza [at] yahoo [nokta] com adresinden bana ulaşabilirsiniz. Ben size elimden geldiğince yardımcı olmaya çalışırım.
3. Python'ı kuramamış olabilirsiniz. *Başlat/Programlar* yolu içinde bir "Python26" girdisi olup olmadığına bakın. Ayrıca C:\ dizininin içini de kontrol edin. Orada *Python26* adlı bir klasör görüyor olmalısınız. Eğer programlar listesinde veya C:\ dizini içinde "Python26" diye bir şey yoksa Python'ı kuramamışsınız demektir. Bu durumda Python'ı yeniden kurmayı deneyebilirsiniz.

Ben sizin yukarıdaki komutu düzgün bir şekilde çalıştırabilmiş olduğunuzu varsayıyorum.

## 1.5 Python'dan Nasıl Çıkılır?

Farklı işletim sistemlerinde python komutunu vererek Python'ın komut satırına nasıl erişebileceğimizi öğrendik. Peki bu komut satırından çıkmak istersek ne yapacağız? Elbette doğrudan komut penceresi üzerindeki çarpı tuşuna basarak bu ortamı terk edebilirsiniz. Ancak bu işlemi kaba kuvvete başvurmadan yapmanın bir yolu olmalı, değil mi?

Komut satırından çıkmanın birkaç farklı yolu vardır:

1. Komut ekranı üzerindeki çarpı düğmesine basmak (kaba kuvvet)
2. Önce CTRL+Z tuşlarına, ardından da ENTER tuşuna basmak (Windows)
3. CTRL+Z tuşlarına basmak (GNU/Linux)
4. `quit()` yazıp ENTER tuşuna basmak (Bütün işletim sistemleri)
5. Önce `import sys`, ardından da `sys.exit()` komutlarını vermek (Bütün işletim sistemleri)

Siz bu farklı yöntemler arasından, kolayınıza hangisi geliyorsa onu seçebilirsiniz.

Böylece Python'ı nereden indireceğimizi, nasıl kuracağımızı ve Python'ın komut satırını nasıl başlatıp kapatabileceğimizi öğrenmiş olduk. Python'a ilişkin en temel bilgileri edinmiş olduğumuza göre artık Python'la daha ciddi işler yapmaya doğru ilk adımlarımızı atabiliriz. Ama önce bölüm sorularına bakalım.

## 1.6 Bölüm Soruları

1. Python'ın GNU/Linux, Windows ve Mac OS X sürümleri olduğunu biliyoruz. <http://www.python.org/download> adresini ziyaret ederek, Python'ın başka hangi platformlara ait sürümlerinin olduğunu inceleyin. Sizce Python'ın bu kadar farklı işletim sistemi ve platform üzerinde çalışabiliyor olması bu dilin hangi özelliğini gösteriyor?
2. Eğer GNU/Linux dağıtımlarından birini kullanıyorsanız, sisteminizde Python programlama dilinin kurulu olup olmadığını denetleyin. Kullandığınız dağıtımda Python kurulumla birlikte mi geliyor, yoksa başka bir paketin bağımlılığı olarak mı sisteme kuruluyor? Eğer Python kurulumla birlikte geliyorsa, kurulu gelen, Python'ın hangi sürümü? Dağıtımınızın depolarındaki en yeni Python sürümü hangisi?
3. Tercihen VirtualBox gibi bir sanallaştırma aracı ile kurduğunuz bir GNU/Linux dağıtımı üzerinde Python kurulumuna ilişkin bazı denemeler yapın. Örneğin Python'ın resmi sitesinden dilin kaynak kodlarını indirip programı kaynaktan derleyin. Sistemde kurulu olarak gelen Python sürümüyle, sizin kaynaktan derlediğiniz Python sürümünün birbiriyle çalışmaması için gerekli önlemleri alın. Diyelim ki sisteminizde Python'ın 2.6 sürümü var. Siz Python'ın sitesinden farklı bir Python sürümü indirdiğinizde Python'ın öntanımlı sürümüne ve kaynaktan derlenen sürümüne ayrı ayrı nasıl ulaşabileceğinizi düşünün.
4. Eğer siz bir Windows kullanıcısıysanız ve .Net çatısı ile aşinalığınız varsa IronPython'ın ne olduğunu araştırın.
5. Eğer siz bir Java programcısı iseniz Jython'ın ne olduğunu araştırın.
6. Kullandığınız işletim sisteminde Python'ı kaç farklı biçimde çalıştırabildiğinizi kontrol edin.
7. Windows'ta Python'ın hangi araçlarla birlikte kurulduğunu kontrol edin. Kurulumla birlikte gelen çevrimdışı İngilizce kılavuzları inceleyin. Localhost'tan hizmet veren "pydoc" (*Module Docs*) sunucusunu çalıştırın ve bunun ne işe yaradığını anlamaya çalışın.
8. Windows'ta YOL (PATH) yapısını inceleyin. Windows dizinleri YOL'a nasıl ekleniyor? YOL'a eklenen dizinler birbirinden hangi işaret ile ayrılıyor? Bir dizinin YOL üstünde olup olmaması neyi değiştiriyor? Sitesinden indirip kurduğunuz Python sürümünü YOL'a eklemeyi deneyin. Bu işlem sırasında ne gibi sorunlarla karşılaştığınızı değerlendirin.

---

# Python'a Giriş

---

Bir önceki bölümde Python'ın komut satırına nasıl ulaşacağımızı görmüştük. Bu komut satırına teknik dilde **etkileşimli kabuk** (*interactive shell*) veya **yorumlayıcı** (*interpreter*) adı verilir.

Şimdi önceki bölümde anlattığımız yöntemlerden herhangi birini kullanarak Python'ın etkileşimli kabuğunu açalım ve şuna benzer bir ekranla karşılaşalım:

```
istihza@istihza:~$ python
Python 2.6.5 (r265:79063, Apr  3 2010, 01:57:29)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license"
for more information.
>>>
```

Komutlarımızı bu >>> işaretinden hemen sonra, hiç boşluk bırakmadan yazmaya başlayacağımızı daha önce söylemiştik. Şimdi isterseniz birkaç parmak alıştırmayı yapıp kendimizi Python'a ısındıralım.

Bu bölümde inceleyeceğimiz ilk komutumuzun adı *print*.

## 2.1 print Komutu

*print* komutunun görevi, ekrana bir şeyler yazdırmamızı sağlamaktır. Mesela bu komutu tek başına kullanmayı deneyelim:

```
>>> print
```

yazıp hemen ENTER tuşuna basalım.

**Not:** Yukarıdaki komutta gördüğümüz >>> işaretini kendimiz yazmayacağız. O işaret, etkileşimli kabuğun görüntüsünü temsil ediyor. Bizim sadece *print* yazmamız yeterli.

Ne oldu? Python bir satır boşluk bırakarak alt satıra geçti, değil mi? Bunu ona yapmasını biz söyledik, o da yaptı...

Şimdi de boş bir satır bırakmak yerine ekrana bir şeyler yazmasını söyleyelim Python'a:

```
>>> print "Programa hoşgeldiniz. Lütfen giriş yapın!"
```

Bu satırı yazıp ENTER tuşuna bastıktan sonra ekranda *Programa hoşgeldiniz. Lütfen giriş yapın!* çıktısını görmemiz gerekiyor.

Gördüğünüz gibi *print* komutunun ardından gelen "Programa hoşgeldiniz. Lütfen giriş yapın!" ifadesini çift tırnak içinde belirtiyoruz. Eğer burada çift tırnak işaretini koymazsak veya koymayı unutursak Python bize bir hata çıktısı gösterecektir. Zaten yukarıdaki komutu verdikten sonra bir hata alıyorsanız, çok büyük bir ihtimalle tırnakları düzgün yerleştirmemişsinizdir.

---

**Not:** Eğer yukarıdaki komutu doğru yazdığınız halde hata alıyorsanız, muhtemelen kullandığınız Python sürümü 2.x değil, 3.x'tir. Eğer öyleyse, [http://www.istihza.com/py3/icindekiler\\_python.html](http://www.istihza.com/py3/icindekiler_python.html) adresindeki Python 3.x belgelerinden yararlanabilirsiniz.

---

Biz istersek yukarıdaki örnekte çift tırnak yerine tek tırnak (') da kullanabiliriz. Tırnak seçiminde özgürüz:

```
>>> print 'Programa hoşgeldiniz. Lütfen giriş yapın!'
```

Bu komut da bize istediğimiz çıktıyı başarıyla verecektir. Dediğimiz gibi, kullanacağınız tırnak tipi konusunda özgürsünüz. Ancak tırnak seçiminde bazı noktalara da dikkat etmemiz gerekiyor. Diyelim ki "İstanbul'un 5 günlük hava tahmini" ifadesini ekrana yazdırmak istiyoruz. Bunu çift tırnak kullanarak şu şekilde gösterebiliriz:

```
>>> print "İstanbul'un 5 günlük hava tahmini"
```

Bu komut bize hatasız bir şekilde *İstanbul'un 5 günlük hava tahmini* çıktısını verir. Ancak aynı işlemi tek tırnakla yapmaya çalışırsak şöyle bir hata mesajı alırız:

```
File "<stdin>", line 1
  print 'İstanbul'un 5 günlük hava tahmini'
        ^
SyntaxError: invalid syntax
```

Bunun sebebi, "İstanbul'un" kelimesindeki kesme işaretinden ötürü Python'ın tırnakların nerede başlayıp nerede bittiğini anlamamasıdır. Eğer mutlaka tek tırnak kullanmak istiyorsak, kodu şu hale getirmemiz gerekir:

```
>>> print 'İstanbul\'un 5 günlük hava tahmini'
```

Aynı şekilde, şöyle bir örnekte de çift tırnak bize zorluk çıkarır:

```
>>> print "Python'da önemli komutlardan biri de "print" komutudur."
```

Bu komut bize şöyle bir hata mesajı verecektir:

```
File "<stdin>", line 1
  print "Python'daki önemli komutlardan biri de "print" komutudur."
                                                ^
SyntaxError: invalid syntax
```

Bu cümleyi düzgün olarak gösterebilmek için yukarıdaki komutu şöyle yazmamız gerekir:

```
>>> print "Python'da önemli komutlardan biri de \"print\" komutudur."
```

Yukarıdaki iki örnekte gördüğümüz “\” işaretleri, kendilerinden sonra gelen karakterlerin Python tarafından normalden farklı algılanmasını sağlar. Yani bu işaret sayesinde, cümle içinde geçen tırnak işaretleri, cümleyi başlatan ve bitiren tırnak işaretleriyle karışmaz. Buna benzer hatalardan kaçmamızı sağlayan bu tür işaretlere Python dilinde Kaçış Dizileri (*Escape Sequences*) adı verilir. Şimdilik bu kaçış dizisi kavramını çok fazla kafamıza takmadan yolumuza devam edelim. Zaten birkaç bölüm sonra bu kaçış dizilerinden ayrıntılı olarak söz edeceğiz. Biz şu anda sadece, cümle içinde geçen tırnak işaretlerine dikkat etmemiz gerektiğini bilelim yeter.

Dediğimiz gibi, *print* komutu Python’daki en önemli ve en temel komutlardan biridir. Python’la yazdığınız programlarda kullanıcılarınıza herhangi bir mesaj göstermek istediğinizde bu *print* komutundan yararlanacaksınız. Dilerseniz bu önemli komutla birkaç örnek daha yapalım elimizi alıştırmak için:

```
>>> print "GAME OVER"

GAME OVER

>>> print "FINISH HIM!"

FINISH HIM!

>>> print "Linux’un güçlü yanları"

Linux’un güçlü yanları

>>> print 'Linux\'un güçlü yanları'

Linux'un güçlü yanları
```

Yukarıdaki son örnekte “\” işaretini neden kullandığımızı biliyorsunuz. “Linux’un güçlü yanları” cümlesini tek tırnak içine aldığımız için, **Linux’un** kelimesindeki kesme işaretinden (') ötürü Python tırnakların nerede başlayıp nerede bittiğini anlayamaz. Python’ın kafasını karıştırmamak için burada kaçış dizilerinden yararlanarak düzgün çıktı almayı sağlıyoruz. Yani **Linux’un** kelimesindeki kesme işaretiyle karşılaşan Python’a şöyle demiş oluyoruz: “*Bu tırnak senin aradığın tırnak değil. Sen yoluna devam et. Biraz ilerde aradığın tırnağı bulacaksın!*”

## 2.2 Karakter Dizileri

Python’daki en önemli komutlardan biri olan *print*’i öğrendiğimize göre, bundan sonraki konuları daha rahat anlayabilmemiz için bazı kavramlardan söz etmemiz faydalı olabilir. Bu bölümde “karakter dizisi” diye bir şeyden söz edeceğiz. Yabancılar buna *string* adı veriyor...

Aslında biz karakter dizisinin ne olduğunu biliyoruz. Yukarıda *print* komutunu işlerken bu karakter dizilerini zaten kullandık. Oradaki örneklerden de göreceğiniz gibi, karakter dizileri, tırnak içinde gösterilen herhangi bir “şey”dir. Tanımından da anlaşılacağı gibi, tırnak içinde gösterebildiğimiz her şey bir karakter dizisi olabilir. Mesela şu örnekler birer karakter dizisidir:

```
"Python"

"Perl"

"Ruby"

"23"
```

```
"35"
```

```
"@falanca"
```

```
"c"
```

Karakter dizilerinin ayırt edici özelliği, tırnak işareti taşımalarıdır. Eğer yukarıdaki örnekleri tırnak işaretsiz olarak kullanırsak, artık bu öğeler birer karakter dizisi olmayacaktır... Yukarıdaki örneklerden de gördüğümüz gibi, bir karakter dizisinin ne kadar uzun veya ne kadar kısa olduğunun hiçbir önemi yok. Bir karakter dizisi tek bir karakterden oluşabileceği gibi, birden fazla karakterden de oluşabilir. Bu nedenle "c" de bir karakter dizisidir, "@falanca" da...

Python'da karakter dizilerini ekrana yazdırabilmek için *print* komutundan yararlanabileceğimizi biliyorsunuz. Mesela:

```
>>> print "k"
```

```
k
```

```
>>> print "456"
```

```
456
```

```
>>> print "Ruby"
```

```
Ruby
```

```
>>> print "İstanbul'un 5 günlük hava tahmini"
```

```
İstanbul'un 5 günlük hava tahmini
```

Etkileşimli kabukta çalışırken, istersek yukarıdaki örnekleri *print* komutunu kullanmadan da ekrana yazdırabiliriz:

```
>>> "Python"
```

```
'Python'
```

```
>>> "Perl"
```

```
'Perl'
```

```
>>> "Ruby"
```

```
'Ruby'
```

```
>>> "4243"
```

```
'4243'
```

Ancak bu durum sadece etkileşimli kabuk için geçerlidir. Yani ilerde kodlarımızı bir dosyaya kaydettiğimiz zaman, yukarıdaki kodların çıktı vermediğini, herhangi bir şeyi ekrana yazdırabilmek için mutlaka *print* komutundan yararlanmamız gerektiğini göreceğiz.

Şu örneğe bir bakalım:

```
>>> print "Python programlama dili"
```

Python yukarıdaki gibi bir komutla karşılaştığında iki basamaklı bir işlem gerçekleştirir. Önce

“Python programlama dili” adlı karakter dizisini okur, ardından da *print* komutunun etkisiyle bu karakter dizisini ekrana basıp kullanıcıya gösterir. Yani şöyle bir çıktı verir:

```
Python programlama dili
```

Etkileşimli kabukta bir karakter dizisini *print* komutu olmadan kullandığımızda Python yalnızca birinci basamağı gerçekleştirmiş olur. Böyle bir durumda yaptığı tek şey bu karakter dizisini okumaktır. *print* komutunu kullanarak bu karakter dizisini kullanıcıya göstermediğimiz için ikinci basamak gerçekleşmez. Dolayısıyla elde ettiğimiz çıktı, söz konusu karakter dizisini kullanıcının değil, Python’ın ne şekilde gördüğünü gösterir. Yani Python bize şöyle bir şey söylemiş olur:

*“Sen şimdi etkileşimli kabuğa bir şeyler yazdın. Ben de bunu algıladım. Yazdığın şeyi algıladığımı göstermek için de, çıktıyı sana tek tırnak işaretleri içinde gösterdim. Ancak print gibi bir komut yardımıyla ekrana herhangi bir şey basmamış olman da dikkatimden kaçmadı değil!”*

Basit bir örnek verelim:

```
>>> "dünya"

'd\x3\xbcnya'
```

Gördüğünüz gibi, “dünya” kelimesi içindeki Türkçe karakter (“ü”) çıktıda düzgün görünmüyor. Demek ki Python bizim girdiğimiz “dünya” adlı karakter dizisini “d\x3\xbcnya” şeklinde görüyor... (Buradaki “\x3\xbc” kısmı kullandığınız işletim sistemine göre farklılık gösterebilir.) Biz bu karakter dizisini kullanıcıya gösterebilmek için şöyle yazacağız:

```
>>> print "dünya"

dünya
```

Yazdığımız Python programlarını dosyaya kaydedip çalıştırdığımızda yukarıda anlattıklarımızın ne demek olduğunu çok daha net bir şekilde anlayacağız. O yüzden, eğer yukarıdaki açıklamalar size kafa karıştırıcı geldiyse hiç endişe etmenize gerek yok. Okumaya devam edebilirsiniz.

Karakter dizilerini göstermek için tırnak işaretlerinden yararlanıyoruz demiştik. Python’da karakter dizilerini göstermek için farklı tırnak tiplerini kullanabilirsiniz. Örneğin çift tırnak kullanabilirsiniz:

```
>>> print "Python güçlü bir programlama dilidir."
```

veya tek tırnak:

```
>>> print 'Python güçlü bir programlama dilidir.'
```

Bu tırnak tiplerini önceki örneklerimizde görmüştük. Ama Python bize bunların dışında bir tırnak alternatifi daha sunar. Üç tırnak:

```
>>> print """Python güçlü bir programlama dilidir."""
```

Gördüğünüz gibi, karakter dizileriyle birlikte üç farklı tırnak çeşidi kullanabiliyoruz. Hangi tırnak çeşidini kullandığınızın çok fazla bir önemi yok. Önemli olan, karakter dizisini hangi tırnakla açtıysanız o tırnakla kapatmanızdır.

Bu tırnak tipleri içinde en yaygın kullanılan çift tırnaktır. Tabii siz istediğiniz tırnak biçimini kullanmakta özgürsünüz. Yalnız önceki bölümde, duruma göre kullanılan tırnak tipinin önem kazandığını gösteren şöyle bir örnek verdiğimizizi biliyorsunuz:

```
>>> print "İstanbul'un 5 günlük hava tahmini"
```

Burada "İstanbul'un 5 günlük hava tahmini" bir karakter dizisidir. Bu karakter dizisini göstermek için çift tırnaklardan yararlandık. İstersek elbette bu karakter dizisini tek tırnak ile de gösterebiliriz. Ancak **İstanbul'un** kelimesi içinde geçen kesme işaretinden ötürü, kullanılan tırnakların birbirine karışması gibi bir tehlike söz konusu. Yani, daha önce de gördüğümüz gibi, şöyle bir kullanım Python'ın kafasının karışmasına yol açacaktır:

```
>>> print 'İstanbul'un 5 günlük hava tahmini'
```

Bu kullanımda, Python karakter dizisini kapatan tırnağın hangisi olduğunu anlayamayacak, o yüzden de şöyle bir hata mesajı verecektir:

```
File "<stdin>", line 1
    print 'İstanbul'un 5 günlük hava tahmini'
          ^
SyntaxError: invalid syntax
```

Dikkat ederseniz hata çıktısındaki **İstanbul'un** kelimesinin sonunda minik bir ok işareti var. İşte bu ok, hata üreten sorunlu bölgeyi gösteriyor.

Böyle bir hata mesajı almamak için kaçış dizilerinden yararlanmamız gerektiğini söylemiştik:

```
>>> print 'İstanbul\'un 5 günlük hava tahmini'
```

Biraz sonra kaçış dizilerinden ayrıntılı olarak söz edeceğiz. O yüzden yukarıdaki kullanımın kafanızı karıştırmasına izin vermeyin.

Eğer bunların hiçbirisiyle uğraşmak istemezseniz üç tırnaktan da yararlanabilirsiniz:

```
>>> print """Ahmet'in öğretmeni Cem Bey, "ödevini yapmadan gelme!" dedi."""
```

Gördüğünüz gibi, üç tırnak işareti sayesinde karakter dizisi içinde hem tek tırnak, hem de çift tırnak kullanabildik. Zira yukarıdaki karakter dizisine tek tırnakla başlasak **Ahmet'in** kelimesi içindeki kesme işareti, çift tırnakla başlasak **"ödevini yapmadan gelme!"** ifadesi içindeki çift tırnak işaretleri bize sorun yaratacak... Bu engellere takılmadan, kodumuzu düzgün bir şekilde ekrana yazdırabilmek için üç tırnak işaretlerini kullanmayı tercih ettik.

Üç tırnak işaretinin marifetleri yukarıdakilerle sınırlı değildir. Bu tırnak tipi, kimi durumlarda işimizi bir hayli kolaylaştırır. Mesela kullanıcılarımıza şöyle bir çıktı göstermemiz gerektiğini varsayalım:

```
Günün menüsü
-----
Makarna .... 6 TL
Çorba..... 3 TL
Tatlı..... 2 TL
```

Böyle bir çıktı vermenin en kolay yolu üç tırnak işaretlerinden yararlanmaktır:

```
>>> print """
... Günün menüsü
... -----
... Makarna .... 6 TL
... Çorba..... 3 TL
... Tatlı..... 2 TL
... """
```



Burada ilk üç tırnak işaretini koyduktan sonra ENTER tuşuna bastık. Alt satıra geçildiğinde >>> işaretinin ... işaretine döndüğüne dikkat edin. Bu işaret Python'ın bizden komut girmeye devam etmemizi beklediğini gösteriyor. Biz de bu beklentiye uyarak, "Günün mönüsü" satırını yazıyoruz. Tekrar ENTER tuşuna bastığımızda yine ... işaretini görüyoruz ve hemen "-----" satırını yazıyoruz. Yazacağımız kodlar bitene kadar böyle devam ediyoruz. İşimiz bittiğinde de kapanış tırnaklarını koyup ENTER tuşuna basarak çıktıyı alıyoruz:

```
Günün mönüsü
-----
Makarna .... 6 TL
Çorba..... 3 TL
Tatlı..... 2 TL
```

Normalde, tek tırnak veya çift tırnak ile başladığımız bir karakter dizisinde kapanış tırnağını koymadan ENTER tuşuna basarsak hata alırız:

```
>>> print "Günün Mönüsü
File "<stdin>", line 1
      print "Günün Mönüsü
      ^
SyntaxError: EOL while scanning string literal
```

Burada "Günün Mönüsü" karakter dizisine çift tırnakla başladık, ama tırnağı kapatmadan ENTER tuşuna bastığımız için Python bize bir hata mesajı gösterdi. Aynı şey tek tırnak için de geçerlidir. Ama yukarıda da gördüğünüz gibi üç tırnak farklıdır:

```
>>> print """Günün Mönüsü
...

```

Üç tırnak işareti, karakter dizileri içinde geçen farklı tipte tırnakları atlatmamızı sağlamanın yanı sıra, bize karakter dizilerimizi birden fazla satıra bölme imkanı da tanıyor. Aynı şeyi tek veya çift tırnakla yapamıyoruz. İlerde bu özellikten sık sık faydalandığımızı göreceksiniz.

Karakter dizilerine ilişkin önemli bir konu da bu karakter dizilerinin birbirleriyle nasıl birleştirileceğidir. Mesela bir kimsenin adı ile soyadını birleştirmek istersek ne yapmalıyız?

```
>>> print "Fırat" + "Özgül"

FıratÖzgül
```

Gördüğünüz gibi, "+" işaretini kullanarak iki karakter dizisini birleştirebiliyoruz. Ancak burada bir sorun var. Çıktıda isim ve soyisim birbirine bitişik olarak görünüyor. Bu problemi çözmek için şöyle bir şey yapabilirsiniz:

```
>>> print "Fırat" + " " + "Özgül"

Fırat Özgül
```

Burada "Fırat" ile "Özgül" karakter dizileri arasına boş bir karakter dizisi yerleştirdik. Böylece çıktımız tam istediğimiz gibi olmuş oldu.

Python karakter dizilerini birleştirebilmemiz için bize başka imkanlar da tanır. Mesela şu örneğe bir bakın:

```
>>> print "İstanbul" "Çeliktepe"

İstanbulÇeliktepe
```

Gördüğünüz gibi, iki karakter dizisini, araya herhangi bir işaret koymadan da birleştirebiliyoruz. Ancak burada da biraz öncekine benzer bir sorunla karşı karşıyayız. Karakter dizilerimiz yine birbirine bitişik görünüyor. Biraz önceki yöntemi kullanarak bu karakter dizilerini ayırmamız mümkün:

```
>>> print "İstanbul " "Çeliktepe"
```

Hatta yukarıdaki gibi bir çıktı alabilmek için şöyle bir şeyler de yapabilirsiniz:

```
>>> print "İstanbul " + "Çeliktepe"
```

veya:

```
>>> print "İstanbul " "Çeliktepe"
```

Burada ayrı bir boş karakter dizisi eklemek yerine, "İstanbul" adlı karakter dizisinin kapanış tırnağını koymadan önce bir boşluk bırakıyoruz...

Elbette Python'da çareler tükenmez. Dilerseniz yukarıdaki bütün her şeyi unutup şöyle bir şey yazmayı da tercih edebilirsiniz:

```
>>> print "Fırat", "Özgül"
```

```
Fırat Özgül
```

Burada da karakter dizilerini birbirlerinden virgülle ayırmayı tercih ettik. Bu son kullanım sıklıkla karşımıza çıkacak. O yüzden bu kullanımı aklımızda tutmamız gerekiyor.

Karakter dizilerinden ve bunların inceliklerinden ileride de söz etmeye devam edeceğiz. Ama isterseniz bu konuya şimdilik bir ara verelim ve Python'da farklı bir konuya giriş yapalım.

## 2.3 Python'da Sayılar ve Matematik İşlemleri

Şimdiye kadar Python'da bazı şeyler öğrendik. Ancak öğrendiklerimiz henüz dört dörtlük bir program yazmaya yetmiyor. Ama bu durum, Python'ı şimdilik basit bir hesap makinesi niyetine kullanabilmemize engel değil.

Örneğin:

```
>>> 2 + 5
```

```
7
```

... veya:

```
>>> 5 - 2
```

```
3
```

... ya da:

```
>>> 2 * 5
```

```
10
```

... hatta:

```
>>> 6 / 2
```

```
3
```

İsterseniz bunların başına *print* komutu ekleyerek de kullanabilirsiniz bu kodları. Bir örnek verelim:

```
>>> print 234 + 546
```

```
780
```

Aritmetik işlemler yapabilmek için kullandığımız işaretlerin size mantıklı gelmiş olduğunu zannediyorum. Toplama işlemi için “+”, çıkarma işlemi için “-”, çarpma işlemi için “\*”, bölme işlemi için ise “/” işaretlerini kullanmak gayet makul duruyor, değil mi?

Gördüğünüz gibi sayıları yazarken tırnak işaretlerini kullanmıyoruz. Eğer tırnak işareti kullanırsak Python yazdıklarımızı sayı olarak değil karakter dizisi olarak algılayacaktır. Bu durumu birkaç örnekle görelim:

```
>>> print 25 + 50
```

```
75
```

Bu komut, 25 ve 50’yi toplayıp sonucu çıktı olarak verir. Şimdi aşağıdaki örneğe bakalım:

```
>>> print "25 + 50"
```

```
25 + 50
```

Bu komut 25 ile 50’yi toplamak yerine, ekrana 25 + 50 şeklinde bir çıktı verecektir. Peki, şöyle bir komut verirsek ne olur?

```
>>> print "25" + "50"
```

Böyle bir komutla karşılaşan Python derhal “25” ve “50” karakter dizilerini (bu sayılar tırnak içinde olduğu için Python bunları sayı olarak algılamaz) yan yana getirip birleştirecektir. Yani şöyle bir şey yapacaktır:

```
>>> print "25" + "50"
```

```
2550
```

Uzun lafın kısıası, “25” ifadesi ile “Çeliktepe” ifadesi arasında Python açısından hiç bir fark yoktur. Bunların ikisi de karakter dizisi sınıfına girer. Ancak tırnak işareti olmayan 25 ile “Çeliktepe” ifadeleri Python dilinde ayrı anlamlar taşır. Çünkü bunlardan biri sayı öteki ise karakter dizisidir (*string*).

Şimdi aritmetik işlemlere geri dönelim. Öncelikle şu komutun çıktısını inceleyelim:

```
>>> print 5 / 2
```

```
2
```

Ama biz biliyoruz ki 5’i 2’ye bölerseniz 2 değil 2,5 çıkar. Aynı komutu bir de şöyle deneyelim:

```
>>> print 5.0 / 2
```

```
2.5
```

Gördüğünüz gibi bölme işlemini oluşturan bileşenlerden birinin yanına ".0" koyulursa sorun çözülüyor. Böylelikle Python bizim sonucu kayan noktalı sayı olarak görmek istediğimizi anlıyor. Bu ".0" ifadesini istediğimiz sayının önüne koyabiliriz. Birkaç örnek görelim:

```
>>> print 5 / 2.0
2.5
>>> print 5.0 / 2.0
2.5
```

Python'da aritmetik işlemler yapılırken alıştığımız matematik kuralları geçerlidir. Yani mesela aynı anda bölme çıkarma, toplama, çarpma işlemleri yapılacaksa işlem öncelik sırası, önce bölme ve çarpma sonra toplama ve çıkarma şeklinde olacaktır. Örneğin:

```
>>> print 2 + 6 / 3 * 5 - 4
```

işleminin sonucu 8 olacaktır. Tabii biz istersek parantezler yardımıyla Python'ın kendiliğinden kullandığı öncelik sırasını değiştirebiliriz. Bu arada yapacağımız aritmetik işlemlerde sayıları kayan noktalı sayı cinsinden yazmamız işlem sonucunun kesinliği açısından büyük önem taşır. Eğer her defasında ".0" koymaktan sıkılıyorsanız, şu komutla Python'a, "*Bana her zaman kesin sonuçlar göster,*" mesajı gönderebilirsiniz:

```
>>> from __future__ import division
```

---

**Not:** Burada "\_\_" işaretini klavyedeki alt çizgi tuşuna iki kez art arda basarak yapabilirsiniz.

---

Artık bir sonraki Python oturumuna kadar bütün işlemlerinizin sonucu kayan noktalı sayı olarak gösterilecektir. Bu arada dilerseniz konuyla ilgili bazı ufak tefek teknik bilgiler vereyim. Böylece karşılaştığımız örnekleri daha kolay anlayabiliriz. Mesela yukarıda sayılardan bahsettik. Python'da sayılar çeşit çeşittir:

**Tamsayılar (integers)** Tamsayılar, herhangi bir ondalık kısım barındırmayan sayılardır. Mesela 3, 5, 6, 100, 1450, -56, -3 vb...

**Kayan Noktalı Sayılar (floating point numbers)** Ondalık bir kısım da barındıran sayılardır. Mesela 3.4, 5.5, 6.4, vb... Bu arada ondalık kısmı virgülle değil noktayla gösterdiğimizize dikkat edin.

**Uzun Sayılar (long integers)** -2.147.483.648 ile 2.147.483.647 arasında kalan sayılardır.

**Karmaşık Sayılar (complex numbers)** Karmaşık sayılar bir gerçel, bir de sanal kısımdan oluşan sayılardır. Mesela 34.3j, 2j+4, vb...

Bu sayılar içinde en sık karşılaştığımız tamsayılar ve kayan noktalı sayılardır. Eğer özel olarak matematikle ilgili işler yapmıyorsanız uzun sayılar ve karmaşık sayılar pek karşınıza çıkmaz.

Hatırlarsanız, karakter dizilerinden bahsederken, *print*'li ve *print*'siz kullanımdan söz etmiştik. Orada sözünü ettiğimiz durum sayılar için de geçerlidir. Mesela şu örneğe bakalım:

```
>>> 12 - 3.3
8.6999999999999993
```

Herhalde beklediğiniz çıktı bu değildi. Bu çıktı, 12 - 3.3 işlemini Python'ın nasıl gördüğünü gösteriyor. Eğer bu işlemi insanların anlayabileceği bir biçime getirmek istersek şöyle yazmamız gerekir:

```
>>> print 12 - 3.3
```

```
8.7
```

İlerde yeri geldiğinde sayılardan daha ayrıntılı bir şekilde bahsedeceğiz. O yüzden sayılar konusunu şimdi bir kenara bırakıp çok önemli başka bir konuya değinelim.

## 2.4 Değişkenler

Kabaca, bir veriyi kendi içinde depolayan birimlere değişken adı veriyorlar. Ama şu anda aslında bizi değişkenin ne olduğundan ziyade neye yaradığı ilgilendiriyor. O yüzden hemen bir örnekle durumu açıklamaya çalışalım. Mesela;

```
>>> n = 5
```

ifadesinde n bir değişkendir. Bu n değişkeni 5 verisini sonradan tekrar kullanılmak üzere depolar. Python komut satırında n = 5 şeklinde değişkeni tanımladıktan sonra print n komutunu verirse ekrana yazdırılacak veri 5 olacaktır. Yani:

```
>>> n = 5  
>>> print n
```

```
5
```

Bu n değişkenini alıp bununla aritmetik işlemler de yapabiliriz:

```
>>> n * 2  
  
10  
  
>>> n / 2.0  
  
2.5
```

Hatta bu n değişkeni, içinde bir aritmetik işlem de barındırabilir:

```
>>> n = 34 * 45  
>>> print n  
  
1530
```

Şu örneklerle bir göz atalım:

```
>>> a = 5  
>>> b = 3  
>>> print a * b  
  
15  
  
>>> print "a ile b'yi çarparsak", a * b, "elde ederiz"  
  
a ile b'yi çarparsak 15 elde ederiz
```

Burada değişkenleri karakter dizileri arasına nasıl yerleştirdiğimize, virgülleri nerede kullandığımıza dikkat edin.

Aynı değişkenlerle yaptığımız şu örneğe bakalım bir de:

```
>>> print a, "sayısı", b, "sayısından büyüktür"
```

Gördüğünüz gibi, aslında burada yaptığımız şey, karakter dizilerini birbiriyle birleştirmekten ibarettir. Hatırlarsanız bir önceki bölümde iki karakter dizisini şu şekilde birleştirebileceğimizi öğrenmiştik:

```
>>> print "Fırat", "Özgül"
```

İşte yukarıda yaptığımız şeyin de bundan hiçbir farkı yoktur. Dikkatlice bakın:

```
>>> print a, "sayısı", b, "sayısından büyüktür"
```

Yukarıdaki kod şununla eşdeğerdir:

```
>>> print 5, "sayısı", 3, "sayısından büyüktür"
```

Biz burada doğrudan sayıları yazmak yerine, bu sayıları tutan değişkenleri yazdık...

Değişkenleri kullanmanın başka bir yolu da özel işaretler yardımıyla bunları karakter dizileri içine gömmektir. Şu örneğe bir bakalım:

```
>>> print "%s ile %s çarpılırsa %s elde edilir" %(3, 5, 3*5)
```

Bu da oldukça kullanışlı bir tekniktir. Burada, parantez içinde göstereceğimiz her bir öge için karakter dizisi içine "%s" işaretini ekliyoruz. Karakter dizisini yazdıktan sonra da "%" işaretinin ardından parantez içinde bu değişkenleri teker teker tanımlıyoruz. Buna göre birinci değişkenimiz 3, ikincisi 5, üçüncüsü ise bunların çarpımı...

Bir de şu örneği inceleyelim:

```
>>> print "%s ve %s iyi bir ikilidir." %("Python", "Django")
```

Görüleceği gibi, bu kez değişkenlerimiz tamsayı yerine karakter dizisi olduğu için parantez içinde değişkenleri belirtirken tırnak işaretlerini kullanmayı unutmuyoruz.

## 2.5 Kaçış Dizileri

Önceki bölümlerde "kaçış dizisi" diye bir şeyden söz ettik. Hatta bununla ilgili bir örnek de yaptık. İsterseniz o örneği hatırlayalım:

```
>>> print 'İstanbul\'un 5 günlük hava tahmini'
```

İşte bu karakter dizisi içinde kullandığımız "\"" işareti bir kaçış dizisidir. Bu kaçış dizisinin bu radaki görevi, **İstanbul'un** kelimesi içinde geçen kesme işaretinin, karakter dizisini açan ve kapatan tırnak işaretleriyle karışmasını önlemek. Python bir karakter dizisiyle karşılaştığında, bu karakter dizisini soldan sağa doğru okur. Mesela yukarıdaki örnekte önce tek tırnak işaretini görüyor ve bir karakter dizisi tanımlayacağımızı anlıyor. Ardından karakter dizisi içindeki kesme işaretine rastlıyor ve bu işaretin karakter dizisinin bitişini gösteren kesme işareti olduğunu, yani karakter dizisinin sona erdiğini zannediyor. Ancak okumaya devam ettiğinde, karakter dizisine birleşik bir halde "un 5 günlük hava tahmini" diye bir şeyle karşılaşılıyor ve işte bu noktada kafası allak bullak oluyor. Bizim **İstanbul'un** kelimesindeki kesme işaretinin sol tarafına yerleştirdiğimiz "\"" kaçış dizisi Python'a, bu kesme işaretini görmezden gelme emri veriyor. Böylece karakter dizimiz hatasız bir şekilde ekrana basılıyor...

Dilerseniz bununla ilgili birkaç örnek daha yapalım:

```
>>> print "1 numaralı oyuncudan gelen cevap: "Sola dön!"
```

Bu karakter dizisi içinde “Sola dön!” ifadesi var. Biz yukarıdaki karakter dizisini tanımlarken çift tırnaktan yararlandığımız için, karakter dizisi içindeki bu “Sola dön!” ifadesi çakışmaya sebep oluyor. İstersek yukarıdaki karakter dizisini şu şekilde yazabiliriz:

```
>>> print '1 numaralı oyuncudan gelen cevap: "Sola dön!"'
```

Bu komut sorunsuz bir şekilde yazdırılır. Ama eğer karakter dizisini çift tırnakla tanımlamanız gerekirse, kaçış dizilerinden yararlanabilirsiniz:

```
>>> print "1 numaralı oyuncudan gelen cevap: \"Sola dön!\""
```

Burada iki adet kaçış dizisi kullandığımıza dikkat edin. Bunun nedeni, karakter dizisi içinde başlangıç ve bitiş tırnaklarıyla karışabilecek iki adet tırnak işareti olmasıdır... Ayrıca kaçış dizisini, sorunlu tırnağın soluna koyduğumuza da dikkat edin.

Hatırlarsanız kaçış dizilerinden ilk bahsettiğimiz yerde şöyle bir örnek de vermiştik:

```
>>> print "Python'da önemli komutlardan biri de "print" komutudur."
```

Bu komutun hata vereceğini biliyoruz. Çünkü karakter dizisini çift tırnak içinde tanımladık. Dolayısıyla karakter dizisi içinde geçen çift tırnaklı “print” ifadesi çakışmaya yol açtı. Bu sorunu aşmak için karakter dizisini şöyle yazıyoruz:

```
>>> print "Python'da önemli komutlardan biri de \"print\" komutudur."
```

Burada kaçış dizilerini çift tırnaklı “print” kelimesine nasıl uyguladığımıza dikkat edin. Her bir çift tırnağın soluna bir adet “\” işareti yerleştirdik. Böylece hatadan kıvrak hareketlerle kaçmış olduk...

“\” işareti yalnızca tırnak işaretleriyle birlikte kullanılmaz. Bu işaret başka karakterlerle birlikte de kullanılarak farklı amaçlara hizmet edebilir.

Şu örneğe bir bakın:

```
>>> print "birinci satır\nikinci satır"
```

Burada “\” işaretini “n” harfiyle birlikte kullandık. “\” işareti “n” harfiyle birlikte kullanıldığında özel bir anlam ifade eder. Yukarıdaki kod şöyle bir çıktı verir:

```
birinci satır
ikinci satır
```

Gördüğünüz gibi, karakter dizisini iki satıra böldük. Bunu yapmamızı sağlayan şey “\n” adlı kaçış dizisidir. Bu kaçış dizisi yeni bir satıra geçmemizi sağlar. Daha belirgin bir örnek verelim:

```
>>> print "Yolun sonu\n"
```

Bu komut, “Yolun sonu” karakter dizisini çıktı olarak verdikten sonra bir satır boşluk bırakacaktır.

Hatırlarsanız önceki bölümde, *print* komutunu kullanmadan herhangi bir karakter dizisi yazdığımızda elde ettiğimiz çıktının Python’ın bakış açısını yansıttığını söylemiştik. Bu durumu çok daha net anlayabilmemizi sağlayacak bir örnek verelim:

```
>>> """
... Günün mönüsü
... -----
```

```
... Makarna..... 6 TL
... Çorba..... 3 TL
... Tatlı..... 2 TL
... ""
```

Bu örneği hatırlıyorsunuz. Daha önce aynı örneği *print* komutu ile birlikte kullanmıştık.

Bu kodlar şöyle bir çıktı verir:

```
'\nGünün mönüsü\n-----\nMakarna..... 6 TL\nÇorba..... 3 TL\nTatlı..... 2 TL\n'
```

Gördüğünüz gibi, çıktıda “\n” adlı kaçış dizileri de görünüyor. Demek ki Python üç tırnak yardımıyla girdiğimiz kodları böyle görüyormuş. Bu çıktıdan şunu da anlıyoruz: Demek ki biz üç tırnak yerine tek tırnak kullanmak istesek aynı çıktıyı elde etmek için şöyle gibi bir komut vermemiz gerekecek:

```
>>> print '\nGünün mönüsü\n-----\nMakarna..... 6 TL\nÇorba..... \n... 3 TL\nTatlı..... 2 TL\n'
```

```
Günün mönüsü
-----
Makarna..... 6 TL
Çorba..... 3 TL
Tatlı..... 2 TL
```

Buradan, üç tırnak işaretlerinin kimi durumlarda hayatımızı ne kadar da kolaylaştırabileceğini anlayabilirsiniz...

Bu arada, yukarıdaki komutlarda bir şey dikkatinizi çekmiş olmalı. `>>> print '\nGünün mönüsü\n--` ile başlayan komutun ilk satırının sonunda, ENTER tuşuna basıp yeni satıra geçmeden önce “\n” adlı kaçış dizisini kullandık. Elbette istersek bu iki satırı tek bir satırda da yazabiliriz. Ancak görüntü açısından kodların bu şekilde sağa doğru uzaması hoş değil. Kötü bir kod görünümü ortaya çıkmaması için, bu kaçış dizisini kullanarak kodlarımızı iki satıra bölmeyi tercih ettik.

---

**Not:** Python’da doğru kod yazmak kadar, yazdığınız kodların düzgün görünmesi ve okunaklı olması da önemlidir. Bu yüzden yazdığımız programlarda satır uzunluğunun mümkün olduğunca 79 karakterden fazla olmamasına dikkat ediyoruz. Eğer yazdığımız bir satır 79 karakterden uzun olarsa satırı uygun bir yerinden bölüp yeni bir satıra geçmemiz gerekir. Bu durum teknik bir zorunluluk değildir, ancak okunaklılığı artırdığı için uyulması tavsiye edilir. Örneğin biz de yukarıdaki örnekte kodlarımızın sağa doğru çirkin bir şekilde uzamaması için satırı uygun bir yerinden bölüp alt satıra geçtik.

---

Bu kaçış dizisini kullanmamızın amacı, Python’ın herhangi bir hata vermeden alt satıra geçmesini sağlamak. Eğer orada o kaçış dizisini kullanmazsak, ENTER tuşuna bastığımız anda Python hata verecektir. “\n” işareti, alt satıra geçilmiş de olsa kodların ilk satırdan devam etmesi gerektiğini gösteriyor. Bu durumu belirgin bir şekilde gösteren bir örnek verelim:

```
>>> print 'mer\
... haba'

merhaba
```

Gördüğünüz gibi, “merhaba” kelimesini “mer” hecesinden bölüp “\n” işareti ile alt satıra geçtiğimiz halde, çıktımız “merhaba” oldu. Eğer amacınız “merhaba” kelimesini “mer” ve “haba” olarak bölmekse şöyle bir şey yapabilirsiniz:



```
>>> print """mer
...   haba"""
```

```
mer
haba
```

Eğer üç tırnak kullandığınız halde, “merhaba” kelimesinin bölünmesini istemiyorsanız yine “\” kaçış dizisinden yararlanabilirsiniz:

```
>>> print """mer\
...   haba"""
```

```
merhaba
```

Eğer üç tırnak yerine çift veya tek tırnak kullanarak “merhaba” kelimesini ortadan bölmek isterseniz elbette “\n” adlı kaçış dizisinden yararlanabilirsiniz:

```
>>> print "mer\nhaba"
```

```
mer
haba
```

```
>>> print 'mer\nhaba'
```

```
mer
haba
```

---

**Not:** Çok uzun bir karakter dizisini iki satıra bölmek için “\” işaretini kullanırken, bu işaretin satırdaki son karakter olmasına özen gösterin. Yani “\” işaretini koyar koymaz ENTER tuşuna basmalısınız. “\” işaretinden sonra boşluk bile olmamalı.

---

Bütün bu örneklerden şunu anlıyoruz: Python’da temel olarak tek bir kaçış dizisi var. O da “\” işareti... Bu işaret farklı karakterlerle bir araya gelerek, değişik anlamlar ortaya çıkarabiliyor. Mesela bu karakter dizisi tırnak işaretiyle birlikte kullanıldığında, karakter dizisini başlatıp bitiren tırnaklarla çakışılmasını önüyor. Eğer bu kaçış dizisini “n” harfiyle birlikte kullanırsak, karakter dizilerini bölüp alt satıra geçebiliyoruz. Eğer bu kaçış dizisini tek başına kullanırsak, bir karakter dizisini herhangi bir yerinden bölüp alt satıra geçtiğimiz halde, o karakter dizisi çıktıda tek satır olarak görünüyor. Bütün bunların dışında bu kaçış dizisi “t” harfiyle de yan yana gelebilir. O zaman da şöyle bir anlam kazanır:

```
>>> print "Uzak... \t Çok uzak"
```

```
Uzak...      Çok uzak
```

Bu kaçış dizisinin görevi karakter dizisi içindeki karakterler arasına mesafe koymak. Bu kaçış dizisi de kimi yerlerde hayatınızı kolaylaştırabilir. “\” kaçış dizisi başka karakterlerle de bir araya gelebilir. Ancak en önemlilerini zaten yukarıda verdik. Programlama maceramızda en çok işimize yarayacak olanlar bunlardır.

Yukarıdakilerin dışında, Python’da şablona uymayan önemli bir kaçış dizisi daha bulunur. Bu kaçış dizisinin adı “r”’dir. Bu kaçış dizisinin hem görünüşü hem de kullanımı yukarıdaki kaçış dizilerinden biraz farklıdır. Dilerseniz bu kaçış dizisini anlatmadan önce size şöyle bir soru sorayım:

Acaba aşağıdaki karakter dizisini nasıl yazdırırız?

“C:\Documents and Settings\nergis\test”

İsterseniz bu karakter dizisini önce şöyle yazdırmayı deneyelim:

```
>>> print "C:\Documents and Settings\nergis\test"
```

```
C:\Documents and Settings  
nergis  est
```

Gördüğünüz gibi Python bize tuhaf bir çıktı verdi. Sizce neden böyle oldu?

Cevap aslında çok basit. Karakter dizisinde kalın harflerle gösterdiğimiz kısımlara dikkat edin: C:\Documents and Settings\nergis\test

Burada “nergis” ve “test” kelimelerinin başındaki harfler, dizin ayracı olan “\” işareti ile yan yana geldiği için Python bunları birer kaçış dizisi olarak algılıyor. Bu durumdan kurtulmak için birkaç seçeneğimiz var. İstersek yukarıdaki karakter dizisini şöyle yazabiliriz:

```
>>> print "C:\\Documents and Settings\\nergis\\test"
```

Burada “\” işaretlerini çiftleyerek sorunun üstesinden gelebiliyoruz. Ama eğer istersek yukarıdaki soruna çok daha temiz bir çözüm de bulabiliriz. İşte böyle bir duruma karşı Python’da “r” adlı bir kaçış dizisi daha bulunur. Bunu şöyle kullanıyoruz:

```
>>> print r"C:\Documents and Settings\nergis\test"
```

Gördüğünüz gibi, “r” adlı kaçış dizisi, karakter dizisi içinde geçen bütün kaçış dizilerini etkisiz hale getiriyor. Bu kaçış dizisi özellikle dizin adlarını yazdırırken çok işinize yarayacaktır. İsterseniz bununla ilgili şöyle bir örnek daha verelim. Diyelim ki şu cümleyi ekrana yazdırmak istiyoruz:

```
Python’daki kaçış dizileri şunlardır: \, \n, \t
```

Bu cümleyi şöyle yazdıramayız:

```
>>> print "Python’daki kaçış dizileri şunlardır: \, \n, \t"
```

```
Python’daki kaçış dizileri şunlardır: \,  
,
```

Gördüğünüz gibi çıktı hiç de beklediğimiz gibi değil. Çünkü Python karakter dizisi içinde geçen kaçış dizilerini işleme sokuyor ve buradaki her bir kaçış dizisinin görevini yerine getirmesine izin veriyor. Bu kaçış dizilerinin etkisizleştirilebilmesi için şu kodu yazabilirsiniz:

```
print "Python’daki kaçış dizileri şunlardır: \\, \\n, \\t"
```

Ama burada “r” kaçış dizisini kullanırsak çok daha temiz bir kod görünümü elde etmiş oluruz:

```
print r"Python’daki kaçış dizileri şunlardır: \, \n, \t"
```

“r” adlı kaçış dizisi, karakter dizisi içindeki kaçış dizilerinin görevlerini yerine getirmesine engel olacaktır.

Dilerseniz şimdiye kadar gördüğümüz kaçış dizilerini şöyle bir özetleyelim:

\ ENTER tuşuna basılarak alt satıra kaydırılan karakter dizilerinin çıktıda tek satır olarak görünmesini sağlar.

\' Karakter dizilerini başlatıp bitiren tek tırnaklarla, karakter dizisi içinde geçen tek tırnakların karışmasını engeller.

\" Karakter dizilerini başlatıp bitiren çift tırnaklarla, karakter dizisi içinde geçen çift tırnakların karışmasını engeller.

**\n** Yeni bir satıra geçilmesini sağlar.

**\t** Karakter dizileri içinde sekme (TAB) tuşuna basılmış gibi boşluklar oluşturulmasını sağlar.

**r** Karakter dizileri içinde geçen kaçış dizilerini etkisiz hale getirir.

Şimdi dilerseniz bölüm sorularına geçelim ve bu konuyu kapatalım.

## 2.6 Bölüm Soruları

1. Python'ın 2.x sürümünü mü yoksa 3.x sürümünü mü kullandığınızı tespit edin. Eğer hangi sürümü kullanmanız gerektiği konusunda tereddütleriniz varsa <http://www.istihza.com/blog/hangisinden-baslamali-python-2x-mi-yoksa-3x-mi.html/> adresindeki makalemizi inceleyin.

2. Aşağıdakilerden hangisi bir karakter dizisi değildir?

```
"python"
www.google.com
'23'
"""@%&"""
'''istihza'''
```

3. Aşağıdaki tabloyu ekrana çıktı olarak verin:

```
Yıllara Göre Asgari Ücret
-----
2005..... 488,70 YTL
2006..... 531,00 YTL
2007..... 585,00 YTL
2008..... 638,70 YTL
2009..... 666,00 TL
2010..... 760,50 TL
```

4. Aşağıdaki karakter dizilerini farklı yöntemler kullanarak birleştirin ve tek bir cümle elde edin:

```
"Ahmet"
"yarın Ankara'ya"
"gidecek"
```

Sizce en kolay ve kullanışlı yöntem hangisi?

5. Aşağıdaki işlemin sonucunu önce kafanızdan sonra da Python'la hesaplayın. Eğer Python'dan aldığınız sonuç beklediğiniz gibi değilse nedenini bulmaya çalışın.

```
2 + 3 x 7
```

6. Aşağıdaki ifadeyi **üç tırnak işareti kullanmadan** ekrana çıktı olarak verin:

```
Bilmezler yalnız yaşamayanlar,
Nasıl korku verir sessizlik insana.
                                0.Veli
```

7. Aşağıdaki değişkenleri kullanarak “Depoda 10 kilo elma, 20 kilo da armut kaldı” cümlesini elde edin:

```
elma = 10  
armut = 20
```

8. Aşağıdaki cümleleri, farklı tırnak tiplerini kullanarak yazdırın:

```
Doktor Kemal'in yaptığı açıklamaya göre, "Endişe edecek hiçbir şey yok!"
```

```
Python'daki en kullanışlı tırnak tiplerinden biri de "üç tırnak" (""") işaretidir.
```

---

# Python Programlarını Kaydetmek

---

Özellikle küçük kod parçaları yazıp bunları denemek için Python komut satırı mükemmel bir ortamdır. Ancak kodlar çoğalıp büyümeye başlayınca komut satırı yetersiz gelmeye başlayacaktır. Üstelik tabii ki yazdığınız kodları bir yere kaydedip saklamak isteyeceksiniz. İşte burada metin düzenleyiciler devreye girecek.

Python kodlarını yazmak için istediğiniz herhangi bir metin düzenleyiciyi kullanabilirsiniz. Hatta Notepad bile olur. Ancak Python kodlarını ayırt edip renklendirebilen bir metin düzenleyici ile yola çıkmak her bakımdan hayatınızı kolaylaştıracaktır.

Biz bu bölümde farklı işletim sistemlerinde, metin düzenleyici kullanılarak Python programlarının nasıl yazılacağını tek tek inceleyeceğiz. Daha önce de söylediğimiz gibi, hangi işletim sistemini kullanıyor olursanız olun, hem Windows hem de GNU/Linux başlığı altında yazılanları okumanızı tavsiye ederim.

Dilerseniz önce GNU/Linux ile başlayalım:

## 3.1 GNU/Linux Sistemi

Eğer kullandığınız sistem GNU/Linux'ta KDE masaüstü ortamı ise başlangıç düzeyi için Kwrite veya Kate adlı metin düzenleyicilerden herhangi biri yeterli olacaktır. Şu aşamada kullanım kolaylığı ve sadeliği nedeniyle Kwrite önerilebilir.

Eğer kullandığınız sistem GNU/Linux'ta GNOME masaüstü ortamı ise Gedit'i kullanabilirsiniz.

İşe yeni bir Kwrite belgesi açarak başlayalım. Yeni bir Kwrite belgesi açmanın en kolay yolu ALT+F2 tuşlarına basıp, çıkan ekranda:

```
kwrite
```

yazmaktır.

Yeni bir Gedit belgesi oluşturmak için ise ALT+F2 tuşlarına basıp:

```
gedit
```

komutunu veriyoruz.

Boş Kwrite veya Gedit belgesi karşımıza geldikten sonra ilk yapmamız gereken, ilk satıra:

```
#!/usr/bin/env python
```

yazmak olacaktır. Peki bu komut ne işe yarıyor?

Bu komut, biraz sonra yazacağımız kodların birer Python kodu olduğunu ve Python'ın da sistemde hangi konumda yer aldığını gösteriyor. Böylece sistemimiz Python'ı nerede bulması gerektiğini anlıyor.

GNU/Linux sistemlerinde Python'ın çalıştırılabilir dosyası genellikle */usr/bin* dizini altındadır. Dolayısıyla yukarıdaki satırı aslında şöyle de yazabilirsiniz:

```
#!/usr/bin/python
```

Böylece sisteminiz Python'ı */usr/bin* dizini altında arayacak ve yazdığınız programı Python'la çalıştırması gerektiğini anlayacaktır. Ancak bazı GNU/Linux sistemlerinde Python'ın çalıştırılabilir dosyası başka bir dizinin içinde de olabilir (Mesela */usr/local/bin*). Bu durumda, çalıştırılabilir dosya */usr/bin* altında bulunamayacağı için, bazı koşullarda sistem yazdığınız programı çalıştıramayacaktır. Python'ın çalıştırılabilir dosyasının her sistemde aynı dizin altında bulunmama ihtimalinden ötürü yukarıdaki gibi sabit bir dizin adı vermek iyi bir fikir olmayabilir. Bu tür sistem farklılıklarına karşı önlem olarak GNU/Linux sistemlerindeki **env** adlı bir betikten yararlanabiliriz. */usr/bin* dizini altında bulunduğunu varsayabileceğimiz bu **env** betiği Python'ın sistemde hangi dizin içinde bulunduğunu bulmamızı sağlar. Böylece yazdığımız bir programın, Python'ın */usr/bin* dizini haricinde bir konuma kurulduğu sistemlerde çalıştırılması konusunda endişe etmemize gerek kalmaz. Programımızın en başına eklediğimiz *#!/usr/bin/env python* satırı sayesinde Python sistemde nereye kurulmuş olursa olsun kolaylıkla tespit edilebilecektir.

Uzun lafın kısıası, *#!/usr/bin/python* yazdığımızda sisteme şu emri vermiş oluyoruz: "Python'ı */usr/bin* dizini içinde ara!"

*#!/usr/bin/env python* yazdığımızda ise şu emri: "Python'ı nereye saklandıysa bul!"

Neyse... Biz daha fazla teknik ayrıntıya dalmadan yolumuza devam edelim.

Aslında metin içine kod yazmak, Python komut satırına kod yazmaktan çok farklı değildir. Şimdi aşağıda verilen satırları Kwrite veya Gedit belgesinin içine ekleyelim:

```
#!/usr/bin/env python
```

```
a = "elma"  
b = "armut"  
c = "muz"
```

```
print "bir", a, "bir", b, "bir de", c, "almak istiyorum"
```

Bunları yazıp bitirdikten sonra sıra geldi dosyamızı kaydetmeye. Şimdi dosyamızı *deneme.py* adıyla herhangi bir yere kaydediyoruz. Gelin biz masaüstüne kaydedelim dosyamızı. Şu anda masaüstünde *deneme.py* adında bir dosya görüyor olmamız lazım. Şimdi hemen bir konsol ekranı açıyoruz. (Ama Python komut satırını çalıştırmıyoruz.) Şu komutu vererek, masaüstüne, yani dosyayı kaydettiğimiz yere geliyoruz:

```
cd Desktop
```

Tabii burada ben sizin ev dizininde olduğunuzu varsaydım. Komut satırını ilk başlattığınızda içinde bulunduğunuz dizin ev dizininiz olacaktır (*/home/kullanici\_adi*). Dolayısıyla yukarıda gösterdiğimiz *cd Desktop* komutu sizi masaüstünün olduğu dizine götürür. Tabii eğer siz komut satırını farklı bir dizin içinde açmışsanız tek başına yukarıdaki komutu vermeniz sizi masaüstüne götürmez. Öyle bir durumda, şuna benzer bir komut vermeniz gerekir:

```
cd /home/kullanici_adi/Desktop
```

**Not:** Bazı Türkçe GNU/Linux sistemlerinde masaüstünü gösteren dizin “Desktop” yerine “Masaüstü” adını da taşıyabilmektedir. Öyle ise tabii ki vereceğiniz komutta “Desktop” kelimesini “Masaüstü” kelimesiyle değiştirmeniz gerekir.

Eğer başarıyla masaüstüne gelmişseniz, yazdığınız programı çalıştırmak için şu komutu verip ENTER tuşuna basın:

```
python deneme.py
```

Eğer her şey yolunda gitmişse şu çıktıyı almamız lazım:

```
bir elma, bir armut, bir de muz almak istiyorum
```

GNOME kullanıcıları da yukarıda anlatılan işlemleri takip ederek ve Kwrite yerine Gedit adlı metin düzenleyiciyi kullanarak dosyayı kaydedip çalıştırabilir.

Gördüğünüz gibi, `python deneme.py` komutuyla programlarımızı çalıştırabiliyoruz. Normal şartlar altında tercih edeceğimiz program çalıştırma biçimi de bu olacaktır. Peki, ama acaba Python programlarını başa `python` komutu eklemekten çalıştırmanın bir yolu var mı? İşte burada biraz önce bahsettiğimiz `#!/usr/bin/env python` satırının önemi ortaya çıkıyor.

Başta `python` komutu getirmeden programımızı çalıştırabilmek için öncelikle programımıza çalıştırma yetkisi vermemiz gerekiyor. Bunu şu komut yardımıyla yapıyoruz:

Öncelikle:

```
cd Desktop
```

komutuyla dosyayı kaydettiğimiz yer olan masaüstüne geliyoruz. Bunun ardından:

```
chmod a+x deneme.py
```

komutuyla da *deneme.py* adlı dosyaya çalıştırma yetkisi veriyoruz, yani dosyayı çalıştırılabilir (*executable*) bir dosya haline getiriyoruz.

Artık komut satırında şu komutu vererek programımızı çalıştırabiliriz:

```
./deneme.py
```

Peki, tüm bu işlemlerin `#!/usr/bin/env python` satırıyla ne ilgisi var?

Eğer bu satırı metne yerleştirmesek, sistem bu betiği hangi programla çalıştırması gerektiğini anlayamayacağı için `./deneme.py` komutu çalışmayacaktır. `python deneme.py` komutunu verdiğimizde, betiği Python programının çalıştıracağı anlaşılıyor. Ancak baştaki “python” ifadesini kaldırdığımızda, eğer betiğin ilk satırında `#!/usr/bin/env python` da yoksa, sistem bu betikle ne yapması gerektiğine karar veremez.

Eğer bu satırı `#!/usr/bin/python` şeklinde yazarsanız ve eğer programınızın çalıştırıldığı sistemde Python’ın çalıştırılabilir dosyası `/usr/bin/` dizini altında değilse şuna benzer bir hata çıktısı alırsınız:

```
bash: ./deneme.py: /usr/bin/python: bad interpreter: No such file or directory
```

Böyle bir hata almamak için o satırı `#!/usr/bin/env python` şeklinde yazmaya özen gösteriyoruz. Böylece Python nereye kurulmuş olursa olsun sistemimiz Python’ı tespit edebiliyor.

Yukarıdaki işlemlerden sonra bu *deneme.py* dosyasının isminin sonundaki *.py* uzantısını kaldırıp,

```
./deneme
```

komutuyla da programımızı çalıştırabiliriz.

Ya biz programımızı sadece ismini yazarak çalıştırmak istersek ne yapmamız gerekiyor?

Bunu yapabilmek için programımızın *PATH* değişkeni içinde yer alması, yani Türkçe ifade etmek gerekirse, programın YOL üstünde olması gerekir. Peki, bir programın YOL üstünde olması ne demek?

Bilindiği gibi, bir programın veya dosyanın yolu, kabaca o programın veya dosyanın içinde yer aldığı dizindir. Örneğin *fstab* dosyasının yolu */etc/fstab*'dır. Başka bir örnek vermek gerekirse, *xorg.conf* dosyasının yolu */etc/X11/xorg.conf*'tur. Bu "yol" kelimesinin bir de daha özel bir anlamı bulunur. Bilgisayar dilinde, çalıştırılabilir dosyaların (*.bin* ve *.sh* dosyaları çalıştırılabilir dosyalardır) içinde yer aldığı dizinlere de özel olarak YOL adı verilir ve bu anlamda kullanıldığında "yol" kelimesi genellikle büyük harfle yazılır.

İşletim sistemleri, bir programı çağırdığımızda, söz konusu programı çalıştırabilmek için bazı özel dizinlerin içine bakar. Çalıştırılabilir dosyalar eğer bu özel dizinler içinde iseler, işletim sistemi bu dosyaları bulur ve çalıştırılmalarını sağlar. Böylece biz de bu programları doğrudan isimleriyle çağırabiliriz. Şimdi bu konuyu daha iyi anlayabilmek için birkaç deneme yapalım. Hemen bir konsol ekranı açıp şu komutu veriyoruz:

```
echo $PATH
```

Bu komutun çıktısı şuna benzer bir şey olacaktır:

```
/usr/local/sbin:/usr/local/bin:/usr/sbin:  
/usr/bin:/sbin:/bin:/usr/games
```

Bu çıktı bize YOL değişkeni (İngilizce'de *PATH variable*) dediğimiz şeyi gösteriyor. İşletim sistemimiz, bir programı çağırmak istediğimizde öncelikle yukarıda adı verilen dizinlerin içini kontrol edecektir. Eğer çağırdığımız programın çalıştırılabilir dosyası bu dizinlerden herhangi birinin içinde ise o programı ismiyle çağırabiliyoruz.

Örneğin;

```
which gedit
```

komutunun çıktısına bir bakalım:

```
/usr/bin/gedit
```

Gördüğünüz gibi Gedit programının YOL'u */usr/bin/gedit*. Hemen yukarıda `echo $PATH` komutunun çıktısını kontrol ediyoruz ve görüyoruz ki */usr/bin/* dizini YOL değişkenleri arasında var. Dolayısıyla, Gedit programı YOL üstündedir, diyoruz. Zaten Gedit programının YOL üstünde olması sayesinde, konsolda sadece `gedit` komutunu vererek Gedit programını çalıştırabiliyoruz.

Şimdi eğer biz de yazdığımız programı doğrudan ismiyle çağırabilmek istiyorsak programımızı `echo $PATH` çıktısında verilen dizinlerden birinin içine kopyalamamız gerekiyor. Mesela programımızı */usr/bin* içine kopyalayalım. Tabii ki bu dizin içine bir dosya kopyalayabilmek için *root* yetkilerine sahip olmalısınız. Şu komut işinizi görecek:

```
sudo cp Desktop/deneme /usr/bin
```

Şimdi konsol ekranında:



```
deneme
```

yazınca programımızın çalıştığını görmemiz lazım.

Program dosyamızı YOL üstünde bulunan dizinlerden birine eklemek yerine, dosyamızın içinde bulunduğu dizini YOL'a eklemek de mümkün. Şöyle ki:

```
PATH=$PATH:$HOME/Desktop
```

Bu şekilde masaüstü dizinini YOL'a eklemiş olduk. İsterseniz;

```
echo $PATH
```

komutuyla masaüstünüzün YOL üstünde görünüp görünmediğini kontrol edebilirsiniz. Bu sayede artık masaüstünde bulunan çalıştırılabilir dosyalar da kendi adlarıyla çağrılabilirler. Ancak masaüstünü YOL'a eklediğinizde, masaüstünüz hep YOL üstünde kalmayacak, mevcut konsol oturumu kapatılınca her şey yine eski haline dönecektir. Eğer herhangi bir dizini kalıcı olarak YOL'a eklemek isterseniz, `.profile` dosyanızda değişiklik yapmanız gerekir. Mesela masaüstünü YOL'a eklemek için `/home/kullanici_adi/.profile` dosyanızın en sonuna şu satırı eklemelisiniz:

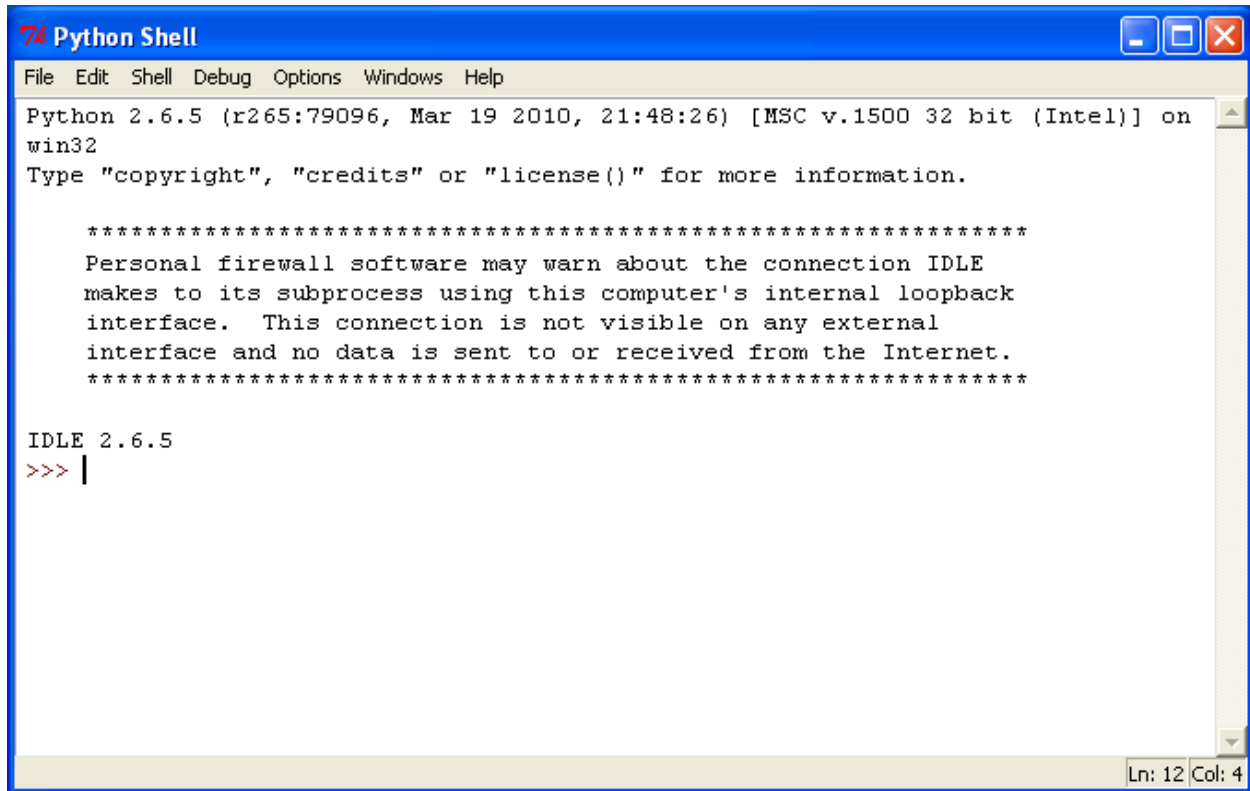
```
export PATH=$PATH:$HOME/Desktop
```

Yoruldunuz mu? Yoksa sıkıldınız mı? Bence ne yorulun, ne de sıkılın... Çünkü bu bölümde verdiğimiz bilgiler, Python'la bir program yazıp bu programı, kullandığınız dağıtımın KDE veya GNOME menüsüne eklemek istediğinizde işinize çok yarayacaktır. Eğer GNU/Linux altında herhangi bir programlama faaliyetinde bulunmak istiyorsanız yukarıda verdiğimiz bilgileri özümlemeniz gerekir.

Gelelim Windows kullanıcılarına...

## 3.2 Windows Sistemi

Windows kullanıcıları IDLE adlı metin düzenleyici ile çalışabilirler. IDLE'a ulaşmak için *Başlat/Programlar/Python/IDLE (Python GUI)* yolunu takip ediyoruz. IDLE ilk başlatıldığında şöyle bir görünüme sahip olacaktır:



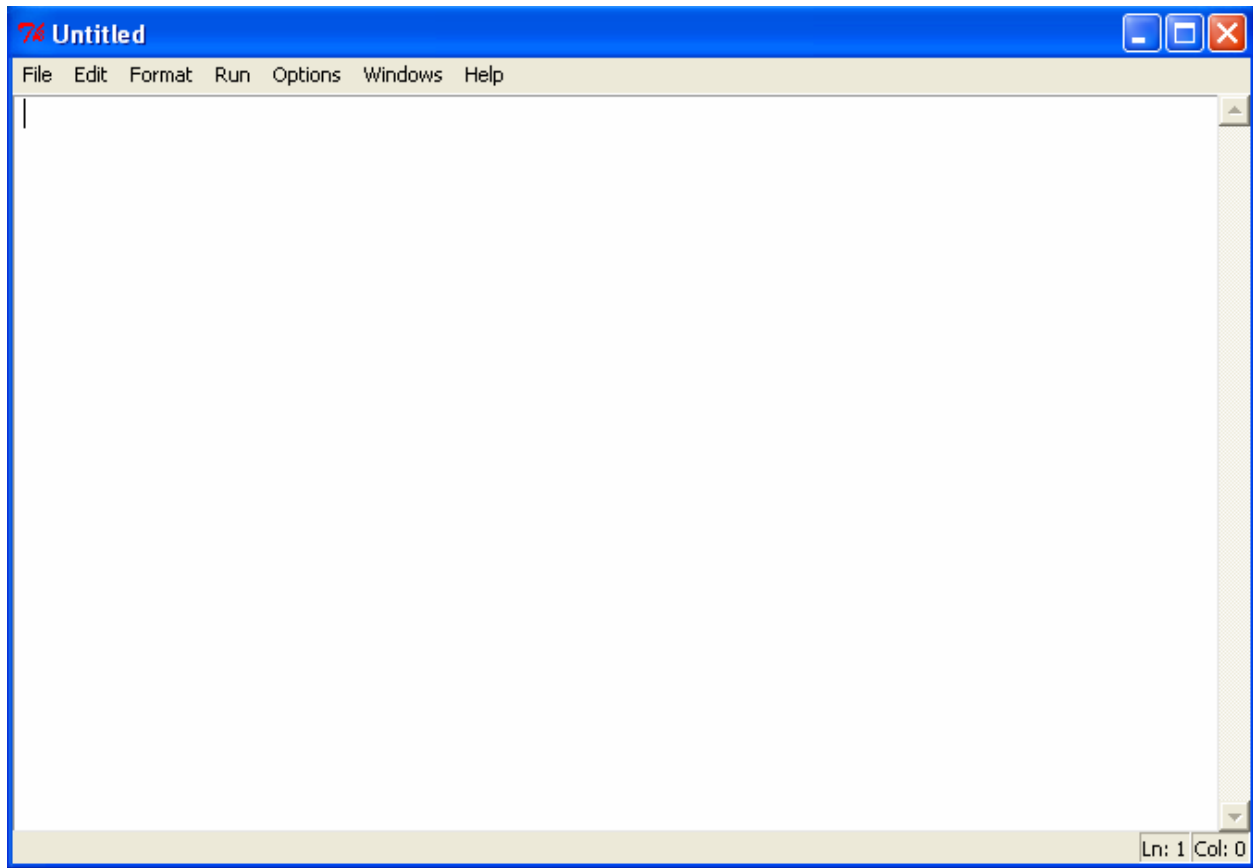
```
Python 2.6.5 (r265:79096, Mar 19 2010, 21:48:26) [MSC v.1500 32 bit (Intel)] on
win32
Type "copyright", "credits" or "license()" for more information.

*****
Personal firewall software may warn about the connection IDLE
makes to its subprocess using this computer's internal loopback
interface.  This connection is not visible on any external
interface and no data is sent to or received from the Internet.
*****

IDLE 2.6.5
>>> |
```

Burada gördüğümüz >>> işaretinden de anlayabileceğimiz gibi, bu ekran aslında Python'ın etkileşimli kabuğunun ta kendisidir... Dilerseniz, etkileşimli kabukta yapacağınız işlemleri bu ekranda da yapabilirsiniz. Ama şu anda bizim amacımız etkileşimli kabukla oynamak değil. Biz Python programlarımızı yazabileceğimiz bir metin düzenleyici arıyoruz.

Burada *File* menüsü içindeki *New Window* düğmesine tıklayarak aşağıdaki ekrana ulaşıyoruz:

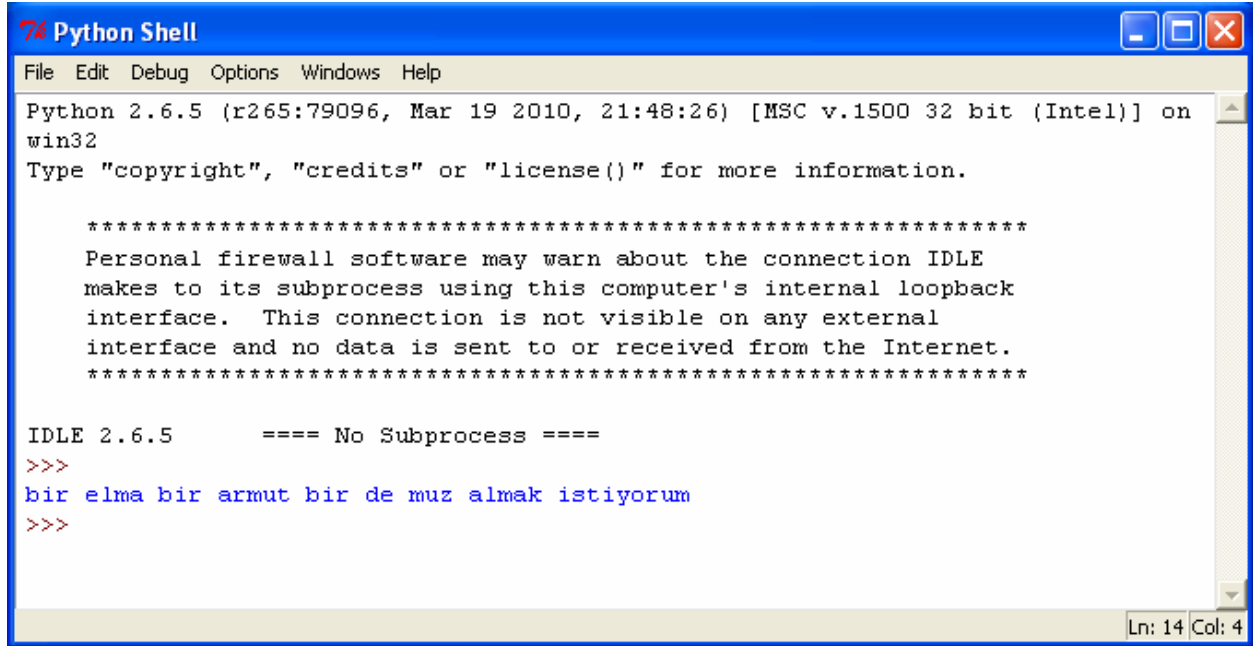


İşte Python kodlarını yazacağımız yer burası. Şimdi bu boş sayfaya şu kodları ekliyoruz:

```
a = "elma"
b = "armut"
c = "muz"

print "bir", a, "bir", b, "bir de", c, "almak istiyorum"
```

Kodlarımızı yazdıktan sonra yapmamız gereken şey dosyayı bir yere kaydetmek olacaktır. Bunun için *File/Save as* yolunu takip ederek dosyayı *deneme.py* adıyla masaüstüne kaydediyoruz. Dosyayı kaydettikten sonra *Run/Run Module* yolunu takip ederek veya doğrudan F5 tuşuna basarak yazdığımız programı çalıştırabiliriz. Bu durumda şöyle bir görüntü elde ederiz:



```
Python 2.6.5 (r265:79096, Mar 19 2010, 21:48:26) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.

*****
Personal firewall software may warn about the connection IDLE
makes to its subprocess using this computer's internal loopback
interface. This connection is not visible on any external
interface and no data is sent to or received from the Internet.
*****

IDLE 2.6.5      ==== No Subprocess ====
>>>
bir elma bir armut bir de muz almak istiyorum
>>>
```

Eğer programınızı doğrudan komut satırını kullanarak çalıştırmak isterseniz şu işlemleri yapın:

1. *Başlat/Çalıştır* yolunu takip edip, açılan pencereye “cmd” yazın ve ENTER tuşuna basın.
2. Şu komutu vererek, masaüstüne, yani dosyayı kaydettiğiniz yere gelin:

```
cd Desktop
```

Windows’ta komut satırı ilk açıldığında *C:\Documents and Settings\kullanici\_adi* gibi bir dizinin içinde olacaksınız. Dolayısıyla yukarıda gösterdiğimiz *cd Desktop* komutu sizi masaüstünün olduğu dizine götürecektir. Tabii eğer siz komut satırını farklı bir dizin içinde açmışsanız tek başına yukarıdaki komutu vermeniz sizi masaüstüne götürmez. Öyle bir durumda şuna benzer bir komut vermeniz gerekir:

```
cd C:/Documents and Settings/Kullanici_adi/Desktop
```

Masaüstüne geldikten sonra şu komutu vererek programınızı çalıştırabilirsiniz:

```
python deneme.py
```

Tabii ben burada sizin daha önce anlattığımız şekilde Python’ı YOL’a eklediğinizi varsaydım. Eğer Python’ı YOL’a eklemediyseniz programınızı çalıştırmak için şu yöntemi de kullanabilirsiniz:

```
c:/python26/python deneme.py
```

Eğer her şey yolunda gitmişse şu çıktıyı almamız lazım:

```
bir elma bir armut bir de muz almak istiyorum
```

Gördüğünüz gibi gayet basit.

Bu arada birkaç defadır “YOL” diye bir şeyden söz ediyoruz. Hatta geçen bölümde Python’ı nasıl YOL’a ekleyeceğimizi de öğrendik. Peki ama nedir bu YOL denen şey? Dilerseniz biraz bu konuyu inceleyelim. Zira ileride bir Python programı yazıp bu programı kullanıcılarınıza sunmak istediğinizde bu bilgiler çok işinize yarayacak:

Bir programın veya dosyanın yolu, kabaca o programın veya dosyanın içinde yer aldığı dizindir. Örneğin Internet Explorer programının yolu *C:\Program Files\Internet Explorer\iexplore.exe*'dir. Dolayısıyla şu komutu vererek IE'yi komut satırından başlatabilirsiniz:

```
"C:\Program Files\Internet Explorer\iexplore.exe"
```

Başka bir örnek vermek gerekirse, *hosts* dosyasının yolu *C:\WINDOWS\system32\drivers\etc\hosts*'tur. Dolayısıyla bu dosyanın içeriğini görüntülemek için şu komutu verebilirsiniz:

```
notepad "C:\WINDOWS\system32\drivers\etc\hosts"
```

Python'ın yolu ise *C:\Python26\python.exe*'dir. Dolayısıyla Python'ı çalıştırmak için şu komutu verebiliyoruz:

```
C:\Python26\python.exe
```

Bu "yol" kelimesinin bir de daha özel bir anlamı bulunur. Bilgisayar dilinde, çalıştırılabilir dosyaların (.exe dosyaları çalıştırılabilir dosyalardır) içinde yer aldığı dizinlere de özel olarak YOL adı verilir ve bu anlamda kullanıldığında "yol" kelimesi genellikle büyük harfle yazılır.

İşletim sistemleri, bir programı çağırdığımızda, söz konusu programı çalıştırabilmek için bazı özel dizinlerin içine bakar. Çalıştırılabilir dosyalar eğer bu özel dizinler içinde iseler, işletim sistemi bu dosyaları bulur ve çalıştırılmalarını sağlar. Böylece biz de bu programları doğrudan isimleriyle çağırabiliriz. Şimdi bu konuyu daha iyi anlayabilmek için birkaç deneme yapalım. *Başlat/Çalıştır* yolunu takip edip, açılan pencerede cmd komutunu vererek MS-DOS'u başlatalım ve orada şöyle bir komut yazalım:

```
echo %PATH%
```

Bu komutun çıktısı şuna benzer bir şey olacaktır:

```
C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;
```

Bu çıktı bize YOL değişkeni (İngilizce'de *PATH variable*) dediğimiz şeyi gösteriyor. İşletim sistemimiz, bir programı çağırmak istediğimizde öncelikle yukarıda adı verilen dizinlerin içini kontrol edecektir. Eğer çağırdığımız programın çalıştırılabilir dosyası bu dizinlerden herhangi birinin içinde ise o programı ismiyle çağırabiliyoruz.

---

**Not:** YOL dizinlerinin birbirlerinden noktalı virgül ile ayrıldığına dikkat edin! Dolayısıyla, YOL'a yeni bir dizin eklemek isterseniz, sizin de bu dizini önceki dizinden noktalı virgül ile ayırmanız gerekir.

---

Örneğin *notepad.exe* *C:\Windows* dizini altında bulunan bir programdır. Bu dizin, YOL üstünde yer aldığı için, Notepad programını doğrudan ismiyle çağırabiliriz. Komut satırında notepad komutunu vererek bunu kendiniz de test edebilirsiniz. Aynı şekilde *calc.exe* de *C:\Windows\System32* klasörü altında bulunan bir programdır. Bu klasör de YOL üstünde olduğu için, Calc programını doğrudan ismiyle çağırabiliyoruz...

Şimdi eğer biz de yazdığımız programı doğrudan ismiyle çağırabilmek istiyorsak programımızı `echo %PATH%` çıktısında verilen dizinlerden birinin içine kopyalamamız gerekiyor. Mesela programımızı *C:\Windows* dizini içine kopyalayalım.

Şimdi komut satırında:

```
deneme.py
```

yazınca programımızın çalıştığını görmemiz lazım.

Program dosyamızı YOL üstünde bulunan dizinlerden birine eklemek yerine, dosyamızın içinde bulunduğu dizini YOL'a eklemek de mümkün. Bunun için MS-DOS'ta şöyle bir komut vermemiz gerekiyor:

```
PATH=%PATH%;%USERPROFILE%\Desktop
```

Bu şekilde masaüstü dizinini YOL'a eklemiş olduk. İsterseniz;

```
echo %PATH%
```

komutuyla masaüstünüzün YOL üstünde görünüp görünmediğini kontrol edebilirsiniz. Bu sayede artık masaüstünde bulunan çalıştırılabilir dosyalar da kendi adlarıyla çağrılabilirler. Ancak masaüstünü YOL'a eklediğinizde, masaüstünüz hep YOL üstünde kalmayacak, mevcut MS-DOS oturumu kapatılınca her şey yine eski haline dönecektir. Eğer masaüstü dizinini kalıcı olarak YOL'a eklemek isterseniz, daha önce gösterdiğimiz şekilde, bu dizini Sistem Değişkenleri içindeki PATH listesine yazmanız gerekir.

Burada hemen şöyle bir soru akla geliyor: Notepad ve Calc gibi programları, hiç .exe uzantısını belirtmeden doğrudan isimleriyle çağırabildik. Ama mesela deneme.py adlı programımızı çalıştırmak için .py uzantısını da belirtmemiz gerekti. Peki bu durumun nedeni nedir?

Windows'ta PATHEXT adlı özel bir değişken vardır. Bu değişken, sistemin çalıştırılabilir kabul ettiği uzantıları tutar. Şu komutu verelim:

```
echo %PATHEXT%
```

Buradan şu çıktıyı alıyoruz:

```
.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH
```

Windows'ta eğer bir program yukarıda görülen uzantılardan birine sahipse çalıştırılabilir olarak kabul edilecektir. Gördüğümüz gibi, .exe de bu uzantılardan biri. Dolayısıyla bu uzantıya sahip bir dosyayı, uzantısını belirtmeden de çağırabiliyoruz. Eğer isterseniz bu listeye .py uzantısını da ekleyebilirsiniz. Bunun için şu işlemleri yapabilirsiniz:

1. Denetim Masasında "Sistem" simgesine çift tıklayın.
2. "Gelişmiş" sekmesine girin ve "Ortam Değişkenleri" düğmesine basın.
3. "Sistem Değişkenleri" bölümünde "PATHEXT" öğesini bulup buna çift tıklayın.
4. En son girdi olan .WSH'den sonra ";.PY" girdisini ekleyin.
5. TAMAM'a basıp çıkın.

Böylece artık .py uzantılı dosyaları da, uzantı belirtmeden çalıştırabilirsiniz...

Şimdiye kadar bir Python programını nasıl yazacağımızı ve IDLE'ı ya da MS-DOS'u kullanarak bu programı nasıl çalıştıracığımızı öğrendik. Peki yazdığımız bir programı çift tıklayarak çalıştırmaz mıyız? Elbette çalıştırabiliriz.

Python Windows'a kurulurken kendini kütüğe (*Registry*) kaydeder. Dolayısıyla Windows Python programlarını nasıl çalıştırması gerektiğini bilir. Hatta bu sayede Windows üzerinde yazdığımız Python programlarını, **programın bulunduğu dizin içinde** sadece ismini kullanarak çalıştırmamız da mümkündür. Yani yazdığımız programı masaüstüne kaydettiğimizi varsayarsak ve şu komutla da masaüstüne geldiğimizi düşünersek:

```
cd C:/Documents and Settings/Kullanici_adi/Desktop
```

Masaüstüne geldikten sonra şu komutu vererek programınızı çalıştırma imkanına sahibiz:

```
deneme.py
```

Ancak bu komutun işe yarayabilmesi için, MS-DOS'ta o anda programın bulunduğu dizin içinde olmamız gerek. Yani eğer programınız *Belgelerim* dizini içinde ise ve siz de MS-DOS'ta *C:/Documents and Settings/Kullanici\_adi/Desktop* dizini altındaysanız bu komut bir işe yaramayacaktır. Yazdığınız programı konum farketmeksizin her yerden çağırabilmek istiyorsanız, programınızı biraz önce anlattığımız şekilde YOL'a eklemelisiniz... Neyse... Biz şimdi tekrar konumuza dönelim.

Ne diyorduk? Evet, yazdığımız programları çift tıklayarak çalıştıramaz mıyız? Tabii ki çalıştırabiliriz. Python Windows'a kurulurken kendini kütüğe kaydettiği için, yazdığımız .py uzantılı dosyaya çift tıkladığımızda programımız çalışacaktır. Ancak burada şöyle bir problem var. Dosyamıza çift tıkladığımızda programımız çalışır çalışmasına, ama bu işlem o kadar hızlı olup biter ki, biz sadece arkada son hızla siyah bir komut satırının açılıp kapandığını görürüz... Aslında programımız çalışıyor, ama çok hızlı bir şekilde çalışıp sona erdiği için biz programımızın çalıştığını göremiyoruz. Programımızın çalıştığını görebilmemiz için o siyah komut satırının kapanmasına engel olmamız gerekir. Bunu da, programımızın en son satırına şu ilaveyi yaparak hallediyoruz:

```
raw_input()
```

Yani kodlarımız şöyle görünmeli:

```
a = "elma"
b = "armut"
c = "muz"

print "bir", a, "bir", b, "bir de", c, "almak istiyorum"

raw_input()
```

Son satıra eklediğimiz bu *raw\_input()* ifadesinin ne olduğunu ve tam olarak ne işe yaradığını birkaç bölüm sonra inceleyeceğiz. Şimdilik siz sadece bu satırın, MS-DOS'un kapanmasını engellediğini bilin. Bu komut sayesinde MS-DOS siz ENTER tuşuna basana kadar açık kalacaktır...

Ancak bu noktada çok önemli bir uyarı yapmamız gerekiyor. Python'la bir program geliştirirken, programlarınızı çalıştırmak için asla bu çift tıklama yöntemini kullanmayın. Programlarınızı her zaman komut satırından çalıştırın. Böylece, eğer programınızda bir hata varsa, bu hatanın nereden kaynaklandığını anlayabilirsiniz. Ama eğer programınızı çift tıklayarak çalıştırırsanız hatayı görme şansınız olmaz. Buna bir örnek verelim.

Diyelim ki bir üçgenin alanını hesaplayan bir program yazacaksınız. Bir üçgenin alan formülü şudur:

```
alan = taban x (yukseklik / 2)
```

Kodlarımız şöyle olmalı:

```
yukseklik = 4
taban = 6

alan = taban * (yukseklik / 2)

print alan
```

Burada yaptığımız şey şu:

1. Öncelikle *yukseklik* adlı bir değişken tanımladık. Bu değişkenin değeri 4

2. Ardından taban adlı başka bir değişken daha tanımladık. Bu değişkenin değeri ise 6
3. Üçgenin alanını hesaplamamızı sağlayan formülümüzü yazıyoruz:  $\text{alan} = \text{taban} * (\text{yukseklik} / 2)$
4. Son olarak da alan adlı değişkenin değerini *print* komutu yardımıyla ekrana çıktı olarak veriyoruz.
5. Bu programı çalıştırdığımızda alacağımız sonuç 12 olacaktır...

Şimdi şöyle bir düşünün. Diyelim ki biz bu programı yazarken ufak bir hata yaptık ve şöyle bir şey çıkardık ortaya:

```
yukseklik = 4
taban = 6

ala = taban * (yukseklik / 2)

print alan
```

Eğer bu programı çift tıklayarak çalıştırırsanız, programdaki hatanın nereden kaynaklandığını anlayamazsınız. Çünkü MS-DOS ekranı son hızla açılıp kapanacaktır. Üstelik son satıra *raw\_input()* eklemeniz de hiç bir işe yaramaz. Bu satır sadece hatasız programların çift tıklanarak çalışmasını sağlar. Çünkü Python kodları yukarıdan aşağıya doğru okur. Bir hatayla karşılaştığında da kodların geri kalanını okumadan programdan çıkar. Dolayısıyla, *raw\_input()* satırından önce bir hata varsa, Python bu satıra ulaşmadan programı kapatacak, bu sebeple MS-DOS satırının kapanmasını önleyen kod hiçbir zaman çalışmamış olacaktır...

Ama eğer aynı programı komut satırından çalıştırmayı denerseniz hatanın nereden kaynaklandığı konusunda bir fikir edinebilirsiniz. Bu program şöyle bir hata verecektir:

```
Traceback (most recent call last):
  File "deneme.py", line 6, in <module>
    print alan
NameError: name 'alan' is not defined
```

Bu hata bize, “alan” diye bir şeyin tanımlanmadığını söylüyor. Kodlarımıza dikkatli bakınca görüyoruz ki, hakikaten “alan” diye bir değişken tanımlamamışız. Çünkü formülde “alan” yazacağımıza yanlışlıkla “ala” yazmışız!

Böylece bir bölümü daha bitirmiş olduk. Yalnız bu noktada şu hatırlatmayı yapmadan geçmeye-  
lim. Yukarıda verdiğimiz bilgiler son derece önemlidir. Sadece Python’la değil, hangi dille programlama yapıyor olursanız olun, yukarıda anlattığımız şeyleri özümsemiş olmalısınız. İyi bir programcı, üzerinde program geliştireceği işletim sistemini yakından tanımalıdır.

### 3.3 Türkçe Karakter Kullanımı

Şimdiye dek Python hakkında epey bilgi edindik. Dilerseniz elimizi alıştırmak için santigratı fahrenheit’e çeviren bir betik yazalım ve neler olduğuna bakalım:

```
santigrat = 22

fahrenheit = santigrat * (9.0/5.0) + 32

print "%s santigrat derece %s fahrenheit'e \n"
      "karşılık gelir." %(santigrat, fahrenheit)
```



Bu programı çalıştırdığınızda biraz hayal kırıklığına uğramış olabilirsiniz. Çünkü yukarıdaki kodlar şuna benzer bir hata mesajı almanıza sebep olacaktır:

```
File "deneme.py", line 6
SyntaxError: Non-ASCII character '\xc5' in file deneme.py
on line 6, but no encoding declared; see
http://www.python.org/peps/pep-0263.html for details
```

Bu hata mesajını almamızın nedeni, programımız içinde kullandığımız “ı, ş” gibi Türkçe karakterler. Python yazdığımız kodlar içindeki bu Türkçe karakterler nedeniyle bize bu hata mesajını gösterecektir. Böyle bir hata mesajı almamak için kodlarımızın arasına şu satırı eklememiz gerekir:

```
# -*- coding: utf-8 -*-
```

Windows kullanıcıları ise utf-8 yerine cp1254 adlı karakter kodlamasını kullanmayı tercih edebilir:

```
# -*- coding: cp1254 -*-
```

Böylelikle kullandığımız karakter tipini Python’a tanıtmış oluyoruz. İlerde *ascii*, *unicode* ve *Python* konusunu işlerken Python’ın Türkçe karakterlere neden böyle davrandığını inceleyeceğiz. Ama siz şimdilik bu konuyu çok fazla dert edinmeyin.

Programımızın en son hali şöyle olacak:

```
# -*- coding: utf-8 -*-

santigrat = 22

fahrenheit = santigrat * (9.0/5.0) + 32

print "%s santigrat derece %s fahrenheitta \
karşılık gelir." %(santigrat, fahrenheit)
```

Gördüğünüz gibi bu defa programımız Türkçe karakterleri düzgün bir şekilde gösterebildi. Bu arada isterseniz yukarıdaki kodları biraz inceleyelim:

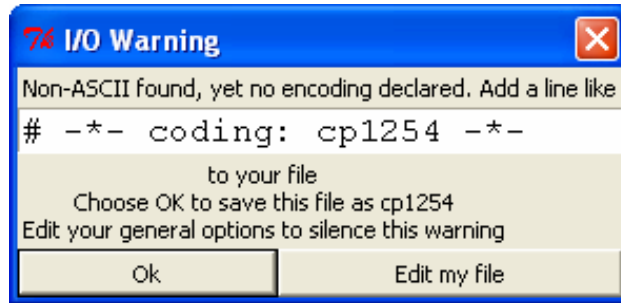
Burada öncelikle `# -*- coding: utf-8 -*-` satırını yazdık. Bu satır sayesinde kullanacağımız kodlama biçimini Python’a tanıtmış olduk. Böylece kodlarımız içinde Türkçe karakter kullanabileceğiz. Bu satırın Windows’ta `# -*- coding: cp1254 -*-` biçimini alacağını biliyorsunuz.

Ardından `santigrat` adlı bir değişken tanımladık ve bu değişkenin değerini 22 olarak belirledik.

Daha sonra da `fahrenheit`ı `santigrat`a çevirebilmek için gereken formülümüzü yazdık. Formülde 9/5 yerine 9.0/5.0 yazdığımıza dikkat edin. Böylece 9 sayısını 5 sayısına hassas bir şekilde bölebilmiş olduk. Aksi halde formülümüz sonucu hatalı hesaplayacaktır.

Son olarak da 22 santigrat derecenin kaç fahrenheitta karşılık geldiğini düzgün bir cümle halinde kullanıcıya gösterdik. Burada karakter dizisini nasıl biçimlendirdiğimize dikkat edin.

IDLE üzerinde çalışan Windows kullanıcıları bu programı `# -*- coding: cp1254 -*-` satırını yazmadan kaydetmeye çalıştıklarında şöyle bir uyarı penceresiyle karşılaşacaklar:



IDLE, karakter kodlamasını belirtmeniz konusunda sizi uyarıyor. Eğer burada “Edit My File” düğmesine basacak olursanız, IDLE programınızın ilk satırına sizin yerinize # -\*- coding: cp1254 -\*- komutunu ekleyecektir...

Yukarıdaki kodlarda dikkat etmemiz gereken başka bir nokta da kodları nasıl böldüğümüzdür. Daha önceki derslerimizde de bahsettiğimiz gibi Python’da kodların görünümü de önem taşır. Kodların çirkin bir şekilde sağa doğru uzamasını engellemek amacıyla “\” kaçış dizisini kullanarak kodlarımızı alt satıra kaydırdık. Eğer arzu ederseniz, yukarıdaki kodları şu şekilde de bölebilirsiniz:

```
# -*- coding: cp1254 -*-

santigrat = 22

fahrenheit = santigrat * (9.0/5.0) + 32

print ("%s santigrat derece %s fahrenheitta karşılık gelir."
        %(santigrat, fahrenheit))
```

Burada kullandığımız bölme biçimine çok dikkat edin. Burada karakter dizisinin tamamını parantez içine aldık:

```
print ("%s santigrat derece %s fahrenheitta karşılık gelir."
        %(santigrat, fahrenheit))
```

Yalnız bu yöntemi, bir karakter dizisini ortadan bölmek için kullanamazsınız. Bu yöntemi kullanabilmeniz için “%s” işaretlerinin kullanıldığı bir kodla karşı karşıya olmanız ve satırı “%” işaretinden bölmeniz gerekir. Yani şu kullanımlar yanlıştır:

```
print ("Ben çok uzun bir karakter dizisiyim. 0 yüzden
iki satıra bölünmem gerekir.")
```

veya:

```
print ("%s santigrat derece %s fahrenheitta
karşılık gelir."%(santigrat, fahrenheit))
```

Python satır bölme konusunda pek çok alternatif sunar. Hatta şöyle bir şey dahi yapmanıza izin verir. Dikkatlice inceleyin:

```
print ("Ben çok uzun bir karakter dizisiyim. 0 yüzden"
        " iki satıra bölünmem gerekir.")
```

ya da:

```
print ("%s santigrat derece %s fahrenheitta"
        " karşılık gelir."%(santigrat, fahrenheit))
```

Aslında şu son iki yöntemi biliyor olmanız lazım. Hatırlarsanız ilk derslerimizde karakter dizilerini birleştirmeyi öğrenirken şöyle bir örnek vermiştik:

```
>>> print "İstanbul " "Çeliktepe"
```

Yukarıdaki örneğin de bundan hiçbir farkı yoktur. Biz şimdiki örneğimizde alt satıra geçtiğimiz için bütün karakter dizilerini parantez içine aldık, o kadar... Yani önceki derste gördüğümüz örneği iki satıra bölmek isteseydik şöyle bir kod yazacaktık:

```
>>> print ("İstanbul "  
... "Çeliktepe")
```

Bu yöntemlerden herhangi birini tercih edebilirsiniz. Ama etrafta göreceğiniz programlarda hepsiyle karşılaşma ihtimaliniz var...

### 3.4 MS-DOS'ta Türkçe Karakter Problemi

Yukarıda kodlama biçimi belirterek Türkçe karakterleri nasıl göstereceğimizi inceledik. Türkçe karakterlerin problem çıkardığı bir başka ortam da MS-DOS'tur. Windows'un komut satırı Türkçe karakterleri düzgün göstermekte zorlanabiliyor... Bu bölümde bu sorunu nasıl aşabileceğimizi inceleyeceğiz.

Mesela şu kodları *deneme.py* adlı bir dosyaya kaydettiğimizi varsayalım:

```
# -*- coding: cp1254 -*-  
  
santigrat = 22  
  
fahrenheit = santigrat * (9.0/5.0) + 32  
  
print ("%s santigrat derece %s fahrenheit karşılık gelir."  
%(santigrat, fahrenheit))
```

Bu dosyayı MS-DOS'ta python *deneme.py* komutunu vererek çalıştırmak istediğimizde şöyle bir çıktı alıyoruz:

```
22 santigrat derece 71.6 fahrenheit kar?²l²k gelir.
```

Gördüğünüz gibi bütün Türkçe karakterler birbirine girmiş durumda. Bunun temel sebebi Windows'ta komut satırının öntanımlı yazı tipinin Türkçe karakterleri düzgün gösterememesi. Bu durumu düzeltmek için şu adımları takip ediyoruz:

- Önce MS-DOS ekranını açıyoruz.
- Ardından şu komutu veriyoruz:

```
chcp 1254
```

- Daha sonra pencere başlığına sağ tıklayıp "özellikler"e giriyoruz.
- Yazı Tipi sekmesi içinde yazı tipini "Lucida console" olarak değiştiriyoruz.
- Tamam'a basıyoruz.
- Karşımıza çıkan pencerede, "özellikleri, aynı başlıkla ileride oluşturulacak pencereler için kaydet" seçeneğini işaretliyoruz.

Şimdi en başta verdiğimiz kodları çalıştırdığımızda şu çıktıyı alıyoruz:

22 santigrat derece 71.6 fahrenheitte karşılık gelir.

Böylece çok önemli bir konuyu daha geride bırakmış olduk. Üstelik bu noktaya kadar Python'la ilgili epey bilgi de edindik. Dilerseniz şimdi bölüm sorularımıza bir göz atalım ve ondan sonra yeni ve yine çok önemli bir konuya geçelim.

### 3.5 Bölüm Soruları

1. Komut satırında masaüstünün bulunduğu konuma gelebilmek için hangi komutu kullanmanız gerekiyor?
2. Eğer Windows kullanıyorsanız *C:\Program Files* dizini altında, GNU/Linux kullanıyorsanız */usr/bin* dizini altında bir komut ekranı açın.
3. Eğer Windows kullanıyorsanız, önce <http://www.videolan.org/vlc/> adresini ziyaret edin ve oradan VLC adlı programı indirin. Şimdi indirdiğiniz bu programı normal bir şekilde kurun. Ardından, bilgisayarınızda VLC klasörünü bulup, bu klasör içindeki *vlc.exe*'yi YOL'a (PATH) ekleyin. Son olarak, *vlc* komutu ile programı MS-DOS'tan çalıştırın.
4. Hem Windows'ta hem de GNU/Linux'ta şu betiği herhangi bir konumdan sadece ismi ile çağrılabilir hale getirin. Yani betiğe *hesaplayici.py* adını verdiğiniz varsayarsak, komut satırından sadece *hesaplayici* komutu girilerek programınız çalıştırılabilir:

```
yukseklık = 4
taban = 6

alan = taban * (yukseklık / 2)

print alan
```

5. Eğer Windows kullanıyorsanız, 4. soruda yazdığınız Python programının çift tıklamayla çalışmasını sağlayın.
6. Aşağıdaki bilgileri kullanarak bir Python programı yazın. Buradaki uzun karakter dizisini yazdırırken, farklı yöntemler kullanarak karakter dizisini uygun yerinden bölmeye dikkat edin:

```
maaş: 3500 TL, ünvan: İş Geliştirme Uzmanı,
çalıştığı süre: 5

"Ahmet Bey ... TL maaşla çalışan bir ...'dır.
Kendisi ... yıldır bu görevde bulunmaktadır."
```

7. Eğer GNU/Linux kullanıyorsanız, yazdığınız bir Python programının ilk satırına *#!/usr/bin/env python* yerine farklı bir şey yazın. Mesela şunu:

```
#!/usr/python
```

Şimdi bu programı *chmod* komutuyla çalıştırılabilir hale getirin ve programı adıyla çağırmaı deneyin. Ne tür bir hata mesajı alıyorsunuz?

Aynı programı bir de *#!/usr/bin/env python* gibi bir satır yazmadan çalıştırmayı deneyin. Bu defa nasıl bir hata mesajı alıyorsunuz? Sizce bu hata mesajlarını almanızın sebebi nedir?

---

# Kullanıcıyla İletişim: Veri Alış-Verişi

---

Bu bölümde de Python'daki önemli konulara değinmeye devam ediyoruz. Bu defaki konumuz “kullanıcıyla iletişim”.

Dikkat ettiyseniz, şimdiye dek sadece tek yönlü bir programlama faaliyeti içinde bulunduk. Yani yazdığımız kodlar kullanıcıyla herhangi bir etkileşim içermiyordu. Ama artık, Python'da bazı temel şeyleri öğrendiğimize göre, kullanıcıyla nasıl iletişim kurabileceğimizi de öğrenmemizin vakti geldi. Bu bölümde kullanıcılarımızla nasıl veri alış-verişinde bulunacağımızı göreceğiz.

Hatırlarsanız geçen bölümde şöyle bir örnek vermiştik:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

santigrat = 22

fahrenheit = santigrat * (9.0/5.0) + 32

print ("%s santigrat derece %s fahrenheit karşılık gelir."
%(santigrat, fahrenheit))
```

Dikkat ederseniz burada santigrat değişkenini kendimiz kod içinde elle belirledik. Dolayısıyla formüle giren bütün sayılar zaten belli olduğu için bu kodların nasıl bir çıktı vereceğini de biliyoruz. Böyle bir kod yazımının ne kadar sıkıcı olduğu ortada... Bu santigrat değişkenini kendimiz elle yazmak yerine, kullanıcıya sorarsak ve kullanıcının girdiği miktara göre hesaplama yaparsak ne güzel olurdu, değil mi? Böylece her defasında farklı bir değer alıp bunu hesaplayabilen esnek bir programımız olurdu. İşte bu bölümde böyle bir programı nasıl yazabileceğimizi öğreneceğiz.

Python'da kullanıcıdan birtakım veriler alabilmek, yani kullanıcıyla iletişime geçebilmek için iki tane fonksiyondan faydalanılır: *raw\_input()* ve *input()* (Bu “fonksiyon” kelimesine takılmayın. Birkaç bölüm sonra fonksiyonun ne demek olduğunu inceleyeceğiz.) Bu iki fonksiyon birbirlerinden bazı farklılıklar gösterir. Bunlardan öncelikle ilkinde bakalım.

## 4.1 raw\_input() fonksiyonu

*raw\_input()* fonksiyonu kullanıcılarımızın klavye aracılığıyla girdiği verileri almamızı sağlar. İsterseniz bu fonksiyonu tarif etmeye çalışmak yerine hemen bununla ilgili bir örnek verelim. Öncelikle boş bir Kwrite (veya Gedit ya da IDLE) belgesi açalım. Her zaman yaptığımız gibi, ilk satırımızı ekleyelim belgeye:

```
#!/usr/bin/env python
```

**Not:** Bu satır sadece GNU/Linux kullanıcılarını ilgilendirir. Windows kullanıcıları bu satırı yazmasalar da olur. Bu satırın Windows'ta hiçbir etkisi yoktur.

Şimdi *raw\_input()* fonksiyonuyla kullanıcıdan bazı bilgiler alacağız. Mesela kullanıcıya bir parola sorup kendisine teşekkür edelim...

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

raw_input("Lütfen parolanızı girin:")
print "Teşekkürler!"
```

*raw\_input()* fonksiyonunu nasıl kullandığımıza çok dikkat edin. Parantez içinde yazdığımız karakter dizisi, kullanıcıdan ne beklediğimizi göstermemizi sağlıyor. Buna göre biz bu kodlarla kullanıcıdan parolasını girmesini isteyeceğiz.

Şimdi bu belgeyi *deneme.py* ismiyle kaydediyoruz. Daha sonra bir konsol ekranı açıp, programımızın kayıtlı olduğu dizine geçerek şu komutla programımızı çalıştırıyoruz:

```
python deneme.py
```

Elbette siz isterseniz daha önce anlattığımız şekilde dosyaya çalıştırma yetkisi vererek ve gerekli düzenlemeleri yaparak programınızı doğrudan ismiyle de çağırabilirsiniz. Bu sizin tercihinize kalmış.

Program çalıştığında, kullanıcı "Lütfen parolanızı girin: " cümlesini görecektir ve böylece kendisinden ne beklendiğini anlayacak. Elbette *raw\_input()* fonksiyonundaki parantezi boş da bırakabilirsiniz. Ancak çoğu zaman, parantez içine açıklayıcı bir şeyler yazmak isteyeceksiniz. Çünkü aksi halde programınız çalıştığında kullanıcınız ne yapması gerektiğini anlayamayacaktır.

Windows kullanıcıları esasında bu fonksiyonu önceki derslerimizden hatırlayacaktır. Bu fonksiyonu Windows'ta çift tıklayarak çalıştırmak istediğimiz programlarda MS-DOS ekranının program sonunda kapanmasını önlemek için kullanmıştık. O örnekte gösterdiğimiz programı şu şekilde yazarsak herhalde *raw\_input()* fonksiyonunun oradaki görevi daha belirgin bir şekilde ortaya çıkacaktır:

```
# -*- coding: cp1254 -*-

a = "elma"
b = "armut"
c = "muz"

print "bir", a, "bir", b, "bir de", c, "almak istiyorum"

raw_input("Programdan çıkmak için ENTER tuşuna basın.")
```

Biz o dersteki örnekte parantez içini boş bırakmıştık...

Dilerseniz biraz önce yazdığımız parola soran programı biraz geliştirelim. Mesela programımız şu işlemleri yapsın:

1. Program ilk çalıştırıldığında kullanıcıya parola sorsun,
2. Kullanıcı parolasını girdikten sonra programımız kullanıcıya teşekkür etsin,
3. Bir sonraki satırda kullanıcı tarafından girilen bu parola ekrana yazdırılsın,
4. Kullanıcı daha sonraki satırda parolanın yanlış olduğu konusunda uyarılsın.

Şimdi kodlarımızı yazmaya başlayabiliriz. Öncelikle yazacağımız kodlardan bağımsız olarak girmemiz gereken bilgileri ekleyelim:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

Şimdi `raw_input()` fonksiyonuyla kullanıcıya parolasını soracağız. Ama isterseniz bu `raw_input()` fonksiyonunu bir değişkene atayalım:

```
parola = raw_input("Parolanızı girin:")
```

Şimdi kullanıcıya teşekkür ediyoruz:

```
print "Teşekkürler!"
```

Kullanıcı tarafından girilen parolayı ekrana yazdırmak için şu satırı ekliyoruz:

```
print parola
```

Biraz önce `raw_input()` fonksiyonunu neden bir değişkene atadığımızı anladınız sanırım. Bu sayede doğrudan parola değişkenini çağırarak kullanıcının yazdığı parolayı ekrana dökabiliyoruz.

Şimdi de kullanıcıya parolasının yanlış olduğunu bildireceğiz:

```
print "Ne yazık ki doğru parola bu değil"
```

Programımızın son hali şöyle olacak:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

parola = raw_input("Parolanızı girin:")

print "Teşekkürler!"
print parola
print "Ne yazık ki doğru parola bu değil."
```

İsterseniz son satırda şu değişikliği yapabiliriz:

```
print "Ne yazık ki doğru parola", parola, "değil."
```

Böylelikle, parola değişkenini, yani kullanıcının yazdığı parolayı cümlemizin içine (ya da Pythonca ifade etmek gerekirse: "karakter dizisi içine") eklemiş olduk.

Bu parola değişkenini karakter dizisi içine eklemenin başka bir yolu da kodu şu şekilde yazmaktır:

```
print "Ne yazık ki doğru parola %s değil" %(parola)
```

Bu sayede kullanıcının girdiği parolayı istediğiniz gibi biçimlendirebilirsiniz. Mesela kullanıcıdan gelen parolayı tırnak içine alarak cümle içinde daha belirgin durmasını sağlayabilirsiniz:

```
print "Ne yazık ki doğru parola '%s' değil" %(parola)
```

Şimdi `raw_input()` fonksiyonuna bir ara verip, kullanıcıdan bilgi almak için kullanabileceğimiz ikinci fonksiyondan biraz bahsedelim. Az sonra `raw_input()` fonksiyonuna geri döneceğiz.

## 4.2 input() fonksiyonu

Tıpkı `raw_input()` fonksiyonunda olduğu gibi, `input()` fonksiyonuyla da kullanıcılardan bazı bilgileri alabiliyoruz. Şu basit örneğe bir bakalım:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

a = input("Lütfen bir sayı girin:")
b = input("Lütfen başka bir sayı daha girin:")

print a + b
```

Kullanım açısından, görüldüğü gibi, `raw_input()` ve `input()` fonksiyonları birbirlerine çok benzer. Ama bunların arasında çok önemli bir fark vardır. Hemen yukarıda verilen kodları bir de `raw_input()` fonksiyonuyla yazmayı denersek bu fark çok açık bir şekilde ortaya çıkacaktır:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

a = raw_input("Lütfen bir sayı girin:")
b = raw_input("Lütfen başka bir sayı daha girin:")

print a + b
```

Bu kodları yazarken `input()` fonksiyonunu kullanırsak, kullanıcı tarafından girilen sayılar birbirleriyle toplanacaktır. Diyelim ki ilk girilen sayı 25, ikinci sayı ise 40 olsun. Programın sonunda elde edeceğimiz sayı 65 olacaktır.

Ancak bu kodları yazarken eğer `raw_input()` fonksiyonunu kullanırsak, girilen sayılar birbirleriyle toplanmayacak, sadece yan yana yazılacaklardır. Yani elde edeceğimiz şey 2540 olacaktır.

Hatırlarsanız buna benzer bir şeyle “sayılar” konusunu işlerken de karşılaşmıştık. O zaman şu örnekleri vermiştik:

```
>>> print 25 + 50
75
>>> print "25" + "50"
2550
```

İşte `raw_input()` ile `input()` arasındaki fark, yukarıdaki iki örnek arasındaki farka benzer. Yukarıdaki örneklerde, Python’ın sayılara ve karakter dizilerine nasıl davrandığını görüyoruz. Eğer



Python iki adet sayıyla karşılaşınca bu sayıları birbiriyle topluyor. Ama eğer iki adet karakter dizisiyle karşılaşınca, bu karakter dizilerini yanyana yazmakla yetiniyor...

Bütün bu açıklamalardan anlıyoruz ki `raw_input()` fonksiyonu kullanıcının girdiği verileri karakter dizisine dönüştürüyor. Bu durumu daha net görebilmek için dilerseniz şöyle bir örnek verelim:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

sayi = raw_input("Bir sayı girin. Ben size o \
sayının karesini söyleyeyim:\n")

print "girdiğiniz sayı\t\t: %s" %(sayi)
print "girdiğiniz sayının karesi: %s" % (sayi * sayi)
```

Bu kodları çalıştırdığınızda sayıyı girip ENTER tuşuna bastıktan sonra şöyle bir hata mesajı alacaksınız:

```
File "deneme.py", line 8, in <module>
    print "girdiğiniz sayının karesi: %s" % (sayi * sayi)
TypeError: can't multiply sequence by non-int of type 'str'
```

Bu durum, `raw_input()` fonksiyonunun kullanıcı verilerini karakter dizisi olarak almasından kaynaklanıyor. Karakter dizileri ile aritmetik işlem yapılamayacağı için de hata vermekten başka çaresi kalmıyor. Burada tam olarak ne döndüğünü anlayabilmek için etkileşimli kabukta verdiğimiz şu örnekleri bakabilirsiniz:

```
>>> a = "12"
>>> b = "12"
>>> print a * b

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

Gördüğümüz gibi biraz öncekiyle aynı hatayı aldık. İki karakter dizisini birbiriyle çarpamaz, bölemez ve bu karakter dizilerini birbirinden çıkaramazsınız. Çünkü aritmetik işlemler ancak sayılar arasında olur. Karakter dizileri ile aritmetik işlem yapılmaz... Yukarıdaki örneklerin düzgün çıktı verebilmesi için o örnekleri şöyle yazmamız gerekir:

```
>>> a = 12
>>> b = 12
>>> print a * b

144
```

Dolayısıyla, yukarıda hata veren kodlarımızın da düzgün çalışabilmesi için o kodları şöyle yazmamız gerek:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

sayi = input("Bir sayı girin. Ben size o \
sayının karesini söyleyeyim:\n")

print "girdiğiniz sayı\t\t: %s" %(sayi)
print "girdiğiniz sayının karesi: %s" % (sayi * sayi)
```

`raw_input()` fonksiyonundan farklı olarak `input()` fonksiyonu kullanıcıdan gelen verileri olduğu gibi alır. Yani bu verileri karakter dizisine dönüştürmez. Bu yüzden, eğer kullanıcı bir sayı

girmişse, `input()` fonksiyonu bu sayıyı olduğu gibi alacağı için, bizim bu sayıyla aritmetik işlem yapmamıza müsaade eder. Bu durumu daha iyi anlayabilmek için mesela aşağıda `raw_input()` fonksiyonuyla yazdığımız kodları siz bir de `input()` fonksiyonuyla yazmayı deneyin:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

a = raw_input("isminiz: ")
b = raw_input("soyisminiz: ")

print a, b
```

Eğer bu kodları `input()` fonksiyonuyla yazmayı denediyseniz, Python'ın ilk veri girişinden sonra şöyle bir hata verdiğini görmüşsünüzdür:

```
SyntaxError: invalid syntax
```

Etkileşimli kabukta şu komutu verdiğinizde de aynı hatayı aldığınızı göreceksiniz:

```
>>> Ahmet

SyntaxError: invalid syntax
```

Burada hata almamak için şöyle yapmak gerek:

```
>>> "Ahmet"

'Ahmet'
```

Dolayısıyla Python'ın `input()` fonksiyonuyla bu hatayı vermemesi için de tek yol, kullanıcının ismini ve soyismini tırnak içinde yazması olacaktır. Ama tabii ki normal şartlarda kimseden ismini ve soyismini tırnak içinde yazmasını bekleyemezsiniz.

Bütün bunlardan şunu anlıyoruz:

`input()` fonksiyonu, kullanıcının geçerli bir Python komutu girmesini bekler. Yani eğer kullanıcının girdiği şey Python'ın etkileşimli kabuğunda hata verecek bir ifade ise, `input()`'lu kodunuz da hata verecektir.

Dolayısıyla eğer biz programımız aracılığıyla kullanıcılardan bazı sayılar isteyeceksek ve eğer biz bu sayıları işleme sokacaksak (çıkarma, toplama, bölme gibi...) `input()` fonksiyonunu tercih edebiliriz. Ama eğer biz kullanıcılardan sayı değil de karakter dizisi girmesini istiyorsak `raw_input()` fonksiyonunu kullanacağız.

Şimdi dilerseniz bu bölümün en başında verdiğimiz fahrenheit dönüşüm programını, yeni öğrendiğimiz bilgiler yardımıyla biraz geliştirelim:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

santigrat = input("Santigrat cinsinden bir değer girin: ")

fahrenheit = santigrat * (9.0/5.0) + 32

print ("%s santigrat derece %s fahrenheit karşılık gelir."
      %(santigrat, fahrenheit))
```

Böylece elimizde gayet şık bir derece dönüşüm programı olmuş oldu.. Günün birinde santigrat dereceleri fahrenheit'e dönüştürmeniz gerekirse yukarıdaki programı kullanabilirsiniz...

## 4.3 Güvenlik Açısından input() ve raw\_input()

Yukarıda da bahsettiğimiz gibi, Python'da kullanıcıdan veri alabilmek için *input()* ve *raw\_input()* adlı iki farklı fonksiyondan faydalanıyoruz. Bu iki fonksiyonun kendilerine has görevleri ve kullanım alanları vardır. Ancak eğer yazdığınız kodlarda güvenliği de ön planda tutmak isterseniz *input()* fonksiyonundan kaçınmayı tercih edebilirsiniz. Çünkü *input()* fonksiyonu kullanıcıdan gelen bilgiyi bir komut olarak algılar. Peki bu ne anlama geliyor?

Şimdi şöyle bir kod yazdığımızı düşünün:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

sayi = input("bir sayı girin: ")

print sayi
```

Bu kodlarımızı çalıştırdığımızda "bir sayı girin: " sorusuyla karşılaşacağız. Şimdi bu soruya cevap olarak şunları yazalım:

```
2 + 2
```

Bunu yazıp ENTER tuşuna bastığımızda 4 çıktısını elde ediyoruz. Bu demek oluyor ki, *input()* fonksiyonu kullanıcıdan gelen bilgiyi bir komut olarak yorumladığı için, bu fonksiyon yardımıyla kullanıcılarımız Python kodlarını çalıştırabiliyor. Ne güzel, değil mi? Hiç de değil! Başta çok hoş bir özellik gibi görünse de bu, aslında beraberinde bazı güvenlik risklerini de getirir. Şimdi yukarıdaki programı tekrar çalıştırın ve soruya şu cevabı yazın:

```
eval("__import__('os').system('dir')")
```

ENTER tuşuna bastığınızda, o anda içinde bulunduğunuz dizin altındaki dosyaların ekranda listelendiğini göreceksiniz. Yukarıdaki satırda geçen komutları henüz öğrenmedik. Ama yukarıdaki satır yardımıyla sistem komutlarını çalıştırabildiğimizi görüyorsunuz. Burada gördüğümüz *dir* bir sistem komutudur. Bu komutun görevi bir dizin altındaki dosyaları listelemek. Mesela GNU/Linux'ta dosya silmemizi sağlayan komut *rm*, Windows'ta ise *del*'dir. Dolayısıyla yukarıdaki komutu şu şekilde yazdığımızda neler olabileceğini bir düşünün:

```
eval("__import__('os').system('rm bir_dosya')")
```

veya:

```
eval("__import__('os').system('del bir_dosya')")
```

Hatta bu satır, şimdi burada göstermek istemediğimiz, çok daha yıkıcı sonuçlar doğurabilecek sistem komutlarının çalıştırılabilmesini sağlar (bütün sabit diski silmek gibi...). Dolayısıyla eğer özellikle sunucu üzerinden çalışacak kodlar yazıyorsanız *input()* fonksiyonunu kullanırken dikkatli olmalısınız.

*input()* fonksiyonunun yukarıdaki risklerinden ötürü Python programcıları genellikle bu fonksiyonu kullanmamayı tercih eder. Bunun yerine her zaman *raw\_input()* fonksiyonu kullanılır. Zaten *raw\_input()* fonksiyonu kullanıcıyla veri alış-verişine ilişkin bütün ihtiyaçlarımızı karşılayacak özelliktedir. Ancak biz henüz *raw\_input()* fonksiyonunu tam kapasite kullanacak kadar bilgi sahibi değiliz. Birkaç bölüm sonra eksik bilgilerimizi de tamamladıktan sonra *input()* fonksiyonuna hiç ihtiyacınız olmadığını göreceksiniz. Ama biz şimdilik *input()* fonksiyonundan yararlanmaya devam edeceğiz. Çünkü düşmanımızı tanımalıyız ki düşmanımızdan korunabilelim...

## 4.4 Bölüm Soruları

1. Ana görevi doğum yılı sorup yaş hesaplamak olan bir program yazın. Yazdığınız program kullanıcıya ismiyle hitap edip, ona yaşını söyleyebilmeli.
2. Türk Lirası'nı günlük kura göre Dolar, Euro ve Sterlin'e çeviren bir program yazın. Program kullanıcıdan TL miktar alıp bu miktarı kullanıcıya Dolar, Euro ve Sterlin olarak geri bildirebilmeli.
3. 2009 yılı Nisan ayının ilk gününe ait Dolar kuru ile 2010 yılı Nisan ayının ilk gününe ait Dolar kurunu karşılaştırın ve bir yıllık değişimin yüzdesini hesaplayın.
4. Bir önceki soruda yazdığınız programı, gerekli değerleri kullanıcıdan alarak tekrar yazın.
5. `raw_input()` ile `input()` fonksiyonlarını birbiriyle karşılaştırın. `input()` fonksiyonunu kullanarak bir sisteme ne tür zararlar verebileceğinizi ve bu zararları önlemek için teorik olarak neler yapabileceğinizi düşünün.
6. Aşağıdaki iki kodu karşılaştırın ve birinci kod hatasız çalışırken, ikinci kodun hata vermesinin sebebini açıklayın.

```
>>> print "Hayır! " * 5
>>> print "Hayır! " * "Hayır! "
```

# Python'da Koşula Bağlı Durumlar

Python'da en önemli konulardan biri de kuşkusuz koşula bağlı durumlardır. İsterseniz ne demek istediğimizi bir örnekle açıklayalım. Diyelim ki Gmail'den aldığınız e-posta hesabınıza gireceksiniz. Gmail'in ilk sayfasında size bir kullanıcı adı ve parola sorulur. Siz de kendinize ait kullanıcı adını ve parolayı sayfadaki kutucuklara yazarsınız. Eğer yazdığınız kullanıcı adı ve parola doğruysa hesabınıza erişebilirsiniz. Yok, eğer kullanıcı adınız ve parolanız doğru değilse, hesabınıza erişemezsiniz. Yani e-posta hesabınıza erişmeniz, kullanıcı adı ve parolayı doğru girme koşuluna bağlıdır.

Ya da şu örneği düşünelim: Diyelim ki Pardus'ta konsol ekranından güncelleme işlemi yapacaksınız. `sudo pisi up` komutunu verdiğiniz zaman güncellemelerin listesi size bildirilecek, bu güncellemeleri yapmak isteyip istemediğiniz size sorulacaktır. Eğer evet cevabı verirsiniz güncelleme işlemi başlayacaktır. Yok, eğer hayır cevabı verirsiniz güncelleme işlemi başlamayacaktır. Yani güncelleme işleminin başlaması kullanıcının evet cevabı vermesi koşuluna bağlıdır. Biz de şimdi Python'da bu tip koşullu durumların nasıl oluşturulacağını öğreneceğiz. Bu iş için kullanacağımız üç tane deyim var: *if*, *elif* ve *else*

Bu konu içinde ayrıca Python'da girintilerin önemine ve yazdığımız kodların içine nasıl açıklama yerleştirebileceğimize de değineceğiz.

## 5.1 if Deyimi

*If* kelimesi İngilizce'de "eğer" anlamına geliyor. Dolayısıyla, adından da anlaşılabilceği gibi, bu deyim yardımıyla Python'da koşula bağlı bir durumu belirtebiliyoruz. Cümle yapısını anlayabilmek için bir örnek verelim:

```
>>> if a == b:
```

Bunun anlamı şudur: *"eğer a ile b birbirine eşit ise..."*

Biraz daha açarak söylemek gerekirse: *"eğer a değişkeninin değeri b değişkeninin değeriyle aynı ise..."*

**Uyarı:** kod içindeki `"=="` (çift eşittir) işaretini birbirine bitişik olarak yazmamız gerekir. Yani bu "çift eşittir" işaretini ayrı yazmamaya özen göstermeliyiz. Aksi halde yazdığımız program hata verecektir.

Gördüğünüz gibi cümlemiz şu anda yarım. Yani belli ki bunun bir de devamı olması gerekiyor. Mesela cümlemizi şöyle tamamlayabiliriz:

```
>>> if a == b:
...     print "a ile b birbirine eşittir"
```

Yukarıda yazdığımız kod şu anlama geliyor: *"Eğer a değişkeninin değeri b değişkeninin değeriyle aynı ise, ekrana 'a ile b birbirine eşittir,' diye bir cümle yazdır."*

Cümlemiz artık tamamlanmış da olsa, tabii ki programımız hâlâ eksik. Bir defa, henüz elimizde tanımlanmış birer a ve b değişkeni yok. Zaten bu kodları bu haliyle çalıştırmaya kalkışırsanız Python size, a ve b değişkenlerinin tanımlanmadığını söyleyen bir hata mesajı gösterecektir.

Biraz sonra bu yarım yamalak kodu eksiksiz bir hale nasıl getireceğimizi göreceğiz. Ama şimdi burada bir parantez açalım ve Python'da girintileme işleminden bahsedelim kısaca. Çünkü bundan sonra girintilerle bol bol haşır neşir olacaksınız.

Dikkat ettiyseniz yukarıda yazdığımız yarım kod içinde *print* ile başlayan ifade, *if* ile başlayan ifadeye göre daha içeride. Bu durum, *print* ile başlayan ifadenin, *if* ile başlayan ifadeye ait bir alt-ifade olduğunu gösteriyor. Yani bu iki satır beraberce bir "if bloğu" oluşturuyor. Eğer metin düzenleyici olarak Kwrite kullanıyorsanız, *if a == b:* yazıp ENTER tuşuna bastıktan sonra Kwrite sizin için bu girintileme işlemini kendiliğinden yapacak, imleci *print "a ile b birbirine eşittir"* komutunu yazmanız gereken yere getirecektir. Ama eğer bu girintileme işlemini elle yapmanız gerekirse genel kural olarak klavyedeki TAB tuşuna bir kez veya SPACE tuşuna dört kez basmalısınız.

Ancak bu kuralı uygularken TAB veya SPACE tuşlarına basma seçeneklerinden yalnızca birini uygulayın. Yani bir yerde TAB tuşuna başka yerde SPACE tuşuna basıp Python'ın kafasının karışmasına yol açmayın.

Şimdi yukarıda verdiğimiz yarım programı tamamlamaya çalışalım. Hemen boş bir Kwrite belgesi açıp içine şunları yazıyoruz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

Bunlar zaten ilk etapta yazmamız gereken kodlardı. Devam ediyoruz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

alinin_yasi = 23
aysenin_yasi = 23
```

Yukarıda *alinin\_yasi* ve *aysenin\_yasi* adında iki tane değişken tanımladık. Bu iki değişkenin de değeri 23.

Programımızı yazmaya devam edelim:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

alinin_yasi = 23
aysenin_yasi = 23

if alinin_yasi == aysenin_yasi:
    print "Ali ile Ayşe aynı yaştadır!"
```

Bu şekilde programımızı tamamlamış olduk. Bu programın pek önemli bir iş yaptığı söylenemez. Yaptığı tek şey, *alinin\_yasi* ile *aysenin\_yasi* değişkenlerinin değerine bakıp, eğer

bunlar birbirleriyle aynıysa ekrana *Ali ile Ayşe aynı yaşıdır!* diye bir çıktı vermektir. Ama bu program ahım şahım bir şey olmasa da, en azından bize *if* deyiminin nasıl kullanılacağı hakkında önemli bir fikir verdi. Artık bilgilerimizi bu programın bize sağladığı temel üzerine inşa etmeye devam edebiliriz. Her zamanki gibi boş bir Kwrite (veya Gedit ya da IDLE) belgesi açıyoruz ve içine şunları yazıyoruz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

parola = "python"
cevap = raw_input("Lütfen parolanızı giriniz: ")

if cevap == parola:
    print "Parola onaylandı! Programa hoş geldiniz!"
```

Gördüğünüz gibi, burada öncelikle parola adlı bir değişken oluşturduk. (Bu arada değişkenlere ad verirken Türkçe karakter kullanmamalısınız.) Bu parola adlı değişkenin değeri, kullanıcının girmesi gereken parolanın kendisi oluyor. Ardından cevap adlı başka bir değişken daha tanımlayıp `raw_input()` fonksiyonunu bu değişkene atadık. Daha sonra da *if* deyimi yardımıyla, *“Eğer cevap değişkeninin değeri parola değişkeninin değeriyle aynı ise ekrana ‘Parola onaylandı! Programa hoş geldiniz!’ yazdır,”* dedik. Bu programı çalıştırdığımızda, eğer kullanıcının girdiği kelime “python” ise parola onaylanacaktır. Yok, eğer kullanıcı başka bir kelime yazarsa, program derhal kapanacaktır. Aynı programı şu şekilde kısaltarak da yazabiliriz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

parola = raw_input("Lütfen parolanızı giriniz: ")

if parola == "python":
    print "Parola onaylandı! Programa hoş geldiniz!"
```

Burada `raw_input()` fonksiyonunun değerini doğrudan parola adlı değişkene atıyoruz. Hemen alttaki satırda ise girilmesi gereken parolanın ne olduğunu şu şekilde ifade ediyoruz: *“Eğer parola python ise ekrana ‘Parola onaylandı! Programa hoş geldiniz!’ yazdır.”*

## 5.2 elif Deyimi

Bazen kullanıcıdan bir veri alırken, kullanıcının verebileceği farklı cevaplara göre programımızın farklı tepkiler vermesini isteyebiliriz. Mesela şu örneğe bakın:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

meyve = raw_input("Bir meyve adı yazınız: ")

if meyve == "elma":
    print "elma bir meyvedir"

elif meyve == "armut":
    print "armut bir meyvedir"

elif meyve == "kiraz":
    print "kiraz bir meyvedir"
```

```
elif meyve == "kabak":  
    print "kabak bir meyve değildir"
```

Burada şu Türkçe ifadeyi Python'caya çevirdik: *"Kullanıcıya, bir meyve ismi yazmasını söyle. Eğer kullanıcının yazdığı isim elma ise, ekrana 'elma bir meyvedir' çıktısı verilsin. Yok, eğer kullanıcının yazdığı isim elma değil, ama armut ise ekrana 'armut bir meyvedir' çıktısı verilsin. Yok, eğer kullanıcının yazdığı isim elma veya armut değil, ama kiraz ise ekrana 'kiraz bir meyvedir' çıktısı verilsin. Yok, eğer kullanıcının yazdığı isim elma, armut veya kiraz değil, ama kabak ise ekrana 'kabak bir meyve değildir' çıktısı verilsin."*

Gördüğünüz gibi, eğer birden fazla koşullu durumla karşı karşıyaysak *elif* deyimini kullanıyoruz. Burada en başa bir adet *if* deyimini koyduğumuza, bundan sonra da üç tane *elif* deyimini kullandığımıza dikkat edin.

Elbette bu tür durumlar için birden fazla *if* deyimini art arda da kullanabiliriz:

```
#!/usr/bin/env python  
# -*- coding: utf-8 -*-  
  
meyve = raw_input("Bir meyve adı yazınız: ")  
  
if meyve == "elma":  
    print "elma bir meyvedir"  
  
if meyve == "armut":  
    print "armut bir meyvedir"  
  
if meyve == "kiraz":  
    print "kiraz bir meyvedir"  
  
if meyve == "kabak":  
    print "kabak bir meyve değildir"
```

Bu örneklerde pek belli olmuyor, ama aslında *if* ile *elif* arasında çok önemli bir fark vardır. Mesela şu örneğe bakalım:

```
#!/usr/bin/env python  
# -*- coding: utf-8 -*-  
  
sayi = 100  
  
if sayi == 100:  
    print "sayi 100'dür"  
  
if sayi <= 150:  
    print "sayi 150'den küçüktür"  
  
if sayi > 50:  
    print "sayi 50'den büyüktür"  
  
if sayi <= 100:  
    print "sayi 100'den küçüktür veya 100'e eşittir"
```

Bu program çalıştırıldığında bütün olası sonuçlar listelenecektir. Yani çıktımız şöyle olacaktır:

```
sayi 100'dür  
sayi 150'den küçüktür  
sayi 50'den büyüktür  
sayi 100'den küçüktür veya 100'e eşittir
```



Eğer bu programı *elif* deyimlerinden de yararlanarak yazarsak sonuç şu olacaktır:

Öncelikle kodumuzu görelim:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

sayi = 100

if sayi == 100:
    print "sayi 100'dür"

elif sayi <= 150:
    print "sayi 150'den küçüktür"

elif sayi > 50:
    print "sayi 50'den büyüktür"

elif sayi <= 100:
    print "sayi 100'den küçüktür veya 100'e eşittir"
```

Bu kodların çıktısı ise şöyle olacaktır:

```
sayı 100'dür
```

Gördüğünüz gibi programımızı *elif* deyimini kullanarak yazarsak Python belirtilen koşulu karşılayan ilk sonucu ekrana yazdıracak ve orada duracaktır. *if* deyimini ise olası bütün sonuçları ekrana basacaktır.

Buraya kadar Python'da pek çok şey öğrenmiş olduk. Şimdiye dek öğrendiklerimizi kullanarak artık çok basit bir hesap makinesi yazabiliriz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from __future__ import division

secenek1 = "(1) toplama"
secenek2 = "(2) çıkarma"
secenek3 = "(3) çarpma"
secenek4 = "(4) bölme"

print secenek1
print secenek2
print secenek3
print secenek4

soru = raw_input("Yapılacak işlemin numarasını girin: ")

if soru == "1":
    sayi1 = input("Toplama için ilk sayıyı girin: ")
    print sayi1
    sayi2 = input("Toplama için ikinci sayıyı girin: ")
    print sayi1, "+", sayi2, "=", sayi1 + sayi2

if soru == "2":
    sayi3 = input("Çıkarma için ilk sayıyı girin: ")
    print sayi3
```

```

    sayi4 = input("Çıkarma için ikinci sayıyı girin: ")
    print sayi3, "-", sayi4, ":", sayi3 - sayi4

if soru == "3":
    sayi5 = input("Çarpma için ilk sayıyı girin: ")
    print sayi5
    sayi6 = input("Çarpma için ikinci sayıyı girin: ")
    print sayi5, "x", sayi6, ":", sayi5 * sayi6

if soru == "4":
    sayi7 = input("Bölme için ilk sayıyı girin: ")
    print sayi7
    sayi8 = input("Bölme için ikinci sayıyı girin: ")
    print sayi7, "/", sayi8, ":", sayi7 / sayi8

```

Bu örnek programı inceleyip, programın nasıl çalıştığını anlamaya uğraşır, eksik yanlarını tespit eder ve bu eksikleri giderme yolları üzerinde kafa yorarsanız, verdiğimiz bu basit örnek amacına ulaşmış demektir.

## 5.3 else Deyimi

*else* deyimi kısaca, *if* ve *elif* deyimleriyle tanımlanan koşullu durumlar dışında kalan bütün durumları göstermek için kullanılır. Küçük bir örnek verelim:

```

#!/usr/bin/env python
# -*- coding: utf-8 -*-

isim = raw_input("Senin ismin ne?")

if isim == "Ferhat":
    print "Ne güzel bir isim bu!"

elif isim == "Serhat":
    print "Hmm.. Fena isim değil..."

else:
    print isim, "adını pek sevmem!"

```

Burada yaptığımız şey şu: Öncelikle kullanıcıya, “*Senin ismin ne?*” diye soruyoruz. Bu soruyu, *isim* adı verdiğimiz bir değişkene atadık. Daha sonra şu cümleyi Pythoncaya çevirdik: “*Eğer isim değişkeninin değeri Ferhat ise, ekrana ‘Ne güzel bir isim bu!’ cümlesini yazdır. Yok, eğer isim değişkeninin değeri Ferhat değil, ama Serhat ise ekrana ‘Hmm.. Fena isim değil...’ cümlesini yazdır. Eğer isim değişkeninin değeri bunların hiçbirisi değil, ama başka herhangi bir şeyse, ekrana isim değişkeninin değerini ve ‘adını pek sevmem!’ cümlesini yazdır.*”

Bu öğrendiğimiz *else* deyimi sayesinde artık kullanıcı yanlış parola girdiğinde uyarı mesajı gösterebileceğiz:

```

#!/usr/bin/env python
# -*- coding: utf-8 -*-

parola = raw_input("Lütfen parolanızı giriniz: ")

if parola == "python":
    print "Parola onaylandı! Programa hoşgeldiniz!"

```

```
else:
    print "Ne yazık ki, yanlış parola girdiniz!"
```

İsterseniz daha önce gördüğümüz bir örneğe de uygulayalım bu *else* deyimini:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

meyve = raw_input("Bir meyve adı yazınız: ")

if meyve == "elma":
    print "elma bir meyvedir"

elif meyve == "armut":
    print "armut bir meyvedir"

elif meyve == "kiraz":
    print "kiraz bir meyvedir"

elif meyve == "kabak":
    print "kabak bir meyve değildir"

else:
    print "%s dediğiniz şeyin ne olduğunu bilmiyorum!" %meyve
```

Böylece özel olarak tanımladığımız durumlar dışında kalan her şey için kullanıcıya gösterecek bir cümlemiz var artık...

## 5.4 Python'da Girintileme Sistemi

Pek çok programlama dilinde girintileme bir tercih meselesidir. Bu dillerde yazdığınız kodlar girintilenmiş de olsa girintilenmemiş de olsa düzgün bir şekilde çalışacaktır. Mesela aşağıdaki C koduna bakalım:

```
#include <stdio.h>

int main()
{
    int a = 1;
    if (a == 1)
    {
        printf("Elveda Zalim Dünya!\n");
        return 0;
    }
}
```

Eğer istersek yukarıdaki kodları şöyle de yazabiliriz:

```
#include <stdio.h>
int main(){int a = 1;if (a == 1){printf("Elveda Zalim Dünya!\n");return 0;}}
```

Bu kodları daha sonra okuyacak kişilerin nefretini kazanmak dışında, kodları bu şekilde yazmamız herhangi bir soruna neden olmaz. Yani yukarıda gösterilen her iki kod da derleyiciler (*compiler*) tarafından aynı şekilde okunup, başarıyla derlenecektir. Eğer yazdığınız kodların okunaklı olmasını istiyorsanız, yukarıda gösterilen ilk kodlama biçimini tercih etmeniz gerekir. Ama dediğim gibi, ikinci kodlama biçimini kullanmanız da programınızın çalışmasını etkilemez.

Ancak Python'la ilgilenen herkesin çok iyi bildiği gibi, Python programlama dilinde girintileme basit bir üslup meselesi değildir.

Yani yukarıdaki C kodlarının yaptığı işi Python'la yerine getirmek istersek şöyle bir kod yazmamız gerekir:

```
a = 1
if a == 1:
    print "Elveda Zalim Dünya"
```

Bu kodların sahip olduğu girintileme yapısı Python açısından büyük önem taşır. Örneğin yukarıdaki kodları şu şekilde yazamayız:

```
a = 1
if a == 1:
print "Elveda Zalim Dünya"
```

Bu kodlar çalışma sırasında hata verecektir.

Aslında Python'daki girintileme mevzuu bundan biraz daha karışıktır. Yukarıdaki örneklerde görüldüğü gibi girinti verilmesi gereken yerde girinti verilmemesinin hataya yol açması dışında, programın yazılması esnasında bazı yerlerde SPACE tuşuna basılarak, bazı yerlerde ise TAB (Sekme) tuşuna basılarak girinti verilmesi de hatalara yol açabilir. Dolayısıyla yazdığınız programlarda girintileme açısından mutlaka tutarlı olmanız gerekir. Boşluk ve sekme tuşlarını karışık bir şekilde kullanmanız, kimi zaman yazdığınız kodların düzgün çalışmasını engellemese bile, farklı metin düzenleyicilerde farklı kod görünümünün ortaya çıkmasına sebep olabilir. Yani mesela herhangi bir metin düzenleyici kullanarak yazdığınız bir programı başka bir metin düzenleyici ile açtığınızda girintilerin birbirine girdiğini görebilirsiniz.

Girintilemelerin düzgün görünmesini sağlamak ve hatalı çalışan veya hiç çalışmayan programlar ortaya çıkmasına sebep olmamak için, kullandığınız metin düzenleyicide de birtakım ayarlamalar yapmanız gerekir. Bir defa kullandığınız metin düzenleyicinin, mutlaka sekmelerin kaç boşluktan oluşacağını belirleyen bir ayarının olması gerekir. Örneğin Gnome'deki Gedit, KDE'deki Kate ve Kwrite, Windows'taki Notepad++ ve Notepad2 adlı metin düzenleyiciler size böyle bir ayar yapma şansı tanır. Herhangi bir program yazmaya başlamadan önce mutlaka sekme ayarlarını yapmanız veya bu ayarların doğru olup olmadığını kontrol etmeniz gerekir. Mesela Gedit programı üzerinden bir örnek verelim.

Gedit'i ilk açtığınızda *düzen/yeğlenenler/düzenleyici* yolunu takip ederek sekmeler başlığı altındaki ayarları kontrol etmelisiniz. Python programlarımızın girinti yapısının düzgün olabilmesi için orada "sekme genişliği"ni "4" olarak ayarlamamız, "Sekme yerine boşluk ekle" seçeneğinin yanındaki kutucuğu da işaretli hale getirmemiz gerekir. Buna benzer ayarlar bütün iyi metin düzenleyicilerde bulunur. Örneğin Geany adlı metin düzenleyiciyi kullanıyorsanız, *düzenle/tercihler/düzenleyici/girinti* yolunu takip ederek şu ayarlamaları yapabilirsiniz:

genişlik => 4

Tür => Boşluklar

sekme genişliği => 4

---

**Not:** Eğer IDLE üzerinde çalışıyorsanız herhangi bir ayar yapmanıza gerek yok. IDLE'da bu söylediğimiz ayarlar zaten yapılmıştır...

---

Öte yandan, bu işin bir de sizin pek elinizde olmayan bir boyutu vardır. Eğer yazdığınız kodlar birden fazla kişi tarafından düzenleniyorsa, sekme ayarları düzgün yapılmamış bir metin düzenleyici kullanan kişiler kodunuzun girinti yapısını allak bullak edebilir. Bu yüzden, ortak proje geliştiren kişilerin de sekme ayarları konusunda belli bir kuralı benimsemesi ve

bu konuda da tutarlı olması gerekir. İngilizce bilenler için, bu girintileme konusuyla ilgili <http://wiki.python.org/moin/HowToEditPythonCode> adresinde güzel bir makale var. Bu makaleyi okumanızı tavsiye ederim.

## 5.5 Python Kodlarına Yorum Ekleme

Program yazarların genellikle ihmal ettiği, ancak aslında son derece önemle bir konudan söz edeceğiz: Yazdığınız programlara yorum/açıklama eklemek.

Diyelim ki, içerisinde satırlar dolusu kod barındıran bir program yazdık ve bu programımızı başkalarının da kullanabilmesi için internet üzerinden dağıtacağız. Bizim yazdığımız programı kullanacak kişiler, kullanmadan önce kodları incelemek istiyor olabilirler. İşte bizim de, kodlarımızı incelemek isteyen kişilere yardımcı olmak amacıyla, programımızın içine neyin ne işe yaradığını açıklayan bazı notlar eklememiz en azından nezaket gereğidir. Başkalarını bir kenara bırakalım, bu açıklayıcı notlar sizin de işinize yarayabilir. Aylar önce yazmaya başladığınız bir programa aylar sonra geri dönmek istediğinizde, *"Acaba ben burada ne yapmaya çalışmışım?"* demenizi de engelleyebilir bu açıklayıcı notlar.

Peki, programımıza bu açıklayıcı notları nasıl ekleyeceğiz?

Kuralımız şu: Python'da kod içine açıklayıcı notlar eklemek için *"#"* işaretini kullanıyoruz.

Hemen bir örnek verelim:

```
print "deneme 1, 2, 3" #deneme yapıyoruz...
```

Yorumlarımızı Python kodları içinde hemen hemen her yere yerleştirebiliriz. Mesela:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

#Program adı: HTMLYap
#Yazarı      : Orçun Kunek
#Tarih       : 19 Nisan 2010
#Amacı       : HTML belgesi üretmek
#Platform    : GNU/Linux, Windows

#HTML belgesinin <title> imine gelecek karakter dizisi
baslik = "Örnek bir HTML Dosyası"

#HTML belgesinin gövde metni
metin = "Merhaba HTML!"

#Burada standart bir HTML şablonu içine, Python'daki
#%s işaretlerini kullanarak bir başlık ve gövde
#metni gömüyoruz.
print """
<html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>%s</title>
</head>
<body>
<p><b>%s</b></p>
</body>
```

```
</html>
""" %(baslik, metin) #başlık ve metin değişkenleri...
```

Kodlarımız içine yorum eklerken göz önünde bulundurmamız gereken önemli bir nokta da ne çok az ne de haddinden fazla yorum girmektir. Özellikle gereğinden fazla yorumlanmış programları okumak çok zor olabilir.

Yukarıda verdiğimiz örneği dikkatli incellerseniz, program içine eklediğimiz yorumların birbirlerinden farklı amaçlara hizmet ettiğini görürsünüz. Mesela şu kısmın görevi geri kalan yorumlarından farklıdır:

```
#Program adı: HTMLYap
#Yazarı      : Orçun KuneK
#Tarih       : 19 Nisan 2010
#Amacı       : HTML belgesi üretmek
#Platform    : GNU/Linux, Windows
```

Burada kodlarımız hakkında açıklayıcı bir yorum yapmak yerine, programımız hakkında genel bir bilgi veriyoruz. Buradan da anlayabileceğiniz gibi, yorumlar bir program içinde çok farklı amaçlara hizmet edebilir. Hatta yorumlardan süsleme amacıyla da yararlanabilirsiniz:

```
#####
#-----#
#          FALANCA v.1          #
#        Yazan: Keramet Su      #
#        Lisans: GPL v2         #
#-----#
#####
```

Python'daki yorumların başka kullanım alanları da vardır. Örneğin, yazdığımız programa bir özellik eklemeyi düşünüyoruz, ama henüz bu özelliği yeni sürüme eklemek istemiyoruz. O zaman şöyle bir şey yapabiliriz:

```
# -*- coding: utf-8 -*-

sistem = raw_input("Hangi işletim sisteminizi kullanıyorsunuz? ")

if sistem == "Windows":
    print "Merhaba Windows kullanıcısı!"

elif sistem == "GNU/Linux":
    print "Merhaba GNU/Linux kullanıcısı!"

#elif sistem == "MacOsX":
#    print "Merhaba MacOsX kullanıcısı!"

else:
    print "Kullandığınız işletim sistemi tanıdık gelmedi..."
```

Burada `elif sistem == "MacOsX":` kısmını yorum içine alarak şimdilik bu kısmı iptal ediyoruz (İngilizce'de bu "yorum içine alma" işlemine "*comment out*" deniyor). Python yorum içinde bir kod bile yer alsın 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).

## 5.6 Bölüm Soruları

1. Çocuklara hayvanların çıkardığı sesleri öğreten basit bir program yazın.
2. Kullanıcının girdiği şehir veya semt bilgisine göre kira fiyat aralıklarını gösteren bir program yazın.
3. Haftanın 5 günü için farklı bir yemek menüsü oluşturun. Örnek bir menü biçimi şöyle olmalı:

```
Gün      :  
Müşteri Adı :  
  
Çorba çeşidi:  
Ana yemek  :  
Tatlı      :
```

Daha sonra müşterinize hangi güne ait menüyü görmek istediğini sorun ve cevaba göre menüyü gösterin.

4. Bir önceki soruda müşterinin yediği yemekler için ayrıntılı hesap dökümü çıkarın ve müşteriyi bilgilendirin.

---

# Bazı Önemli Ayrıntılar

---

Şu noktaya kadar Python’la ilgili epey bilgi edindik. Artık Python’la az çok yönümüzü tayin etmeye yetecek kadar malzemeye sahibiz. Bu bölüme gelene kadar, kafa karışıklığına yol açmamak için bazı önemli ayrıntıları es geçmek zorunda kaldık. İşte bu bölümde şimdiye kadar işlediğimiz konular içinde değinmediğimiz, ama mutlaka bilmemiz gereken bu önemli ayrıntılardan söz edeceğiz. Bu bölümü tamamladıktan sonra Python’la ilgili en temel konuları epey ayrıntılı bir şekilde öğrenmiş olacaksınız. Bu da size ilerki konuları daha rahat anlayabilmeniz açısından sağlam bir temel kazandırmış olacak. Bu bölümün sonunda, eğer varsa, aklınızdaki şüphelerin ve belirsizliklerin büyük bir kısmı da aydınlığa kavuşmuş olacaktır. Üstelik bu bölümde, önceki bölümlerde öğrendiğimiz konularla ilgili de bolca örnek yapma fırsatı bulacağız. Hem eski bilgilerimizi tazeleyeceğiz, hem de yeni şeyler öğreneceğimiz bu bölüme hakettiği ilgiyi göstermeniz halinde, önümüzdeki konuları çok daha rahat kavradığınızı farkederek, kendinizden emin bir şekilde ilerleyebileceksiniz.

## 6.1 İşleçler (Operators)

Hatırlarsanız sayılar konusunu incelerken aritmetik işlemleri yapabilmek için “+”, “-”, “/” ve “\*” gibi işaretlerden yararlanmıştık. Python’da aritmetik işlemler için kullandığımız bu yardımcı işaretlere işleç (*operator*) adı verilir. Bu işleçler bizim yabancıları olduğumuz bir kavram değildir. Bunları daha önce de bol bol kullandık, şimdiden sonra da bol bol kullanmaya devam edeceğiz. Dilerseniz bilgilerimizi tazelemek için birkaç örnek yapalım:

```
>>> print 456 + 654
1110

>>> print 456 - 654
-198

>>> print 45 / 56 #iki tamsayı birbirine bölünürse
... #sonuç da tamsayı olur.
0

>>> print 45.0 / 56 #sayılardan biri kayan noktalıysa
... #sonuç da kayan noktalıdır.
```



```
0.803571428571
```

```
>>> print 5 / 0 #bir sayının 0'a bölünmesi hataya yol açar.
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
ZeroDivisionError: integer division or modulo by zero
```

```
>>> print 45 * 76
```

```
3420
```

Bu işleçler aritmetik işlemleri yapmamızı sağladığı için bunlara özel olarak “aritmetik işleçler” (*arithmetic operators*) adı verilir. Bunların dışında, öğrenmemiz gereken birkaç aritmetik işleç daha var. İsterseniz onlara bir göz atalım:

### “\*\*” işleci

Önceki derslerimizin birinde şuna benzer bir örnek vermiştik:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

sayi = raw_input("Bir sayı girin. Ben size o \
sayının karesini söyleyeyim:\n")

sayi = int(sayi)

print "girdiğiniz sayı\t\t: %s" %(sayi)
print "girdiğiniz sayının karesi: %s" % (sayi * sayi)
```

Burada bir sayının karesini almak için o sayıyı kendisiyle çarptık. Ama Python’da bir sayının karesini almanın çok daha temiz bir yolu var:

```
>>> print 12 ** 2
```

```
144
```

“\*\*” işleci Python’da bir sayının herhangi bir kuvvetini almamızı sağlar. Yukarıdaki komut  $12^2$  ifadesiyle eşdeğerdir. Eğer bir sayının üçüncü kuvvetini almak isterseniz yukarıdaki komutu şöyle yazmalısınız:

```
>>> print 12 ** 3
```

```
1728
```

Bu da  $12^3$  ifadesine eşdeğerdir.

### % işleci

Diyelim ki şöyle bir bölme işlemi yaptınız:

```
>>> print 13 / 3
```

```
4
```

Gördüğünüz gibi bu bölme işleminin sonucu 4’tür. Peki biz bu bölme işleminden kalan sayıyı bulmak istersek ne yapacağız? İşte bunun için Python’da “%” adlı bir işleç var:

```
>>> print 13 % 3
```

```
1
```

Demek ki 13 sayısı 3'e bölündüğünde kalan 1 imiş...

Bu işleç, bir sayının tek mi yoksa çift mi olduğunu tespit etmeniz gerektiği durumlarda çok işinize yarayacaktır. Eğer bir sayının 2'ye bölümünden kalan sayı 0 ise o sayı çifttir. Aksi halde o sayı tektir:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

sayi = input("Bir sayı girin: ")

if sayi % 2 == 0:
    print "Girdiğiniz sayı çifttir."
else:
    print "Girdiğiniz sayı tektir."
```

Bu işlecin, arayüz tasarımı sırasında dahi işinize yaradığını göreceksiniz. O yüzden *"Bölme işleminden kalan sayıyı bilsem ne olur, bilmesem ne olur,"* diye düşünmeyin sakın...

Python'da işleçler yukarıdakilerle sınırlı değildir. Tıpkı yukarıdakiler gibi, önceki derslerimizden bildiğimiz ve matematikten de aşına olduğumuz birkaç işleç daha var:

==	"eşittir" anlamına geliyor
>	"büyüktür" anlamına geliyor
<	"küçüktür" anlamına geliyor
>=	"büyük eşittir" anlamına geliyor
<=	"küçük eşittir" anlamına geliyor

Sayıları karşılaştırmamızı sağladıkları için bu işleçlere "karşılaştırma işleçleri" (*comparison operators*) adı verilir. Biz bu işleçleri önceki derslerimizde *if*, *elif*, ve *else* deyimleri ile birlikte kullanmıştık. Mesela:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

yas = int(raw_input("Yaşınız kaç? "))

if yas < 0:
    print "Yok canım, daha neler!?"

elif yas <= 30:
    print "Hmm.. Epey gençsin."

elif yas >= 60:
    print "Yaş kemale ermiş!"

elif yas > 30:
    print "Yaşlanıyorsun dostum!"
```

Bunlara ek olarak bu bölümde birkaç yeni işleç daha göreceğiz. Bu yeni işleçlere geçmeden önce isterseniz Python'a yeni başlayanların sıkça yaptığı bir hataya değinelim.

Python öğrenmeye yeni başlayanlar, özellikle ilk zamanlarda, "=" ve "==" işaretlerini birbirine karıştırır. Bu işaretler her ne kadar aynı anlama geliyormuş gibi görünse de aslında

birbirlerinden çok farklıdır. Durumu birkaç örnekle açıklamaya çalışalım:

```
>>> sayi = 3456
```

Burada sayi adlı bir değişken tanımladık. Python'da değişkenleri tanımlarken, yani değişkenlere değer atarken "=" işaretinden yararlanıyoruz. Bu işaretin tek görevi bir değişkene değer atamaktır. Burada sayi adlı değişkene 3456 değerini atıyor.

Bir de şuna bakalım:

```
>>> sayi == 3456
```

```
True
```

Burada yaptığımız şey sayi adlı değişkene değer atamak değildir. Burada biz sayi değişkeninin değerini sorguluyoruz. Yani burada şöyle bir soru soruyoruz Python'a: "*sayi adlı değişkenin değeri 3456 mı?*". Bu soruyu duyan Python bize bir cevap veriyor. Python'un cevabını biraz sonra tartışacağız...

Dediğimiz gibi, yukarıda yaptığımız şey bir değer atamak değil, değer sorgulamaktır. Olmayan bir şeyin değeri sorgulanamaz. Dolayısıyla şöyle bir işlem hata verecektir:

```
>>> meyve == "elma"
```

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
NameError: name 'meyve' is not defined
```

Ama siz önce bir meyve değişkeni oluşturursanız, daha sonra bu değişkenin değerini sorgulayabilirsiniz:

```
>>> meyve = "elma"
```

```
>>> meyve == "elma"
```

```
True
```

Burada aldığımız True çıktısına takılmayın. Az sonra bunun ne demek olduğunu inceleyeceğiz. Gördüğünüz gibi, "=" ve "==" işaretleri birbirlerinden çok farklı anlamlara geliyorlar. O yüzden bunları karıştırmamaya özen gösteriyoruz.

Hatırlarsanız "==" işlecini önceki derslerimizde şöyle bir örnek içinde kullanmıştık:

```
#!/usr/bin/env python  
#-*- coding: utf-8 -*-  
  
alinin_yasi = 23  
aysenin_yasi = 23  
  
if alinin_yasi == aysenin_yasi:  
    print "Ali ile Ayşe aynı yaştadır!"
```

Burada da "==" işlecini kullanarak Ali'nin yaşı ile Ayşe'nin yaşını birbiriyle karşılaştırıyoruz. Yukarıdaki örnekte eğer if alinin\_yasi = aysenin\_yasi gibi bir şey yazarsak Python bize bir hata mesajı gösterecektir. Çünkü daha önce de dediğimiz gibi "=" işareti değer atamaya yarar. Bu örnekte bizim yapmamız gereken şey değer atamaktan ziyade iki değeri karşılaştırmak olmalıdır. Eğer siz burada "==" yerine "=" işlecini kullanacak olursanız Python ne yapmaya çalıştığınızı anlayamayacaktır.

Burada şöyle bir soru akla geliyor: “Biz eşitlik karşılaştırması yapmak için “==” işlecini kullanıyoruz. Peki eşitsizlik karşılaştırması için herhangi bir işleç var mı?”

Elbette var. Python’da iki değerin eşitsizliğini sorgulamak için, “eşit değildir,” anlamına gelen “!=” işlecinden yararlanıyoruz. Mesela bu işleci kullanarak yukarıdaki örneği şöyle yazabiliriz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

alinin_yasi = 30
aysenin_yasi = 23

if alinin_yasi != aysenin_yasi:
    print "Ali ile Ayşe aynı yaşta değildir!"
```

Burada şu Türkçe cümleyi Pythonca’ya çevirmiş olduk:

*“Eğer alinin\_yasi adlı değişkenin değeri aysenin\_yasi adlı değişkenin değerine eşit değilse, ekrana ‘Ali ile Ayşe aynı yaşta değildir!’ cümlesini yazdır!”*

Gelin bununla ilgili bir örnek daha verelim konuyu daha net kavramak için:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

parola_ilk = raw_input("Parolanızı giriniz: ")
parola_tekrar = raw_input("Aynı parolayı tekrar giriniz: ")

if parola_ilk != parola_tekrar:
    print "Parolalar eşleşmiyor!"
else:
    print "Programa hoşgeldiniz!"
```

Burada, karşılaştırma işleçlerinden biri olan “!=” işlecini kullanarak parola\_ilk ve parola\_tekrar adlı değişkenlerin değerini karşılaştırıyoruz. Eğer parola\_ilk’in değeri parola\_tekrar’ın değerine eşit değilse *Parolalar eşleşmiyor!* çıktısı veriyoruz. Aksi halde, yani bu iki değer birbirine eşitse kullanıcıyı içeri alıyoruz. Gördüğünüz gibi “!=” işleci oldukça kullanışlı bir araç.

## 6.2 Bool Değerler

İngilizce’de, anlamını bütün programcıların çok iyi bilmesi gereken “true” ve “false” diye iki kelime vardır. “True” kelimesinin anlamı “doğru”, “False” kelimesinin anlamı ise “yanlış”tır. Bilgisayar mantığında her şey ya doğrudur, ya da yanlış... Dolayısıyla herhangi bir şey bilgisayar açısından ya True ya da False değere sahiptir.

Hatırlarsanız yukarıda “==” işlecini incelerken, bu işlecin iki değer arasında karşılaştırma yaptığını söylemiştik. Bunu söylerken de şöyle bir örnek vermiştik:

```
>>> sayi == 3456

True
```

Bu örnek bize True çıktısı veriyor. Dediğimiz gibi, İngilizce’de “true” doğru demektir. Dolayısıyla yukarıdaki çıktıya göre sayi adlı değişkenin değeri gerçekten de 3456.

Bir de şu örneklerle bakalım:

```
>>> a = 3
>>> a == 3
True
>>> a == 4
False
```

Burada öncelikle a adlı bir değişken tanımladık. Bu değişkenin değeri 3. Bir sonraki satırda a değişkeninin değerini sorguluyoruz. Burada sorduğumuz soru şu: “a değişkeninin değeri 3 mü?”. a değişkeninin değeri 3 olduğu için burada True yani “doğru” çıktısını aldık. Bir sonraki satırda ise a == 4 gibi bir ifade görüyoruz. Bu da şu anlama geliyor: “a değişkeninin değeri 4 mü?”. a değişkeninin değeri 4 olmadığı için bu kez False, yani “yanlış” çıktısı alıyoruz.

Dediğimiz gibi, bilgisayar dilinde istisnasız her şey ya True [doğru] ya da False [yanlış]’tur. İşte bu iki değere Bool Değerler adı verilir ve bu sistem bilgisayarın temelini oluşturur.

Şimdi biraz daha ayrıntıya girelim. Yukarıda a değişkeninin değerinin başka bir değere eşit olup olmadığını sorguladık. Yani, “a 3 mü?”, “a 4 mü?” gibi sorular sorduk ve buna uygun cevaplar aldık. Bunun yanı sıra, başlıbaşına a’nın kendisini de doğruluk ve yanlışlık açısından sorgulayabiliriz. Mesela şöyle sorular sorabiliriz: “a doğru mu?” veya “a yanlış mı?”. Python’da böyle bir soru sorabilmek için *bool()* adlı bir fonksiyondan yararlanacağız:

```
>>> a = 3
>>> bool(a)
True
```

Python’da eğer herhangi bir şey, herhangi bir değere sahipse o şey True, yani doğrudur. Ama eğer o şey hiç bir değere sahip değilse, False, yani yanlıştır. Biraz kafa karıştırıcı gelmiş olabilir bu tanım. Örnekler durumu daha net anlatacaktır:

```
>>> a = ""
>>> bool(a)
False
>>> a = "elma"
>>> bool(a)
True
>>> a = " "
>>> bool(a)
True
```

İlk örnekte a boş bir karakter dizisidir. Yani burada a değişkeni hiç bir değere sahip değil. Dolayısıyla bu değişken False’tur. İkinci örnekte ise a değişkenin “elma” diye bir değeri var. Bu yüzden bu değişken True’dur. Üçüncü örnek sizi biraz şaşırtmış olabilir. Ama hiç şaşırtmasın. Çünkü burada a değersiz değil. Bu değişkenin bir değeri var. O da boşluk karakteri! Yani burada a, bir adet boşluk karakteri barındıran bir karakter dizisidir. Unutmayın, boş bir karakter

dizisi ile boşluk karakteri içeren bir karakter dizisi birbiriyle aynı şey değildir. Dilerseniz bu durumu kendi gözlerimizle görelim:

```
>>> a = ""
>>> b = " "
>>> a == b
False
```

Gördüğünüz gibi, a ile b birbirine eşit değil. Çünkü a boş bir karakter dizisi iken, b bir adet boşluk karakterinden oluşan bir karakter dizisidir... Bu ayrımı asla gözden kaçırmayın.

Dilerseniz bununla ilgili küçük bir örnek verelim:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

soru = raw_input("Adınız: ")

if bool(soru) == True:
    print "Teşekkürler"
else:
    print "Bu soruyu boş geçemezsiniz!"
```

Bu örnekte eğer kullanıcı soruya herhangi bir cevap verirse ekrana *Teşekkürler* çıktısı verilir. Ama eğer kullanıcı soruyu cevaplamadan ENTER tuşuna basarsa o zaman *Bu soruyu boş geçemezsiniz* çıktısı alır.

Gördüğünüz gibi burada *bool()* fonksiyonunu kullandık. Buradaki *if bool(soru) == True:* ifadesi şu anlama gelir: *"Eğer soru değişkeninin bool değeri True ise..."*

Yukarıdaki kodları Python'da çok daha kolay yazabiliriz aslında. Zaten Python programcıları yukarıdaki gibi bir şey yazacakları zaman başka bir yol tutmayı tercih ederler. Görelim:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

soru = raw_input("Adınız: ")

if soru:
    print "Teşekkürler"
else:
    print "Bu soruyu boş geçemezsiniz!"
```

Gördüğünüz gibi, hiç *bool()* fonksiyonunu araya sokmadan, sadece *if soru:* diyerek amacımıza ulaşıyoruz. Bu ifade de *"Eğer soru değişkeninin bool değeri True ise..."* anlamına gelir.

Eğer yukarıdakinin tersini yapmak isterseniz *not* işlecinden yararlanabilirsiniz (Bu işleci biraz sonra ayrıntılı olarak inceleyeceğiz):

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

soru = raw_input("Adınız: ")

if not soru:
    print "Bu soruyu boş geçemezsiniz!"
```

```
else:
    print "Teşekkürler"
```

Burada şöyle demiş oluyoruz: “Eğer soru değişkeninin bool değeri True değil ise...”

Bir önceki bölümde şu işlemleri görmüştük:

### Aritmetik İşlemler

+	Toplama işlemi yapar
-	Çıkarma işlemi yapar
/	Bölme işlemi yapar
*	Çarpma işlemi yapar

### Karşılaştırma İşlemleri

==	“eşittir” anlamına geliyor
!=	“eşit değildir” anlamına geliyor
>	“büyüktür” anlamına geliyor
<	“küçüktür” anlamına geliyor
>=	“büyük eşittir” anlamına geliyor
<=	“küçük eşittir” anlamına geliyor

Şimdi de mantık işlemlerini (*logical operators*) inceleyeceğiz. Bir önceki bölümde henüz Bool kavramını öğrenmemiş olduğumuz için mantık işlemlerinden bahsedememiştik. Ama artık bu kavramı bildiğimize göre rahatlıkla mantık işlemlerini işleyebiliriz.

### Mantık İşlemleri

and	“VE” anlamına geliyor
or	“VEYA” anlamına geliyor
not	“DEĞİL” anlamına geliyor

Bu işlemler, lisede mantık dersi almış olanlara hiç yabancı gelmeyecektir.

Peki ne işe yarar bu mantık işlemleri?

İsterseniz uzun uzun açıklama yapmak yerine bir örnek verelim:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

kullanici_adi = raw_input("Kullanıcı adınız: ")
parola = raw_input("Parola: ")

if kullanici_adi == "ahmet" and parola == "ah12345678":
    print "Programa hoşgeldiniz!"
else:
    print "Kullanıcı adınız ya da parolanız yanlış"
```

Burada “and” işleminin ne işe yaradığı apaçık anlaşıyor. Yukarıdaki kodlarda, kullanıcının programa kabul edilmesini, hem kullanıcı adının hem de parolanın doğru girilmesi şartına bağlıyoruz. Eğer kullanıcı adı ve paroladan herhangi biri yanlışsa kullanıcı programa giriş izni elde edemez.

Yukarıdaki kodları bir de şöyle yazalım:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

kullanici_adi = raw_input("Kullanıcı adınız: ")
```

```
parola = raw_input("Parola: ")

if kullanıcı_adi == "ahmet" or parola == "ah12345678":
    print "Programa hoşgeldiniz!"
else:
    print "Kullanıcı adınız ya da parolanız yanlış"
```

Burada “and” yerine “or” kullandığımıza dikkat edin. Bu şekilde kullanıcının programa giriş yapabilmesi için kullanıcı adı veya paroladan birini doğru girmesi yeterli olacaktır.

Dilerseniz etkileşimli kabukta birkaç ufak örnekle durumu daha net anlamaya çalışalım:

```
>>> True and True
```

```
True
```

Dediğimiz gibi, iki True değer birbirine “and” ile bağlanırsa sonuç True olacaktır.

```
>>> True and False
```

```
False
```

Eğer değerlerden biri bile False ise sonuç da False olacaktır.

```
>>> False and False
```

```
False
```

Elbette her iki değer de False olduğu durumda sonuç da False’tur.

```
>>> True or False
```

```
True
```

Eğer iki değer birbirine “or” ile bağlanırsa ve eğer bu değerlerden sadece biri bile True ise sonuç da True olacaktır.

```
>>> False or False
```

```
False
```

“or” ile birbirine bağlanmış iki değer ikisi de False ise sonuç da False’tur.

```
>>> True or True
```

```
True
```

Elbette her iki değer de True ise sonuç da True olacaktır.

Gelelim “not” işlecine... Bu işleç “değil” anlamına gelir. Yani bu işleç cümleye olumsuz bir anlam katar. Birkaç örnek verelim:

```
>>> kullanıcı_adi = "ahmet"
```

```
>>> not kullanıcı_adi
```

```
False
```

Burada Python’a şöyle bir şey söylemiş olduk:

*“kullanıcı\_adi değişkeninin bool değeri True değildir”*



O da bize şu cevabı verdi:

*"Hayır, dostum. Yanılıyorsun! Çünkü kullanıcı\_adi değişkeni bir değere sahip olduğu için True'dur."*

Hatırlarsanız yukarıda şöyle bir örnek vermiştik:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

soru = raw_input("Adınız: ")

if not soru:
    print "Bu soruyu boş geçemezsiniz!"
else:
    print "Teşekkürler"
```

Burada if not soru ifadesine dikkat edin. Burada Python'a şunu diyoruz:

*"Eğer soru değişkeninin bool değeri True değil ise..."*

Yukarıdaki örnekte eğer kullanıcı "Adınız" sorusunu boş geçerse soru değişkeninin değeri şöyle olacaktır:

```
soru = ""
```

Python'da boş karakter dizilerinin False olduğunu biliyoruz. Dolayısıyla eğer kullanıcı soruyu boş geçerse if not soru: satırı işletilecektir...

## 6.3 Dönüştürme İşlemleri

Programcılık maceramız sırasında, verileri birbirine dönüştürmemiz gereken durumlarla sıkça karşılaşacağız. Mesela pek çok durumda bir sayıyı karakter dizisine ve eğer mümkünse bir karakter dizisini de sayıya dönüştürmek zorunda kalacaksınız. Şimdi dilerseniz bu duruma çok basit bir örnek verelim:

Bildiğiniz gibi, Python'da iki tamsayıyı birbirine bölersek elde edeceğimiz çıktı da tamsayı olacaktır. Yani:

```
>>> print 23 / 3
```

```
7
```

Burada iki adet tamsayı olduğu için Python bize sonucu da tamsayı olarak gösterir. Yani ondalık kısmı atar. Bunu önlemek için, işleme giren sayılardan birini kayan noktalı sayı olarak tanımlamamız yeterlidir:

```
>>> print 23.0 / 3
```

```
7.66666666667
```

Peki buna benzer bir şeyi aşağıdaki programda nasıl yaparız?

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

bolen = input("Bölme işlemi için ilk sayıyı girin: ")
bolunen = input("Bölme işlemi için ikinci sayıyı girin: ")
```

```
print "Bölme işleminin sonucu:", bolen / bolunen
```

Biraz önce, sayıları kendimiz girdiğimiz için, bunların sağına birer ".0" işareti koyarak sorunu çözüyorduk. Ama şimdi sayıları kullanıcıdan alıyoruz. Dolayısıyla yukarıdaki programın hassas bir sonuç verebilmesi için burada bir dönüştürme işlemi yapmamız gerekiyor. Bu dönüştürme işlemi için *float()* adlı bir fonksiyondan yararlanacağız:

```
>>> a = 23
>>> float(a)
23.0
```

Gördüğünüz gibi, *float()* fonksiyonu 23 sayısını 23.0 şeklinde bir kayan noktalı sayıya dönüştürdü... Şimdi şuna bakın:

```
>>> a = 23
>>> b = 3
>>> a / b
7
>>> float(a) / b
7.666666666666667
```

Gördüğünüz gibi, bölme işlemine giren sayılardan birini bu *float()* fonksiyonu ile kayan noktalı sayıya döndürerek hassas bir çıktı alabildik. Gelin şimdi bu işlemi yukarıdaki programa uygulayalım:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

bolen = input("Bölme işlemi için ilk sayıyı girin: ")
bolunen = input("Bölme işlemi için ikinci sayıyı girin: ")

print "Bölme işleminin sonucu:", float(bolen) / bolunen
```

Artık bölme işleminin sonucunu hassas bir şekilde alabileceğiz...

Elbette Python'da *float()* gibi başka fonksiyonlar da vardır. Mesela *int()* fonksiyonu bunlardan biridir.

Hatırlarsanız, *input()* ve *raw\_input()* fonksiyonlarını incelerken, *input()* fonksiyonuyla yapabileceğimiz her şeyi aslında *raw\_input()* fonksiyonuyla da yapabileceğimizi söylemiştik. İşte bu *int()* fonksiyonu bize bu konuda yardımcı olacak. Dilerseniz önce bu fonksiyonu etkileşimli kabukta biraz inceleyelim:

```
>>> a = "23"
```

Bildiğiniz gibi yukarıdaki a değişkeni bir karakter dizisidir. Şimdi bunu sayıya çevirelim:

```
>>> int(a)
23
```

Böylece “23” karakter dizisini sayıya çevirmiş olduk. Ancak tahmin edebileceğiniz gibi her karakter dizisi sayıya çevrilemez. `int()` fonksiyonu yalnızca sayı değerli karakter dizilerini sayıya dönüştürebilir:

```
>>> kardiz = "elma"

>>> int(kardiz)

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

Gördüğünüz gibi, “elma” karakter dizisi sayı olarak temsil edilemeyeceği için Python bize bir hata mesajı gösteriyor. Ama “23” gibi bir karakter dizisini sayı olmaktan çıkaran tek şey tırnak işaretleri olduğu için, bu karakter dizisi sayı olarak temsil edilebiliyor... Gelin isterseniz bu bilgiyi şu örneğe uygulayalım:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

ilk_sayi = int(raw_input("İlk sayıyı girin: "))
ikinci_sayi = int(raw_input("İkinci sayıyı girin: "))

toplam = ilk_sayi + ikinci_sayi

print "Bu iki sayının toplamı: ", toplam
```

Gördüğünüz gibi, burada yaptığımız şey çok basit. `raw_input()` fonksiyonunu tümünden `int()` fonksiyonu içine aldık:

```
int(raw_input("İlk sayıyı girin: "))
```

Burada özellikle kapanış tırnaklarını eksik yazmamaya özen gösteriyoruz. Python’a yeni başlayanların en sık yaptığı hatalardan biri de açılmış tırnakları kapatmayı unutmaktır.

Böylece `input()` fonksiyonunun prangasından kurtulmuş olduk! Artık `raw_input()` fonksiyonuyla da aritmetik işlemler yapabiliyoruz. Nihayet `input()` gibi tehlikeli bir fonksiyondan yakayı sıyırdığımıza göre bu durumu bir örnekle kutlayabiliriz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

sayi = int(raw_input("Bir sayı girin. Ben size bu sayının "
                    "istediğiniz kuvvetini hesaplayayım: "))

kuvvet = int(raw_input("Şimdi de %s sayısının kaçınıcı kuvvetini "
                    "hesaplamak istediğinizi söyleyin: "
                    "%sayi))

print "%s sayısının %s. kuvveti %s olur." %(sayi, kuvvet,
                                           sayi ** kuvvet)
```

Burada, yazdığımız kodların nasıl işlediğine dikkat etmenin yanı sıra, kodları görünüş açısından nasıl düzenlediğimize ve satırları nasıl böldüğümüze de dikkat edin. Daha önce de dediğim gibi, Python görünüşe de önem veren zarif bir dildir.

Peki yukarıda yaptığımız şeyin tersi mümkün mü? Yani acaba bir sayıyı karakter dizisine çevirebilir miyiz? Bu sorunun yanıtı evettir. Bu işlem için de `str()` adlı fonksiyondan yararlanacağız:

```
>>> a = 23
>>> str(a)
'23'
```

Gelin isterseniz şimdiye kadar gördüğümüz dönüştürme fonksiyonlarını topluca görelim:

float()	Herhangi bir sayıyı veya sayı değerli karakter dizisini kayan noktalı sayıya dönüştürür.
int()	Herhangi bir sayıyı veya sayı değerli karakter dizisini tamsayıya dönüştürür.
str()	Herhangi bir sayıyı karakter dizisine dönüştürür.

Bu fonksiyonları iyi öğrenmelisiniz, çünkü Python maceranızda bunlar bol bol karşınıza çıkacak...

Bu bölümü kapatmadan önce dilerseniz Python'daki küçük ama önemli bir fonksiyondan daha söz edelim: `type()` fonksiyonu.

`type()` fonksiyonu, herhangi bir değerin hangi tipte olduğunu sorgulamamızı sağlar. Örneğin:

```
>>> i = 23
>>> type(i)
<type 'int'>
```

"int", *integer* (tamsayı) kelimesinin kısaltmasıdır. Demek ki `i` değişkeni bir *integer*, yani tamsayı imiş... Bir de şuna bakalım:

```
>>> s = "elma"
>>> type(s)
<type 'str'>
```

"str" de *string* (karakter dizisi) kelimesinin kısaltmasıdır. Dolayısıyla `s` değişkeni bir karakter dizisidir.

```
>>> f = 14.3
>>> type(f)
<type 'float'>
```

"float", *floating point number* (kayan noktalı sayı) kelimesinin kısaltmasıdır. Yani `f` değişkeni bir kayan noktalı sayıdır.

Bu konuyu da bitirdiğimize göre artık bölüm sorularına geçebiliriz.

## 6.4 Bölüm Soruları

1. Aşağıdaki örnekte *if* ve *elif* deyimlerini farklı sıralarda yerleştirdiğinizde nasıl bir sonuç elde edeceğinizi düşünün. Şöyle bir kod yazımı ne tür sorunların ortaya çıkmasına yol açar? Mesela aşağıdaki kodları çalıştıran bir kullanıcı cevap olarak 60 üzeri bir sayı girerse nasıl bir çıktı alır? Çıktının beklediğimiz gibi olmamasının nedeni nedir?

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

yas = int(raw_input("Yaşınız kaç? "))

if yas < 0:
    print "Yok canım, daha neler!?"

elif yas <= 30:
    print "Hmm.. Epey gençsin."

elif yas > 30:
    print "Yaşlanıyorsun dostum!"

elif yas >= 60:
    print "Yaş kemale ermiş!"
```

2. Aşağıdaki işlemlerin sonucu nedir?

```
>>> True or False
>>> True and False
>>> True and (True or False)
>>> True or (False and True)
>>> True or (False and False)
>>> (False or False) and (False or True)
```

3. Geçen bölümde yaptığımız basit hesap makinesinde *input()* fonksiyonunu kullanmıştık. Siz o hesap makinesini *raw\_input()* fonksiyonunu kullanarak yeniden yazın.

# Python'da Döngüler

Hatırlarsanız geçen bölümlerden birinde basit bir hesap makinesi yapmıştık. Ancak dikkat ettiyseniz, o hesap makinesi programında toplama, çıkarma, çarpma veya bölme işlemlerinden birini seçip, daha sonra o seçtiğimiz işlemi bitirdiğimizde program kapanıyor, başka bir işlem yapmak istediğimizde ise programı yeniden başlatmamız gerekiyordu. Aynı şekilde kullanıcı adı ve parola soran bir program yazsak, şu anki bilgilerimizle her defasında programı yeniden başlatmak zorunda kalırız. Yani kullanıcı adı ve parola yanlış girildiğinde bu kullanıcı adı ve parolayı tekrar tekrar soramayız; programı yeniden başlatmamız gerekir. İşte bu bölümde Python'da yazdığımız kodları sürekli hale getirmeyi, tekrar tekrar döndürmeyi öğreneceğiz.

Kodlarımızı sürekli döndürmemizi sağlamada bize yardımcı olacak parçacıklara Python'da döngü (İngilizce: *Loop*) adı veriliyor. Bu bölümde iki tane döngüden bahsedeceğiz: *while* ve *for* döngüleri. Ayrıca bu bölümde döngüler dışında *break* ve *continue* deyimleri ile *range()* ve *len()* fonksiyonlarına da değineceğiz. Böylece ilerleyen bölümlerde işleyeceğimiz daha karmaşık konuları biraz daha kolay anlamamızı sağlayacak temeli edineceğiz. İsterseniz lafı daha fazla uzatmadan yola koyulalım ve ilk olarak *while* döngüsüyle işe başlayalım...

## 7.1 while Döngüsü

Yazdığımız kodları tekrar tekrar döndürmemizi sağlayan, programımıza bir süreklilik katan öğelere döngü adı verilir. *while* döngüsü, yukarıda verilen tanıma tam olarak uyar. Yani yazdığımız bir programdaki kodların tamamı işletilince programın kapanmasına engel olur ve kod dizisinin en başa dönmesini sağlar. Ne demek istediğimizi anlatmanın en iyi yolu bununla ilgili bir örnek vermek olacaktır. O halde şu küçük örnek bir inceleyelim bakalım:

```
#!/usr/bin/ env python
#-*- coding: utf-8 -*-

a = 0
a = a + 1

print a
```

Bu minicik kodun yaptığı iş, birinci satırda *a* değişkeninin değerine bakıp ikinci satırda bu değere 1 eklemek, üçüncü satırda da bu yeni değeri ekrana yazdırmaktır. Dolayısıyla bu kod parçasının vereceği çıktı da 1 olacaktır. Bu çıktıyı verdikten sonra ise program sona erecektir.

Bu arada ufak bir ipucu verelim size. Python yukarıda gördüğünüz `a = a + 1` satırını şu şekilde yazmanıza da izin verir:

```
a += 1
```

Eğer `a = a - 1` gibi bir şey yazmış olsaydık, bunu da şu şekilde kısaltabilirdik:

```
a -= 1
```

`a = a * 2` veya `a = a / 2` gibi ifadeler de şöyle kısaltılabilir:

```
a *= 2
```

ve:

```
a /= 2
```

Bu kısaltmalar son derece önemlidir, çünkü yazdığınız Python programlarının performansını artırıcı etkileri vardır. O yüzden kodlarınızda mümkün olduğunca bu kısaltmaları kullanmaya çalışın.

Şimdi yukarıdaki koda bazı eklemeler yapalım:

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-

a = 0

while a < 100:
    a += 1
    print a
```

Bu kodu çalıştırdığımızda, 1'den 100'e kadar olan sayıların ekrana yazdırıldığını görürüz.

Konuyu anlayabilmek için şimdi de satırları teker teker inceleyelim:

İlk satırda, 0 değerine sahip bir `a` değişkeni tanımladık.

İkinci ve üçüncü satırlarda, "*a değişkeninin değeri 100 sayısından küçük olduğu müddetçe a değişkeninin değerine 1 ekle,*" cümlesini Pythonca'ya çevirdik.

Son satırda ise, bu yeni `a` değerini ekrana yazdırdık.

İşte bu noktada `while` döngüsünün faziletlerini görüyoruz. Bu döngü sayesinde programımız son satıra her gelişinde başa dönüyor. Yani:

- `a` değişkeninin değerini kontrol ediyor,
- `a` değerinin 0 olduğunu görüyor,
- `a` değerinin 100'den küçük olduğunu idrak ediyor,
- `a` değerine 1 ekliyor ( $0 + 1 = 1$ ),
- Bu değeri ekrana yazdırıyor (1),
- Başa dönüp tekrar `a` değişkeninin değerini kontrol ediyor,
- `a` değerinin şu anda 1 olduğunu görüyor,
- `a` değerinin hâlâ 100'den küçük olduğunu anlıyor,
- `a` değerine 1 ekliyor ( $1 + 1 = 2$ ),
- Bu değeri ekrana yazdırıyor (2),

- Bu işlemi 99 sayısına ulaşana dek tekrarlıyor ve en sonunda bu sayıya da 1 ekleyerek vuslata eriyor.

Gördüğünüz gibi, *while* döngüsünü anlamak ve kullanmak hiç de zor değil. İsterseniz alıştırmaya olması bakımından bir örnek daha verelim:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

a = 1

kullanici_adi = "ahmet_hamdi"
parola = "Saatleri_Ayarlama_Enstitüsü"

while a == 1:
    kull_ad = raw_input("Kullanıcı adı: ")
    par = raw_input("Parola: ")

    if kull_ad == kullanici_adi and par == parola:
        print "Programa hoşgeldiniz!"
        a = 2

    elif not kull_ad or not par:
        print "Bu alanları boş bırakamazsınız!"

    else:
        print "Kullanıcı adı veya parolanız hatalı."
```

Burada öncelikle değeri 1 olan a adlı bir değişken tanımladık. Bu değişkenin değeri 1 olduğu müddetçe programımız çalışmaya devam edecek.

Bir sonraki satırda kullanici\_adi ve parola adlı iki değişken daha tanımlıyoruz. Eğer kullanıcı, bu değişkenlerin değerini doğru girerse sisteme giriş yapabilecek.

Daha sonra da *while* döngümüzü tanımlıyoruz. Burada yaptığımız şeye dikkat edin. *while a == 1* ifadesi ile Python'a şunu söylemiş oluyoruz:

*"a değişkeninin değeri 1 olduğu müddetçe aşağıdaki kodları çalıştırmaya devam et!"*

Bu satırın ardından normal bir şekilde *raw\_input()* fonksiyonlarımız yardımıyla kullanıcıdan kullanıcı adı ve parola bilgilerini istiyoruz.

Ardından gelen *if* deyimini nasıl yazdığımıza dikkat edin. Burada şöyle bir şey demiş olduk:

*"Eğer kull\_ad değişkeninin değeri kullanici\_adi değişkeninin değeriyle VE par değişkeninin değeri parola değişkeninin değeriyle aynıysa..."*

Buradaki *and* (VE) işlecinin görevini biliyorsunuz. Bu sayede kullanıcının hem kullanıcı adı hem de parola bilgisini doğru girmesini isteyebiliyoruz.

Bu arada, buradaki *if* bloğunun içinde görünen *a = 2* ifadesi de çok önemli. Bu ifade ile, en başta belirlediğimiz a değişkeninin 1 olan değerini 2'ye getiriyoruz. Böylece a değişkeninin değeri artık 1 olmamış oluyor. Böylece kullanıcımız kullanıcı adı ve parola bilgilerini doğru girdiğinde *while* döngüsü sona eriyor. Eğer burada a değişkeninin değerini 1 dışında bir değere ayarlamazsak, kullanıcı adı ve parola doğru girildiği halde programımız kullanıcı adı ve parola sormaya devam edecektir.

Sonraki satırda gördüğümüz *elif* bloğunda ise kullanıcı adı ve parola alanlarının boş bırakılıp bırakılmadığını denetliyoruz. Eğer kullanıcımız, kullanıcı adı VEYA (or) parola bilgilerinden herhangi birini boş bırakırsa kendisine *Bu alanları boş bırakamazsınız!* uyarısı gösteriyoruz.



`else` bloğunda ise kullanıcı adı ve parolanın yanlış girilmesi halinde kullanıcıya nasıl bir uyarı göstereceğimizi belirliyoruz.

Yukarıdaki kodlar istediğimiz işlemi rahatlıkla yerine getirmemizi sağlar. Bu açıdan yukarıdaki kodlar tamamen doğru ve geçerli bir Python programıdır. Ancak eğer istersek yukarıdaki kodları biraz sadeleştirme ve profesyonel bir görünüm kazandırma imkanına da sahibiz.

Şu kodlara bir bakın:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

kullanici_adi = "ahmet_hamdi"
parola = "Saatleri_Ayarlama_Enstitüsü"

while True:
    kull_ad = raw_input("Kullanıcı adı: ")
    par = raw_input("Parola: ")

    if kull_ad == kullanici_adi and par == parola:
        print "Programa hoşgeldiniz!"
        quit()

    elif not kull_ad or not par:
        print "Bu alanları boş bırakamazsınız!"

    else:
        print "Kullanıcı adı veya parolanız hatalı."
```

Burada önceki kodlara göre bazı farklılıklar dikkatimizi çekiyor. Mesela önceki kodlarda yer alan `a` değişkenini burada artık kullanmıyoruz. Çünkü bu değişkene ihtiyacımız kalmadı. Bunun yerine `while` döngümüzü şu şekilde yazdık:

```
while True:
```

Hatırlarsanız geçen bölümde Bool değerleri incelerken `True` kelimesinin “doğru” anlamına geldiğini söylemiştik. Dolayısıyla burada gördüğümüz `while True` ifadesi şu anlama geliyor:

*“Doğru olduğu müddetçe aşağıdaki kodları 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.

Bunun dışında farklı olarak ilk `if` bloğu içindeki `a = 2` satırını da kaldırdık. Programın en başındaki `a` değişkenini kaldırdığımız için artık bu satıra da ihtiyacımız yok. Onun yerine, `quit()` adlı bir fonksiyon kullanıyoruz. Siz bu fonksiyonu ilk derslerimizden hatırlıyor olmalısınız. Bu fonksiyon Python’un etkileşimli kabuğundan çıkmamızı sağlıyordu...

`while` döngüsünü yeni öğrenenlerin en sık yaptığı hatalardan biri, döngüyü sona erdirecek bir koşul belirtmeyi unutmaktır.

Şu örneğe bir bakalım:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

soru = raw_input("Python mu Ruby mi?")

while soru != "Python":
    print "Yanlış cevap!"
```

Dikkat ederseniz burada da işleçlerimizden birini kullandık. Kullandığımız işleç “eşit değildir” anlamına gelen “!=” işleci. Ancak burada döngüyü sona erdirecek bir koşul belirlemedik...

Bu programı çalıştırdığımızda sorulan soruya *Python* cevabı vermezsek, program biz müdahale edene kadar ekrana *Yanlış cevap!* çıktısını vermeye devam edecek, yani sonsuz bir döngüye girecektir. Çünkü biz Python’a şu komutu vermiş olduk bu kodla:

*“soru değişkeninin cevabı Python olmadığı müddetçe ekrana ‘Yanlış cevap!’ çıktısını vermeye devam et!”*

Eğer bu programı durdurmak istiyorsak CTRL+C tuşlarına basmamız gerekir.

Yukarıdaki kodların işletilişini sınırlamak için şöyle bir şey yazabiliriz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

olcut = 0

soru = raw_input("Python mu Ruby mi?")

while olcut <= 10:
    olcut += 1
    if soru != "Python":
        print "Yanlış cevap!"
```

Bu kodlar sayesinde, eğer kullanıcının verdiği cevap “Python” değilse, ekrana on kez “Yanlış cevap!” çıktısı verilecektir. Bu kodlar içindeki en önemli satır `olcut += 1`’dir. Çünkü bu kod, *while* döngüsünün her başa dönüşünde `olcut` değişkeninin değerini 1 artırarak 10’a yaklaştırıyor. Eğer bu satırı yazmazsak, `olcut` değişkeninin değeri her zaman 0 olmaya devam edecek, bu nedenle de döngümüz sürekli çalışmaya devam edecektir.

Şimdilik *while* döngüsüne ara verip bu konuda incelememiz gereken ikinci döngümüze geçiyoruz.

## 7.2 for Döngüsü

Python’da en sık kullanılan döngü *for* döngüsüdür desek abartmış olmayız. Siz de kendi yazdığınız programlarda bu döngüyü bolca kullanacaksınız.

*for* döngüsünün görevi bir dizi içindeki öğelerin üzerinden tek tek geçmektir. Bu tanım biraz kafa karıştırıcı gelmiş olabilir. O yüzden isterseniz ne demek istediğimizi basit bir örnek üzerinde anlatalım:

```
>>> for harf in "istihza":
...     print harf
...
i
s
t
i
h
z
a
```

“istihza” bir karakter dizisidir. *for* döngüsü, bu dizi içindeki öğelerin üzerinden tek tek geçmemizi sağlıyor. Burada yaptığımız şey şu:

1. Öncelikle "istihza" karakter dizisinin her bir ögesine "harf" adı veriyoruz. (for harf in "istihza")
2. Daha sonra da bu harf değişkenlerini tek tek ekrana yazdırıyoruz. (print harf)

for döngüsü bazı durumlarda while döngüsü ile benzer bir işlevi yerine getirir. Mesela yukarıda while döngüsünü anlatırken yazdığımız şu kodu hatırlıyorsunuz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

a = 0

while a < 100:
    a += 1
    print a
```

Bu kod yardımıyla ekrana 1'den 100'e kadar olan sayıları yazdırabiliyorduk. Aynı işlemi daha basit bir şekilde for döngüsü (ve range() fonksiyonu) yardımıyla da yapabiliriz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

for i in range(1, 100):
    print i
```

Ben burada değişken adı olarak "i" harfini kullandım, siz isterseniz başka bir harf veya kelime de kullanabilirsiniz.

Yukarıdaki Pythonca kod Türkçe'de aşağı yukarı şu anlama gelir: "1- 100 aralığındaki sayıların her birine i adını verdikten sonra ekrana i'nin değerini yazdır!"

Dediğimiz gibi, for döngüsü, bir dizi içinde bulunan bütün öğelerin üzerinden tek tek geçilmesini sağlar. Yukarıdaki örnekte de bu döngü 1-100 aralığındaki sayıların her biri üzerinden tek tek geçiyor ve bu öğelerin her birini sırayla "i" olarak adlandırıyor.

for döngüsünün bu özelliğinden yararlanarak bazı ilginç şeyler yapabilirsiniz. Mesela 1-100 arasındaki çift sayıları listelemek istiyor olabilirsiniz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

for i in range(1, 100):
    if i % 2 == 0:
        print i
```

Burada yaptığımız şey aslında çok basit. Yukarıdaki kodları Türkçe'ye şöyle çevirebiliriz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

1-100 aralığındaki sayılara i dersek:
    bu i'ler içinde 2'ye tam bölünenleri:
        ekrana bas!
```

Aynı şekilde tek sayıları listelemek de mümkün:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

for i in range(1, 100):
```

```
if i % 2 == 1:
    print i
```

“Eğer bir sayı ikiye bölündüğünde kalan 1 ise o sayı tektir,” kuralından yararlanarak 1-100 arası sayıları süzgeçten geçiriyoruz.

Böylelikle Python’da *while* ve *for* döngülerini de öğrenmiş olduk. Bu arada dikkat ettiyseniz, *for* döngüsü için verdiğimiz ilk örnekte döngü içinde yeni bir fonksiyon kullandık. İsterseniz bu vesileyle biraz da hem döngülerde hem koşullu ifadelerde hem de başka yerlerde karşımıza çıkabilecek faydalı fonksiyonlara ve ifadelere değinelim:

## 7.3 range() fonksiyonu

Önceki bölümde bu fonksiyonla ilgili epey örnek yaptığımız için, artık bu fonksiyonun yabancı sayılmayız. Bu fonksiyonun ne işe yaradığını gayet iyi biliyoruz. Bu fonksiyon Python’da sayı aralıklarını belirtmemizi sağlar. Zaten İngilizce’de de bu kelime “aralık” anlamına gelir. Mesela:

```
print range(100)
```

komutu 0 ile 100 arasındaki sayıları yazdırmamızı sağlar. Dediğimiz gibi, bu `range()` fonksiyonunu bir önceki bölümde birkaç örnekte kullanmıştık.

Başka bir örnek daha verelim:

```
print range(100, 200)
```

komutu 100 ile 200 arasındaki sayıları döker.

Bir örnek daha:

```
print range(1, 100, 2)
```

Bu komut ise 1 ile 100 arasındaki sayıları 2’şer 2’şer atlayarak yazdırmamızı sağlar.

Şu ise 100’den 1’e kadar olan sayıları tersten sıralar:

```
print range(100, 1, -1)
```

Hemen *for* döngüsüyle `range()` fonksiyonunun birlikte kullanıldığı bir örnek verip başka bir fonksiyonu anlatmaya başlayalım:

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-

for sarki in range (1, 15):
    print sarki, "mumdur"
```

## 7.4 len() fonksiyonu

Bu fonksiyon, karakter dizilerinin uzunluğunu gösterir. Mesela:

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-
```

```
a = "Afyonkarahisar"
print len(a)
```

Bu kod, Afyonkarahisar karakter dizisi içindeki harflerin sayısını ekrana dökcektir.

Bu fonksiyonu nerelerde kullanabiliriz? Mesela yazdığınız bir programa kullanıcıların giriş yapabilmesi için parola belirlemelerini istiyorsunuz. Seçilecek parolaların uzunluğunu sınırlamak istiyorsanız bu fonksiyondan yararlanabilirsiniz. Hemen örnek bir kod yazalım:

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-

a = raw_input("Lütfen bir parola belirleyin: ")

if len(a) >= 6:
    print "Parola 5 karakteri geçmemeli!"
else:
    print "Parolanız etkinleştirilmiştir."
```

len() fonksiyonunu yalnızca karakter dizileri ile birlikte kullandığımıza dikkat edin. İlerde bu fonksiyonu başka veri tipleri ile birlikte kullanmayı da öğreneceğiz. Ancak henüz o veri tiplerini görmedik. Dolayısıyla şimdilik bu fonksiyonun karakter dizileriyle birlikte kullanılabildiğini, ama sayılarla birlikte kullanılamadığını bilmemiz yeterli olacaktır. Yani şöyle bir örnek bizi hüsrana uğratır:

```
>>> sayi = 123456
>>> len(sayi)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'int' has no len()
```

Burada gördüğümüz hata mesajı bize, tamsayı veri tipinin len() fonksiyonu ile birlikte kullanılamayacağını söylüyor. Ama şöyle yaparsak olur:

```
>>> kardiz = "123456"
>>> len(kardiz)

6
```

123456 sayısını tırnak içine alarak bunu bir karakter dizisi haline getirdiğimiz için len() fonksiyonu görevini yerine getirecektir.

## 7.5 break deyimi

break deyimi döngü içinde bir noktada programı sona erdirmek gerektiği zaman kullanılır. Aşağıdaki örnek break deyiminin ne işe yaradığını açıkça gösteriyor:

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-

kullanici_adi = "kullanici"
parola = "parola"

while True:
```

```
soru1 = raw_input("Kullanıcı adı: ")
soru2 = raw_input("Parola: ")

if soru1 == kullanıcı_adi and soru2 == parola:
    print "Kullanıcı adı ve parolanız onaylandı."
    break

else:
    print "Kullanıcı adınız veya parolanız yanlış."
    print "Lütfen tekrar deneyiniz!"
```

Bu programda break deyimi yardımıyla, kullanıcı adı ve parola doğru girildiğinde parola sorma işleminin durdurulması sağlanıyor. Yukarıdaki kodlar arasında, dikkat ederseniz, daha önce bahsettiğimiz işleçlerden birini daha kullandık. Kullandığımız bu işleç, “ve” anlamına gelen and işleci. Bu işlecin geçtiği satıra tekrar bakalım:

```
if soru1 == kullanıcı_adi and soru2 == parola:
    print "Kullanıcı adı ve parolanız onaylandı."
```

Burada şu Türkçe ifadeyi Python’caya çevirmiş olduk: *“Eğer soru1 değişkeninin değeri kullanıcı\_adi değişkeniyle; soru2 değişkeninin değeri de parola değişkeniyle aynı ise ekrana ‘Kullanıcı adı ve parolanız onaylandı,’ cümlesini yazdır!”*

Burada dikkat edilmesi gereken nokta şu: and işlecinin birbirine bağladığı soru1 ve soru2 değişkenlerinin ancak ikisi birden doğruysa o bahsedilen cümle ekrana yazdırılacaktır. Yani kullanıcı adı ve paroladan biri yanlışsa if deyiminin gerektirdiği koşul yerine gelmemiş olacaktır. Okulda mantık dersi almış olanlar bu and işlecinin yakından tanıyor olmalı. and işlecinin karşıtı or işlecidir. Bu işleç Türkçe’de “veya” anlamına gelir. Buna göre, *“a veya b doğru ise”* dediğiniz zaman, bu a veya b ifadelerinden birinin doğru olması yetecektir. Şayet *“a ve b doğru ise”* dersiniz, burada hem a hem de b doğru olmalıdır.

## 7.6 continue deyimi

Bu deyim ise döngü içinde kendisinden sonra gelen her şeyin es geçilip döngünün en başına dönülmesini sağlar. Çok bilindik bir örnek verelim:

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-

while True:
    s = raw_input("Bir sayı girin: ")

    if s == "iptal":
        break

    if len(s) <= 3:
        continue

    print "En fazla üç haneli bir sayı girin."
```

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 deyiminin etkisiyle:

```
print "En fazla üç haneli bir sayı girin."
```

satırı es geçilecek ve döngünün en başına dönülecektir.

Eğer kullanıcının girdiği sayıdaki hane üçten fazlaysa ekrana *En fazla üç haneli bir sayı girin.* cümlesi yazdırılacaktır.

## 7.7 Bölüm Soruları

1. Daha önce yazdığımız basit hesap makinesini, *while* döngüsü yardımıyla sürekli çalışabilecek hale getirin.
2. Bu bölümde birleşik işleçlere de değindik. Mesela `a += 1` ifadesinde bu birleşik işleçlerden biri olan `+=`'i kullandık. Gördüğünüz gibi, birleşik işleçler; bir adet aritmetik işleç (+) ve bir adet de atama işlecinden (=) oluşuyor. Birleşik işleçlerde aritmetik işleç atama işlecinden önce geliyor. Peki sizce işleçler neden böyle sıralanmış. Yani neden `a += 1` denmemiş?
3. 1-10 arası bütün **çift sayıların** karesini hesaplayan bir program yazın.
4. Kullanıcıyla sayı tahmin oyunu oynayan bir Python programı yazın. Programda sabit bir sayı belirleyin ve kullanıcıdan bu sayıyı tahmin etmesini isteyin. Kullanıcının girdiği sayılara göre, "yukarı" ve "aşağı" gibi ifadelerle kullanıcıyı doğru sayıya yöneltin.
5. Kullanıcıyla sohbet eden bir program yazın. Yazdığınız bu program kullanıcının verdiği cevaplara göre tavır değiştirebilmeli. Örneğin başlangıç olarak kullanıcıya nereli olduğunu sorabilir, vereceği cevaba göre sohbeti ilerletebilirsiniz.
6. Eğer siz bir GNU/Linux kullanıcısıysanız, "Hangi dağıtım benim ihtiyaçlarıma uygun?" sorusuna cevap veren bir program yazın. Bunun için kullanıcıya bazı sorular sorun. Mesela "Bilgisayarınızın RAM miktarı nedir?", "Depolardaki program sayısı sizin için önem taşır mı?" gibi... Aldığınız cevaplara göre, kullanıcıya önerilerde bulunun. Alternatif olarak, kullanıcıdan aldığı cevaplara göre, herhangi başka bir konuda öneride bulunan bir program da yazabilirsiniz.
7. Eğer siz bir Windows kullanıcısıysanız, "Windows 7'ye mi geçmeliyim?" sorusuna cevap veren bir program yazın. Yazdığınız programda kullanıcıya hız, uyumluluk, yenilik gibi konularla ilgili sorular sorun. Aldığınız cevaplara göre kullanıcıyı yönlendirin. Alternatif olarak, kullanıcıdan aldığı cevaplara göre, herhangi başka bir konuda öneride bulunan bir program da yazabilirsiniz.

---

# Hata Yakalama

---

Hatalar programcılık deneyiminizin bir parçasıdır. Programcılık hayatınız boyunca hatalarla sürekli muhatap olacaksınız. Ancak bizim burada kastettiğimiz, programı yazarken sizin yapacağınız hatalar değil. Kastettiğimiz şey, programınızı çalıştıran kullanıcıların sebep olduğu ve programınızın çökmesine yol açan kusurlar.

Dilerseniz ne demek istediğimizi anlatmak için şöyle bir örnek verelim:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

sayi = int(raw_input("Lütfen bir sayı girin: "))

print "Girdiğiniz sayı", sayi
print "Teşekkürler. Hoşçakalın!"
```

Gayet basit bir program bu, değil mi? Bu program görünüşte herhangi bir kusur taşımıyor. Eğer kullanıcımız burada bizim istediğimiz gibi bir sayı girerse bu programda hiçbir sorun çıkmaz. Programımız görevini başarıyla yerine getirir. Peki ama ya kullanıcı sayı yerine harf girerse ne olacak? Mesela kullanıcı yukarıdaki programı çalıştırıp “a” harfine basarsa ne olur?

Böyle bir durumda programımız şöyle bir hata mesajı verir:

```
Traceback (most recent call last):
  File "deneme.py", line 4, in <module>
    sayi = int(raw_input("Lütfen bir sayı girin: "))
ValueError: invalid literal for int() with base 10: 'a'
```

Gördüğünüz gibi, ortaya çıkan hata nedeniyle programın son satırı ekrana basılamadı. Yani programımız hata yüzünden çöktüğü için işlem yarıda kaldı.

Burada böyle bir hata almamızın nedeni, kullanıcıdan sayı beklediğimiz halde kullanıcının harf girmesi. Biz yukarıdaki programda *int()* fonksiyonunu kullanarak kullanıcıdan aldığımız karakter dizilerini sayıya dönüştürmeye çalışıyoruz. Mesela kullanıcı “23” değerini girerse bu karakter dizisi rahatlıkla sayıya dönüştürülebilir. Dilerseniz bunu etkileşimli kabukta test edelim:

```
>>> int("23")

23
```

Gördüğünüz gibi “23” adlı karakter dizisi rahatlıkla sayıya dönüştürülebiliyor. Peki ya şu?



```
>>> int("a")

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

İşte bizim programımızın yapmaya çalıştığı şey de tam olarak budur. Yani programımız kullanıcıdan aldığı "a" harfini sayıya dönüştürmeye çalışıyor ve tabii ki başarısız oluyor.

Bir kod yazarken, yazılan kodları işletecek kullanıcıları her zaman göz önünde bulundurmanız gerektiğini asla unutmayın. Program çalıştırılırken kullanıcıların ne gibi hatalar yapabileceklerini kestirmeye çalışın. Çünkü kullanıcılar her zaman sizin istediğiniz gibi davranmayabilir. Siz programın yazarı olarak, kodlarınızı tanıdığınız için programınızı nasıl kullanmanız gerektiğini biliyorsunuzdur, ama son kullanıcı böyle değildir.

Yukarıdaki kodun, çok uzun bir programın parçası olduğunu düşünürsek, kullanıcının yanlış veri girişi koskoca bir programın çökmesine veya durmasına yol açabilir. Bu tür durumlarda Python gerekli hata mesajını ekrana yazdırarak kullanıcıyı uyaracaktır, ama tabii ki Python'un sunduğu karmaşık hata mesajlarını kullanıcının anlamasını bekleyemeyiz. Böylesi durumlar için Python'da `try... except` ifadeleri kullanılır. İşte biz de bu bölümde bu tür ifadelerin ne zaman ve nasıl kullanılacağını anlamaya çalışacağız.

## 8.1 try... except...

Dediğimiz gibi, Python'da hata yakalamak için `try... except` bloklarından yararlanılır. Gelin isterseniz bunun nasıl kullanılacağına bakalım.

Giriş bölümünde verdiğimiz örneği hatırlıyorsunuz. O örnek, kullanıcının sayı yerine harf girmesi durumunda hata veriyordu. Şimdi hata mesajına tekrar bakalım:

```
Traceback (most recent call last):
  File "deneme.py", line 4, in <module>
    sayi = int(raw_input("Lütfen bir sayı girin: "))
ValueError: invalid literal for int() with base 10: 'a'
```

Burada önemli kısım `ValueError`'dur. Hatayı yakalarken bu ifade işimize yarayacak.

Python'da hata yakalama işlemleri iki adımdan oluşur. Önce ne tür bir hata ortaya çıkabileceği tespit edilir. Daha sonra bu hata meydana geldiğinde ne yapılacağına karar verilir. Yukarıdaki örnekte biz birinci adımı uyguladık. Yani ne tür bir hata ortaya çıkabileceğini tespit ettik. Buna göre, kullanıcı sayı yerine harf girerse `ValueError` denen bir hata meydana geliyormuş... Şimdi de böyle bir hata ortaya çıkarsa ne yapacağımıza karar vermemiz gerekiyor. Mesela öyle bir durumda kullanıcıya, "*Lütfen harf değil, sayı girin!*" gibi bir uyarı mesajı gösterebiliriz. Dilerseniz bu dediklerimizi somutlaştıralım. Kodlarımızın ilk hali şöyleydi:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

sayi = int(raw_input("Lütfen bir sayı girin: "))

print "Girdiğiniz sayı", sayi
print "Teşekkürler. Hoşçakalın!"
```

Bu tarz bir kodlamanın hata vereceğini biliyoruz. Şimdi şuna bakalım:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

try:
    sayi = int(raw_input("Lütfen bir sayı girin: "))
    print "Girdiğiniz sayı", sayi

except ValueError:
    print "Lütfen harf değil, sayı girin!"

print "Teşekkürler. Hoşçakalın!"
```

Burada, hata vereceğini bildiğimiz kodları bir *try...* bloğu içine aldık. Ardından bir *except* bloğu açarak, ne tür bir hata beklediğimizi belirttik. Buna göre, beklediğimiz hata türü *ValueError*. Son olarak da hata durumunda kullanıcıya göstereceğimiz mesajı yazdık. Artık kullanıcılarımız sayı yerine harfe basarsa programımız çökmeyecek, aksine çalışmaya devam edecektir. Dikkat ederseniz `print "Teşekkürler. Hoşçakalın!"` satırı her koşulda ekrana basılıyor. Yani kullanıcı doğru olarak sayı da girse, yanlışlıkla sayı yerine harf de girse programımız yapması gereken işlemleri tamamlayıp yoluna devam edebiliyor.

Konuyu daha iyi anlayabilmek için bir örnek daha verelim. Hatırlarsanız bir sayıyı 0'a bölmenin hata olduğunu söylemiştik. Şimdi şöyle bir program yazdığımızı düşünün:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

ilk = int(raw_input("Bölme işlemi için ilk sayıyı girin: "))
ikinci = int(raw_input("Şimdi de ikinci sayıyı girin: "))

sonuc = float(ilk) / ikinci

print sonuc
```

Eğer burada kullanıcı ikinci sayıya 0 cevabı verirse programımız şöyle bir hata mesajı verip çökecektir:

```
Traceback (most recent call last):
  File "deneme.py", line 7, in <module>
    sonuc = float(ilk) / ikinci
ZeroDivisionError: float division
```

Böyle bir durumda hata alacağımızı bildiğimize göre ilk adım olarak ne tür bir hata mesajı alabileceğimizi tespit ediyoruz. Buna göre alacağımız hatanın türü *ZeroDivisionError*. Şimdi de böyle bir hata durumunda ne yapacağımıza karar vermemiz gerekiyor. İsterseniz yine kullanıcıya bir uyarı mesajı gösterelim.

Kodlarımızı yazıyoruz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

ilk = int(raw_input("Bölme işlemi için ilk sayıyı girin: "))
ikinci = int(raw_input("Şimdi de ikinci sayıyı girin: "))

try:
    sonuc = float(ilk) / ikinci
    print sonuc
```

```
except ZeroDivisionError:
    print "Lütfen sayıyı 0'a bölmeye çalışmayın!"
```

Bir önceki örnekte de yaptığımız gibi burada da hata vereceğini bildiğimiz kodları *try...* bloğu içine aldık. Böyle bir durumda alınacak hatanın *ZeroDivisionError* olduğunu da bildiğimiz için *except* bloğunu da buna göre yazdık. Son olarak da kullanıcıya gösterilecek uyarıyı belirledik. Böylece programımız hata karşısında çökmeden yoluna devam edebildi.

Burada önemli bir problem dikkatinizi çekmiş olmalı. Biz yukarıdaki kodlarda, kullanıcının bir sayıyı 0'a bölmesi ihtimaline karşı *ZeroDivisionError* hatasını yakaladık. Ama ya kullanıcı sayı yerine harf girerse ne olacak? *ZeroDivisionError* ile birlikte *ValueError*'u da yakalamamız gerekiyor... Eğer yukarıdaki kodları çalıştıran bir kullanıcı sayı yerine harf girerse şöyle bir hatayla karşılaşır:

```
Traceback (most recent call last):
  File "deneme.py", line 4, in <module>
    ilk = int(raw_input("Bölme işlemi için ilk sayıyı girin: "))
ValueError: invalid literal for int() with base 10: 'a'
```

Buradan anladığımıza göre hata veren satır şu: `ilk = int(raw_input("Bölme işlemi için ilk sayıyı girin: "))` Dolayısıyla bu satırı da *try...* bloğu içine almamız gerekiyor.

Şu kodları dikkatlice inceleyin:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

try:
    ilk = int(raw_input("Bölme işlemi için ilk sayıyı girin: "))
    ikinci = int(raw_input("Şimdi de ikinci sayıyı girin: "))

    sonuc = float(ilk) / ikinci
    print sonuc

except ZeroDivisionError:
    print "Lütfen sayıyı 0'a bölmeye çalışmayın!"

except ValueError:
    print "Lütfen harf değil, sayı girin!"
```

Gördüğünüz gibi hata vereceğini bildiğimiz kodların hepsini bir *try...* bloğu içine aldık. Ardından da verilecek hataları birer *except* bloğu içinde teker teker yakaladık. Eğer her iki hata için de aynı mesajı göstermek isterseniz şöyle bir şey yazabilirsiniz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

try:
    ilk = int(raw_input("Bölme işlemi için ilk sayıyı girin: "))
    ikinci = int(raw_input("Şimdi de ikinci sayıyı girin: "))

    sonuc = float(ilk) / ikinci
    print sonuc

except (ZeroDivisionError, ValueError):
    print "Girdiğiniz veri hatalı!"
```

Hata türlerini nasıl grupladığımıza dikkat edin. Hataları birbirinden virgülle ayırıp parantez

içine alıyoruz... Böylece programımız her iki hata türü için de aynı uyarıyı gösterecek kullanıcıya.

Bu bölüm, hata yakalama konusuna iyi bir giriş yapmamızı sağladı. İlerde çok daha farklı hata türleriyle karşılaştığınızda bu konuyu çok daha net bir şekilde içinize sindirmiş olacaksınız.

İsterseniz şimdi bu konuyla bağlantılı olduğunu düşündüğümüz, önemli bir deyim olan *pass*'i inceleyelim.

## 8.2 pass Deyimi

*pass* kelimesi İngilizce'de "geçmek" anlamına gelir. Bu deyim Python programlama dilindeki anlamı da buna çok yakındır. Bu deyim Python'da "görmezden gel, hiçbir şey yapma" anlamında kullanacağız. Mesela bir hata ile karşılaşan programınızın hiçbir şey yapmadan yoluna devam etmesini isterseniz bu deyim kullanabilirsiniz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

try:
    ilk = int(raw_input("Bölme işlemi için ilk sayıyı girin: "))
    ikinci = int(raw_input("Şimdi de ikinci sayıyı girin: "))

    sonuc = float(ilk) / ikinci
    print sonuc

except (ZeroDivisionError, ValueError):
    pass
```

Böylece programınız `ZeroDivisionError` veya `ValueError` ile karşılaştığında sessizce 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 görüldüğünden daha faydalı bir araç olduğunu anlayacaksınız.

## 8.3 Bölüm Soruları

1. Eğer yazdığınız bir programda, kullanıcının yapabileceği olası hataları önceden kestirip bunları yakalamazsanız ne gibi sonuçlarla karşılaşsınız?
2. Daha önce yazdığımız basit hesap makinesini, programı kullanan kişilerin yapabileceği hatalara karşı önlem alarak yeniden yazın.
3. Python'da şöyle bir şey yazmak mümkündür:

```
try:
    bir takım işler
except:
    print "bir hata oluştu!"
```

Burada dikkat ederseniz herhangi bir hata türü belirtmedik. Bu tür bir kod yazımında, ortaya çıkan bütün hatalar yakalanacak, böylece kullanıcıya bütün hatalar için tek bir mesaj gösterilebilecektir. İlk bakışta gayet güzel bir özellik gibi görünen bu durumun sizce ne gibi sakıncaları olabilir?

---

# Listeler

---

Bu bölümden itibaren Python'daki önemli veri tiplerini incelemeye başlayacağız. Burada inceleyeceğimiz veri tipleri sırasıyla şunlardır:

1. Liste (*list*)
2. Demet (*tuple*)
3. Sözlük (*dictionary*)
4. Küme (*set*)

Bu veri tipleri arasında özellikle listeler, Python'un en güçlü olduğu alanlardan biri olması bakımından ayrı bir yere ve öneme sahiptir. O yüzden ilk önce listeleri inceleyeceğiz.

Dilerseniz listenin ne demek olduğunu anlatmaya çalışmakla vakit kaybetmek yerine doğrudan konuya girelim. Böylece soyut kavramlarla kafa karıştırmadan ilerlememiz mümkün olabilir. Zaten ilk liste örneğini görür görmez bu veri tipinin nasıl bir şey olduğunu hemen anlayacaksınız.

## 9.1 Liste Oluşturmak

Listeleri kullanabilmek için öncelikle bir liste oluşturmamız gerekiyor. Python'da herhangi bir liste oluşturmak için önce listemize bir ad vermeli, ardından da köşeli parantezler içinde bu listenin öğelerini belirlemeliyiz. Yani liste oluştururken dikkat etmemiz gereken iki temel nokta var: Birincisi tıpkı değişkenlere isim veriyormuşuz gibi listelerimize de isim vereceğiz. Tabii listelerimizi isimlendirirken Türkçe karakterler kullanmayacağız. İkincisi, listemizi oluşturan öğeleri köşeli parantezler içinde yazacağız. Yani Python'da bir liste oluşturmak için şöyle bir yol izleyeceğiz:

```
>>> liste = []
```

Böylece ilk listemizi başarıyla oluşturmuş olduk. Gördüğünüz gibi, bir liste oluşturmak son derece kolay. Yapacağımız tek şey liste için bir ad belirlemek ve liste öğelerini köşeli parantez içine almak. Burada listemizin henüz herhangi bir öğesi yok. Yani listemiz şu anda boş. Gelin isterseniz öğeleri de olan bir liste oluşturmayı deneyelim:

```
>>> liste = ["Hale", "Jale", "Lale", 12, 23]
```

Böylece içinde öge de barındıran, eksiksiz bir liste oluşturmuş olduk. İsterseniz oluşturduğumuz şeyin bir liste olduğunu teyit edelim:

```
>>> type(liste)
<type 'list'>
```

Tahmin edeceğiniz gibi, *list* kelimesi liste anlamına geliyor. Demek ki gerçekten de bir liste oluşturmuşuz...

Daha önce de söylediğimiz gibi, burada dikkat etmemiz gereken nokta, liste öğelerini oluştururken köşeli parantezler kullanıyor olmamız. Çünkü Python, listeleri parantez işaretlerinden ayırt eder.

Gördüğünüz gibi, liste içine öge eklemek de zor değil. Liste içindeki karakter dizilerini her zamanki gibi tırnak içinde belirtmeyi unutmuyoruz. Tabii ki sayıları yazarken bu tırnak işaretlerini kullanmayacağız.

Olması gerektiği şekilde listemizi oluşturduk. Şimdi komut satırında:

```
>>> print liste
```

komutunu verdiğimizde, oluşturduğumuz bu *liste* adlı listenin öğeleri ekrana yazdırılacaktır.

Python'da liste oluşturmanın bir başka yolu da *list()* adlı bir fonksiyondan yararlanmaktır. Şu örneğe bir bakalım:

```
>>> meyve = "elma"
>>> list(meyve)
['e', 'l', 'm', 'a']
```

Gördüğünüz gibi, bu *list()* fonksiyonunu kullanarak karakter dizisi içindeki karakterlerden bir liste oluşturabiliyoruz. Diyelim ki elinizde birkaç tane karakter dizisi var ve siz bunları bir liste haline getirmek istiyorsunuz. Böyle bir durumda bu fonksiyon işinize yarayabilir:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

a = "elma"
b = "armut"
c = "kiraz"

print list((a, b, c))
```

*list()* fonksiyonunda parantez içine sadece tek bir öge yazabiliriz. Eğer birden fazla öge kullanmak istiyorsanız bu öğeleri ayrı bir parantez içinde belirtmeniz gerekir. Böylece Python bu öğeleri tek bir öğeymiş gibi algılayacak ve size istediğiniz çıktıyı verecektir.

Böylece Python'da bir listenin nasıl oluşturulacağını öğrenmiş olduk. Şimdi isterseniz Python listelerini biraz daha ayrıntılı bir şekilde inceleyelim.

## 9.2 Liste Öğelerine Erişmek

Yukarıdaki örneklerden de gördüğümüz gibi Python'daki listeler tıpkı birer değişken gibi tanımlanıyor:

```
>>> liste = []
```

Bir liste içine öğe eklemek istediğimiz zaman, öğeleri birbirinden virgül ile ayırıyoruz:

```
>>> spor = ["futbol", "basketbol", "tenis", "golf"]
```

Peki bir listedeki öğe sayısını nasıl öğrenebiliriz? Elbette elle sayarak değil... Mesela şu listedeki öğe sayısını elle sayarak bulmak bir Python programcısına hiç yakışmaz:

```
>>> dagitimlar = ["Ubuntu", "Debian", "Fedora", "Arch", "Gentoo", "SuSe",  
... "Pardus", "RedHat", "Truva", "Gelecek", "Mint", "Mandriva", "PCLinuxOs",  
... "Sabayon", "Mepis", "Puppy", "Slackware", "CentOS", "Knoppix", "Zenwalk",  
... "Sidux", "Elive", "Nexenta", "GNewSense", "Ututo", "Vector"]
```

Tanımladığımız bir listenin öğe sayısını, bir önceki bölümde öğrendiğimiz *len()* fonksiyonu yardımıyla elde edebiliriz:

```
>>> len(dagitimlar)
```

```
26
```

Demek ki listemiz 26 öğeden oluşuyormuş. Bu arada uzun bir listeyi nasıl böldüğümüze ve alt satıra geçtiğimize dikkat edin.

Bu listeyi ekrana yazdırmak için şöyle bir şey yapmamız gerektiğini de biliyoruz:

```
>>> print dagitimlar
```

```
['Ubuntu', 'Debian', 'Fedora', 'Arch', 'Gentoo', 'SuSe', 'Pardus', 'RedHat',  
'Truva', 'Gelecek', 'Mint', 'Mandriva', 'PCLinuxOs', 'Sabayon', 'Mepis',  
'Puppy', 'Slackware', 'CentOS', 'Knoppix', 'Zenwalk', 'Sidux', 'Elive',  
'Nexenta', 'GNewSense', 'Ututo', 'Vector']
```

Bu şekilde listenin bütün öğelerini ekrana döktük. Peki ya biz bu 26 öğeli listenin bütün öğelerini değil de, mesela sadece birinci öğesini almak istersek ne yapacağız?

Bunun için şöyle bir yöntemimiz var:

```
>>> print dagitimlar[1]
```

```
Debian
```

Gördüğünüz gibi, formülümüz şöyle:

```
>>> liste_adı[öge_sırası]
```

Yalnız burada dikkat ettiyseniz `print dagitimlar[1]` komutu *Debian* çıktısı verdi. Halbuki listenin ilk öğesi *Ubuntu*. Bu durum Python listelerinin çok önemli bir özelliğidir. Python'da liste öğeleri 0'dan başlar. Yani bir listenin ilk öğesinin sırası her zaman 0'dır. Dolayısıyla yukarıdaki listede *Ubuntu* öğesini elde etmek istiyorsak şu komutu yazmalıyız:

```
>>> print dagitimlar[0]
```

```
Ubuntu
```

Bu yöntem bize çok daha ilginç şeyler yapma imkanı da verir. Mesela bu yöntemi kullanarak bir listenin belli bir aralıktaki öğelerini alabiliriz:

```
>>> liste = ["Hale", "Jale", "Lale", "Ahmet", "Mehmet", "Kezban"]
```



```
>>> print liste[0:2]
```

```
['Hale', 'Jale']
```

Gördüğünüz gibi, bu şekilde listenin birinci ve ikinci öğelerini ayrı bir liste olarak alabiliyoruz. Eğer alacağınız ilk öğenin sırası 0 ise yukarıdaki komutu şöyle de yazabilirsiniz:

```
>>> print liste[:2]
```

Eğer iki nokta üst üste işaretinin sol tarafındaki sayıyı yazmazsanız Python oraya 0 yazmışsınız gibi davranacaktır. Elimiz alışsın diye birkaç örnek yapalım bununla ilgili:

```
>>> print liste[2:5]
```

```
['Lale', 'Ahmet', 'Mehmet']
```

```
>>> print liste[2:6]
```

```
['Lale', 'Ahmet', 'Mehmet', 'Kezban']
```

Son örnekte alacağımız son öğe listenin de son öğesi. O yüzden o komutu şöyle de yazabiliriz:

```
>>> print liste[2:]
```

```
['Lale', 'Ahmet', 'Mehmet', 'Kezban']
```

Eğer iki nokta üst üste işaretinin sağındaki sayıyı yazmazsak Python oraya listenin son öğesinin sırasını yazmışsınız gibi davranacaktır.

Eğer 0'dan başlayarak bir listenin bütün öğelerini almak istersek de şöyle bir şey yazabiliriz:

```
>>> print liste[:]
```

```
['Hale', 'Jale', 'Lale', 'Ahmet', 'Mehmet', 'Kezban']
```

Bu yöntem, bir listeyi tamamen kopyalamanın iyi bir yoludur:

```
>>> liste2 = liste[:]
```

```
>>> print liste2
```

```
['Hale', 'Jale', 'Lale', 'Ahmet', 'Mehmet', 'Kezban']
```

```
>>> print liste
```

```
['Hale', 'Jale', 'Lale', 'Ahmet', 'Mehmet', 'Kezban']
```

Gördüğünüz gibi, aynı öğelere sahip iki farklı liste oluşturduk. Yalnız bu iki listenin birbirinden farklı olduğunu unutmayın. Yani bir liste üzerinde yaptığınız değişiklik öteki listeyi etkilemeyecektir.

Listelere yukarıdaki gibi erişirken üçüncü bir sayı daha verebiliriz:

```
>>> liste[0:6:2]
```

```
['Hale', 'Lale', 'Mehmet']
```

Gördüğünüz gibi, üçüncü sayı liste öğelerinin kaçar kaçar atlanarak ekrana basılacağını gösteriyor. Burada 0'dan başlayarak 6. sıraya kadar olan öğeleri ikişer ikişer atlayarak ekrana bastık.

Yukarıdaki komutu şöyle de yazabilirdik:

```
>>> liste[:2]
['Hale', 'Lale', 'Mehmet']
```

Üçüncü sayı eksi değerli de olabilir. O zaman liste öğeleri geriye doğru sıralanacaktır:

```
>>> liste[::-2]
['Kezban', 'Ahmet', 'Jale']
```

Bütün öğeleri tersten sıralamak istersek şöyle bir şey yazabiliriz:

```
>>> liste[::-1]
['Kezban', 'Mehmet', 'Ahmet', 'Lale', 'Jale', 'Hale']
```

Python'da liste oluşturmayı ve bu listenin öğelerine erişmeyi öğrendiğimize göre şimdi listeleri yönetmeyi; yani listeye öğe ekleme, listeden öğe çıkarma gibi işlemleri nasıl yapacağımızı öğrenebiliriz. Bu işi Python'da bazı parçacıklar (ya da daha teknik bir dille söylemek gerekirse “metotlar”...) yardımıyla yapıyoruz. İsterseniz gelin şimdi bu metodların neler olduğuna ve nasıl kullanıldıklarına bakalım.

## 9.3 Liste Metodları

Dediğimiz gibi, Python'daki listeleri yönetebilmek için “metot” denen bazı araçlardan yararlanmamız gerekiyor. Peki nedir bu metod denen şey?

Metotlar, Python'da bir veri tipinin özelliklerini değiştirmemizi veya sorgulamamızı sağlayan oldukça yetenekli parçacıklardır. Listelerle birlikte hangi metodları kullanabileceğimizi öğrenmek için *dir()* adlı bir fonksiyondan yararlanabiliriz. Bu fonksiyonu listeler üzerine şu şekilde uyguluyoruz:

```
>>> liste = []
>>> dir(liste)
```

Dilerseniz boş bir liste tanımlamakla hiç uğraşmadan, listelerin ayırt edici özelliği olan köşeli parantezleri de kullanabilirsiniz:

```
>>> dir([])
```

Hatta listelerin Python'daki adı olan *list* kelimesini dahi kullanabilirsiniz:

```
>>> dir(list)
```

Tercih tamamen size kalmış...

Bu komutlardan herhangi birini verdiğimizde şöyle bir cevap alırız:

```
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
 '__delslice__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattr__', '__getitem__', '__getslice__', '__gt__', '__hash__',
 '__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
 '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__setslice__',
 '__sizeof__', '__str__', '__subclasshook__', 'append', 'count', 'extend',
```

```
'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

İşte bu gördüğümüz çıktıdaki öğeler listelerin metotları oluyor. Burada bizi ilgilendiren metotlar başında ve sonunda “\_\_” işaretini taşımayanlar. Yani şunlar:

```
append  
count  
extend  
index  
insert  
pop  
remove  
reverse  
sort
```

Biz bu bölümde bu metotları işlevlerine göre gruplayarak inceleyeceğiz. İsterseniz hemen yola koyulalım.

### 9.3.1 Listeye Öğe Ekleme

Python’da herhangi bir listeye öğe eklemek için *append()* metodundan yararlanabiliriz. *append* kelimesi Türkçe’de “eklemek” anlamına gelir. Bu metodun yaptığı şey de tam olarak budur zaten.

Öncelikle bir liste oluşturalım:

```
>>> programlama_dilleri = ["Python", "Ruby", "Perl"]
```

Böylece elimizde *programlama\_dilleri* adlı bir liste olmuş oldu. Şimdi bu listeye C++ öğesini ekleyelim:

```
>>> programlama_dilleri.append("C++")
```

Şimdi listeyi ekrana yazdırdığımızda öğeler arasında C++’ı da göreceğiz:

```
>>> print programlama_dilleri  
['Python', 'Ruby', 'Perl', 'C++']
```

Burada *append()* metodunu nasıl kullandığımıza dikkat edin. Formülümüz şöyle:

```
liste_adı.metot()
```

Alıştırma olması için bir örnek daha yapalım:

```
>>> alisveris_listesi = []  
  
>>> alisveris_listesi.append("soğan")  
  
>>> print alisveris_listesi  
['so\x7a7an']
```

Gördüğünüz gibi, *append()* metoduyla boş bir listeye de öğe ekleyebiliyoruz.

Burada dikkatimizi çeken başka bir şey daha var. Listeyi ekrana yazdırdığımızda Türkçe karakter içeren *soğan* öğesi düzgün görünmüyor. Bu durum normal. Python liste çıktılarında Türkçe

karakterleri gösteremez. Ama liste içindeki Türkçe karakterli öğeleri düzgün göstermenin bir yolu var:

```
>>> for oge in alisveris_listesi:
...     print oge
...
soğan
```

*for* döngüsünden yararlanarak liste öğelerini tek tek ekrana dökümleriz. Bu sırada da Türkçe karakterler, olması gerektiği gibi görünecektir. Bir örnek daha yapalım:

```
>>> alisveris_listesi = ["soğan", "sarımsak", "çilek", "üzüm"]

>>> print alisveris_listesi

['so\x7a7an', 'sar\x8dmsak', '\x87ilek', '\x81z\x81m']

>>> for i in alisveris_listesi:
...     print i
...
soğan
sarımsak
çilek
üzüm
```

Gördüğünüz gibi Python'da çareler tükenmiyor.

*append()* metodunu kullanarak bir listeye tek tek öğe ekleyebiliyoruz. Peki aynı metodu kullanarak listeye birden fazla öğe ekleyebilir miyiz? Deneyelim:

```
>>> iller = []

>>> iller.append("Adana", "Mersin")

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: append() takes exactly one argument (2 given)
```

Burada Python bize bir hata mesajı verir, çünkü *append()* metodu yalnızca tek bir öğe kabul eder. Biz burada iki öğe vermiştik...

Demek ki bu metodu kullanarak bir listeye sadece tek bir öğe ekleyebiliyoruz. *append()* metodu ile şöyle bir şey yapabilirsiniz, ancak istediğiniz tam olarak bu olmayabilir:

```
>>> iller.append(["Adana", "Mersin"])
```

Burada *append()* metodu içinde bir liste oluşturduk (["Adana", "Mersin"]) ve doğrudan bu listeyi *iller* adlı listeye ekledik. Şimdi şuna bakalım:

```
>>> print iller

[['Adana', 'Mersin']]
```

Burada iç içe geçmiş iki tane liste görüyoruz. Demek ki yukarıdaki şekilde öğe eklediğimizde Python bunu yine tek bir öğe olarak ekliyor. O yüzden her zaman istediğiniz şey bu olmayabilir. Peki bir listeye birden fazla öğe eklemek istersek elimizde hiç bir imkan yok mu? Elbette var. Mesela *for* döngülerinden yararlanabilirsiniz:

```
>>> lst = ["Adana", "Mersin"]
>>> iller = []
>>> for i in lst:
...     iller.append(i)
>>> print iller

['Adana', 'Mersin']
```

Böylece istediğimizi elde etmiş olduk. Ama Python bize aynı şeyi çok daha temiz bir şekilde yapma imkanı da sağlar. Bunun için başka bir metottan yararlanacağız. O metodun adı *extend*. Yukarıdaki örneği *extend* metodunu kullanarak yazalım:

```
>>> iller = []
>>> iller.extend(["Adana", "Mersin"])
>>> print iller

['Adana', 'Mersin']
```

*extend* kelimesi Türkçe’de “genişletmek, uzatmak” gibi anlamlara gelir. Bu metod, anlamına uygun olarak bir listeyi başka bir liste ile genişletir... Yani bir listenin öğelerini başka bir listeye tek tek ekler.

*extend* metodunu kullanarak bir listeye tek bir öğe ekleyemezsiniz. Daha doğrusu aldığınız sonuç beklediğiniz gibi olmayabilir. Mesela:

```
>>> os = ["Windows", "Mac"]
>>> os.extend("GNU/Linux")
>>> print os

['Windows', 'Mac', 'G', 'N', 'U', '/', 'L', 'i', 'n', 'u', 'x']
```

Gördüğümüz gibi, *extend* metodu “GNU/Linux” içindeki her bir karakteri tek tek listeye ekledi. Aynı etkiyi *append()* metodu ile elde etmek istersek yine *for* döngüsünden yararlanabiliriz:

```
>>> os = ["Windows", "Mac"]
>>> for harf in "GNU/Linux":
...     os.append(harf)
>>> print os

['Windows', 'Mac', 'G', 'N', 'U', '/', 'L', 'i', 'n', 'u', 'x']
```

Dikkat ederseniz hem *append()*, hem de *extend()* metotları öğeyi hep listenin en sonuna ekliyor. Ama biz yazdığımız programlarda bazen öğeyi belli bir sıraya yerleştirmek isteyebiliriz. İşte bunun için kullanabileceğimiz başka bir metod daha var Python’da: *insert()* metodu.

Kelime olarak *insert*, Türkçe’de “yerleştirmek, sokmak” gibi anlamlara gelir. Bu metodu kullanarak bir öğeyi bir listenin belli bir sırasına yerleştireceğiz. Örneğin:

```
>>> lst = ["Ahmet", "Mehmet", "Salih"]
>>> lst.insert(0, "Ali")
>>> print lst
['Ali', 'Ahmet', 'Mehmet', 'Salih']
```

Burada *insert()* metodunu nasıl kullandığımıza dikkat edin. Metodun parantezi içindeki ilk sayı, yeni öğenin yerleştirileceği konumu gösteriyor. İkinci öğenin ne olduğu ise belli. Burada biz listenin sıfıncı, yani ilk sırasına *Ali* adlı bir öğe yerleştirdik.

### 9.3.2 Listeden Öğe Silmek

Bir önceki bölümde gördüğümüz metotlar yardımıyla listeye öğe ekleyebiliyoruz. Peki ya listeden öğe silmek istersek ne yapacağız?

Bu işlem için de bazı metotlarımız var. Mesela *remove()* metodu.

```
>>> mevsimler = ["İlkbahar", "Yaz", "Sonbahar", "Kış"]
>>> mevsimler.remove("İlkbahar")
>>> print mevsimler
['Yaz', 'Sonbahar', 'K\x8d\x9f']
```

Gördüğünüz gibi, *remove()* metodunu kullanarak liste içindeki istediğimiz bir öğeyi listeden çıkarabiliyoruz. Burada öğeyi listeden çıkarabilmek için öğenin adını yazmamız gerekti. Peki ya biz öğeyi adına göre değil de listedeki sırasına göre çıkarmak istersek ne yapacağız? Yani mesela listedeki 1. öğeyi çıkarmak istiyoruz diyelim...

*del* adlı özel bir deyimini kullanarak bu amacımızı gerçekleştirebiliriz:

```
>>> del mevsimler[0]
```

Bu komut, mevsimler adlı listenin ilk öğesini siler. Aynı deyimle listenin tamamını da silebilirsiniz:

```
>>> del mevsimler
```

Böylece mevsimler adlı listeyi tamamen silmiş olduk:

```
>>> print mevsimler
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'mevsimler' is not defined
```

*remove()* metodu ve *del* deyimini dışında, listelerden öğe silmenin bir başka yolu daha vardır: *pop()* metodu.

```
>>> gunler = ["Pazartesi", "Salı", "Çarşamba", "Perşembe",
... "Cuma", "Cumartesi", "Pazar"]
>>> gunler.pop()
'Pazar'
```

`pop()` metodunun öteki metotlardan farkı, sildiği öğeyi bir de ekrana basmasıdır. Bu metot bir listedeki en son öğeyi siler ve ekrana basar. İsterseniz bu metodun kaçınıcı sıradaki öğeyi sileceğini belirleyebilirsiniz:

```
>>> gunler.pop(0)
```

```
'Pazartesi'
```

Bu komut listedeki 0. öğeyi silip ekrana basacaktır.

### 9.3.3 Liste Öğelerini Sıralamak

Python'da liste öğelerini iki şekilde sıralayabiliriz: Birincisi, liste öğelerini ters çevirebiliriz. İkincisi, liste öğelerini alfabe sırasına dizebiliriz.

Liste öğelerini ters çevirmek için `reverse()` adlı bir metottan yararlanacağız:

```
>>> sayilar = range(10)
```

```
>>> print sayilar
```

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

```
>>> sayilar.reverse()
```

```
>>> print sayilar
```

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

Liste öğelerini alfabe sırasına dizmek için ise `sort()` adlı bir metottan yararlanacağız:

```
>>> isimler = ["Ahmet", "Mehmet", "Selami", "Doruk", "Ege"]
```

```
>>> isimler.sort()
```

```
>>> print isimler
```

```
['Ahmet', 'Doruk', 'Ege', 'Mehmet', 'Selami']
```

Gördüğünüz gibi liste içindeki isimler alfabe sırasına dizildi. Yalnız bu metot Türkçe karakter içeren kelimelerde sorun yaratabilir:

```
>>> isimler = ["Kürşat", "Ökkeş", "Çetin", "Fırat"]
```

```
>>> isimler.sort()
```

```
>>> print isimler
```

```
['F\x8drat', 'K\x81r\x9fat', '\x80etin', '\x99kke\x9f']
```

İsimlerde Türkçe karakter olduğu için alfabe sırasına dizme işlemi başarısız oldu.

Bu sorunun üstesinden gelmek için şöyle bir kod yazabilirsiniz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

```
import locale
```

```
locale.setlocale(locale.LC_ALL, "")

isimler = ["Kürşat", "Ökkeş", "Çetin", "Fırat"]

isimler.sort(key = locale.strxfrm)

for i in isimler:
    print i
```

Bu kodlardaki pek çok şeyi henüz öğrenmedik. Benim amacım sadece *sort()* metoduyla Türkçe karakter içeren kelimelerin de alfabe sırasına dizilebileceğini göstermek. Yukarıdaki kodlarda gördüğümüz her şeyi birkaç bölüm sonra öğrenmiş olacağız.

### 9.3.4 Liste Öğelerinin Sırasını Bulmak

Python'da bir liste içindeki öğelerin sırasını bulmak için *index()* adlı bir metottan yararlanıyoruz. Bu metot bir öğenin liste içinde hangi sırada yer aldığını bildirir:

```
>>> lst = ["erik", "çilek", "karpuz", "kavun"]
>>> lst.index("çilek")
1
```

Eğer bir liste içinde aynı öğeden birden fazla sayıda varsa, bu metot yalnızca ilk rastladığı öğeyi bulacaktır. Mesela:

```
>>> meyveler = ["elma", "erik", "elma", "çilek",
... "karpuz", "kavun", "su", "elma"]
>>> meyveler.index("elma")
0
```

Gördüğünüz gibi, *elma* öğesi liste içinde birkaç farklı konumda geçtiği halde *index()* metodu yalnızca 0. konumdaki öğeyi buldu. Ancak bu metodun liste içinde geçen aynı öğelerin tamamını bulmasını sağlamak da mümkün. Peki ama nasıl?

*index()* metodu, parantez içinde ikinci bir değer daha alır. Şu örneğe bakalım:

```
>>> meyveler = ["elma", "erik", "elma", "çilek",
... "karpuz", "kavun", "su", "elma"]
>>> meyveler.index("elma")
0
>>> meyveler.index("elma", 1)
2
>>> meyveler.index("elma", 2)
2
>>> meyveler.index("elma", 3)
```



Burada parantez içine yazdığımız ikinci değer, *index()* metodunun, bir öğeyi liste içinde hangi konumdan itibaren aramaya başlayacağını gösteriyor. Bunun öntanımlı değeri 0'dır. Yani eğer parantez içindeki ikinci değeri belirtmezseniz, Python siz oraya 0 yazmışsınız gibi davranacak ve aramaya 0. sırada başlayacaktır. Eğer bu değer 1 yaparsanız Python listeyi taramaya 1. öğeden başlayacak ve böylece, 0. sırayı atladığı için 2. sırada yer alan *elma* öğesini bulabilecektir. Eğer bu değeri 3 yaparsak, Python bu kez listeyi taramaya 3. sıradan başlayacak ve böylece, 0. ve 2. sıradaki öğeleri atladığı için 7. sıradaki *elma* öğesini bulabilecektir.

Bu bilgiyi kullanarak, bir liste içindeki öğelerin hangi konumlarda yer aldığını gösteren bir program yazabiliriz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

#meyve listesini tanımlayalım...
meyveler = ["elma", "erik", "elma", "çilek",
            "karpuz", "kavun", "su", "elma"]

#Bu sayı index() metodunun parantezi içindeki
#ikinci değer olacak. Bu değeri bir while
#döngüsü içinde birer birer arttıracamız.
sira = 0

#Bu liste, aranan öğenin listede bulunduğu
#konumları tutuyor.
liste = []

#while döngümüzü yazıyoruz.
while sira < len(meyveler):
    #Bu try... except... bloğunun görevini anlamak
    #için "elma" yerine başka bir öğe koyun ve try...
    #except... bloklarını kaldırın.
    try:
        oge = meyveler.index("elma", sira)
    except ValueError:
        pass

    #sıra değişkeninin değerini birer birer
    #arttırıyoruz. Böylece index() metodu listenin
    #her noktasını tarayabiliyor.
    sira += 1

    #Burada amacımız aynı sayının listeye eklenmesini
    #engellemek. Bunu tam olarak ne işe yaradığını
    #anlamak için de "if not oge in liste:" satırını
    #kaldırmayı deneyebilirsiniz.
    if not oge in liste:
        liste.append(oge)

#listedeki değerleri bir karakter dizisi içine
#alıp ekrana basıyoruz.
for nmr in liste:
    print "aranan öğe %s konumunda bulundu!"%nmr
```

### 9.3.5 Liste Öğelerinin Sayısını Bulmak

Python’da bir liste içindeki öğelerin sayısını bulmak için *count()* adlı bir metottan yararlanıyoruz. Bu metod bir öğenin liste içinde kaç kez geçtiğini bildirir:

```
>>> lst = ["erik", "çilek", "karpuz", "erik", "kavun", "karpuz", "çilek"]
>>> lst.count("erik")
2
```

## 9.4 sum() Fonksiyonu

Diyelim ki kullanıcıdan aldığı sayıları birbiriyle toplayan bir program yazmak istiyorsunuz. Bunun için şöyle bir şey yapabilirsiniz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

a = 0

print """
Toplama işlemi için sayı girin:
(Programdan çıkmak için 'q' tuşuna basın)
"""

while True:
    sayi = raw_input("sayı: ")
    if sayi == "q":
        print "hoşçakalın!"
        break

    else:
        a += int(sayi)

print "girdiğiniz sayıların toplamı: ", a
```

Bu programın mantığı şudur: Öncelikle a adlı bir değişken oluşturuyoruz. Bu değişkenin değeri bir sayıdır. Kullanıcının girdiği her sayı bu değişkenin mevcut değeriyle toplandıktan sonra yine bu değişkene atanacak. Böylece kullanıcının girdiği bütün sayıların toplam değeri bu a değişkeninin de değeri olacak. Dolayısıyla kullanıcının girdiği sayıların toplamını bulmak için programın sonunda a değişkenini ekrana yazdırmamız yeterli olacaktır.

Kullanıcının programı sona erdirebilmesi için “q” tuşunu belirliyoruz. Eğer kullanıcı sayı girmek yerine bu tuşa basarsa program sona erecektir. Ayrıca *else* bloğu içinde, kullanıcıdan aldığımız sayıyı *int*’e, yani tamsayıya dönüştürdüğümüze dikkat edin.

Yukarıdakine benzer bir işlemi Python’da özel bir fonksiyon olan *sum()* yardımıyla da gerçekleştirebiliriz. Bu fonksiyon, bir dizi içindeki sayıların birbiriyle kolayca toplanmasını sağlar. Bununla ilgili bir örnek şöyle olabilir:

```
>>> sayilar = [2, 3, 4]
>>> print sum(sayilar)
```

veya:

```
>>> print sum([2, 3, 4])
```

```
9
```

Gördüğünüz gibi, *sum()* fonksiyonu, kendisine verilen bir dizi sayıyı alıp bu sayıları birbiriyle topluyor. Dilerseniz bu bölümün başında verdiğimiz örneği *sum()* fonksiyonunu kullanarak tekrar yazalım:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

sayilar = []

print """
Toplama işlemi için sayı girin:
(Programmandan çıkmak için 'q' tuşuna basın)
"""

while True:
    sayi = raw_input("sayı: ")
    if sayi == "q":
        print "hoşçakalın!"
        break

    else:
        sayilar.append(int(sayi))

print "girdiğiniz sayıların toplamı: ", sum(sayilar)
```

Bu defa öncelikle *sayilar* adlı bir liste oluşturduk. Kullanıcıdan gelen sayıları bu listede tutacağız. Kullanıcıdan gelen sayıları bu listeye eklemek için *append()* metodundan yararlandığımıza dikkat edin. *sayilar* adlı liste içinde biriken sayıları toplamak için de *sum()* fonksiyonundan yararlandık.

## 9.5 enumerate() Fonksiyonu

Hatırlarsanız, *dir()* adlı bir fonksiyon yardımıyla liste metodlarını topluca ekrana yazdırabileceğimizi öğrenmiştik:

```
>>> dir(list)
```

Eğer burada toplam kaç metod olduğunu öğrenmek istersek *len()* fonksiyonu işimizi görüyor:

```
>>> len(dir(list))
```

```
45
```

Peki ya *dir(list)* çıktısında görünen metodları numaralandırmak istersek ne yapacağız?

İşte bunun için *enumerate()* adlı bir fonksiyonumuz var. Bakalım bu fonksiyonu nasıl kullanıyoruz:

```
>>> liste = dir(list)
```

```
>>> for i in enumerate(liste):
```

```
...     print i
...
(0, '__add__')
(1, '__class__')
(2, '__contains__')
(3, '__delattr__')
(4, '__delitem__')
(5, '__delslice__')
(6, '__doc__')
(7, '__eq__')
(8, '__format__')
(9, '__ge__')
```

Biz burada çıktının tamamını göstermiyoruz, ancak gördüğünüz gibi, `dir(list)` komutunun verdiği bütün çıktılar (*sıra\_no*, *metot\_adi*) şeklinde ekrana dökülüyor. Şimdi şuna bakın:

```
>>> for sıra_no, metod_adi in enumerate(liste):
...     print "%s. %s" %(sıra_no, metod_adi)
...
0. __add__
1. __class__
2. __contains__
3. __delattr__
4. __delitem__
5. __delslice__
6. __doc__
7. __eq__
8. __format__
9. __ge__
```

Bu kodlarla çıktıyı nasıl istediğimiz kıvama getirdiğimizi görüyorsunuz. Burada `for sıra_no, metod_adi in enumerate(liste):` satırına özellikle dikkat edin. Bir önceki örnekte `enumerate(liste)` içeriğinin (*sıra\_no*, *'metot\_adi'*) şeklinde olduğunu görmüştük. İşte biz `for sıra_no, metod_adi` satırı ile, çıktıdaki sıra numaralarını *sıra\_no* adlı değişkene, metod adlarını ise *metot\_adi* adlı değişkene atıyoruz. Yukarıdaki kodu şöyle yazacak olursanız durum daha da netleşecektir:

```
>>> for sıra_no, metod_adi in enumerate(liste):
...     print sıra_no
...
0
1
2
3
4
5
6
7
8
9

>>> for sıra_no, metod_adi in enumerate(liste):
...     print metod_adi
...
__add__
__class__
__contains__
__delattr__
```

```
__delitem__
__delslice__
__doc__
__eq__
__format__
__ge__
```

Gördüğünüz gibi, `sira_no` değişkeni sıra numaralarını, `metot_adi` değişkeni ise metod adlarını tutuyor. Yukarıda `print "%s. %s" %(sira_no, metod_adi)` satırıyla yaptığımız şey ise `enumerate()` fonksiyonundan gelen bu sıra numaralarını ve metod adlarını "%s" işaretlerini kullanarak düzgün bir şekilde biçimlendirmekten ibarettir.

Bu arada dikkat ederseniz `enumerate()` ile verdiğimiz numaralar sıfırdan başlıyor. Eğer ilk sayının 0 yerine 1 olmasını isterseniz yukarıdaki kodu şöyle yazabilirsiniz:

```
>>> for sıra_no, metod_adi in enumerate(liste, 1):
...     print "%s. %s" %(sıra_no, metod_adi)
```

Burada 1 yerine istediğiniz sayıyı yerleştirerek, sıralamaya istediğiniz sayıdan başlanmasını sağlayabilirsiniz.

## 9.6 in Deyimi

Aslında biz bu *in* deyimini biliyoruz. Biz bu deyim *for* döngüsüyle birlikte kullanıyorduk. Mesela:

```
>>> for i in range(10):
...     print i
```

*in* deyimini bir ifadeye temel olarak "-de/-da, içinde" anlamı katar. Bu deyim *in* kullanarak bir şeyin başka bir şeyin içinde olup olmadığını sorgulayabiliriz:

```
>>> liste = ["elma", "armut", "çilek", "erik", "muz"]
```

```
>>> "elma" in liste
```

```
True
```

Burada tam olarak şöyle dedik: "*liste içinde 'elma' var mı?*"

*elma* ögesi *liste* içinde bulunduğu için Python bize `True` cevabını verdi. Bir de şuna bakalım:

```
>>> "karpuz" in liste
```

```
False
```

*Liste içinde "karpuz" diye bir öge olmadığı için Python bize False cevabı verdi...*

Gelin isterseniz bununla ilgili şöyle bir örnek verelim:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

turkce_harfler = "ışğçüöışğçüö"
kontrol_listesi = []

dosya = raw_input("Dosya için bir ad seçin: ")
```

```

for hrf in dosya:
    if hrf in turkce_harfler:
        kontrol_listesi.append(hrf)

if kontrol_listesi:
    print "Dosya adında Türkçe harf olmamalı!"

elif len(dosya) > 8:
    print "Dosya adı en fazla 8 karakter içerebilir!"

else:
    print "Dosya '%s' adıyla kaydedildi!" %dosya

```

Burada yaptığımız şey aslında çok basit. Önce Türkçe harflerin tanımlandığı bir liste oluşturuyoruz. Daha sonra kullanıcıdan gelen dosya adına bakıp bunun içinde Türkçe karakter olup olmadığını denetliyoruz. Eğer dosya adında Türkçe karakter varsa bunları kontrol listesine ekliyoruz. Eğer kontrol listesinde en az bir tane öge varsa dosya adı Türkçe karakter içeriyor demektir.

## 9.7 min() ve max() Fonksiyonları

Bir sayı dizisi içinde yer alan sayıların en büyüğünü veya en küçüğünü öğrenmeniz gerekirse ne yaparsınız? Elbette *min()* ve *max()* fonksiyonlarından yararlanırsınız.

```

>>> lst = [45, 24, 98, 71, 43, 94, 35, 90, 66, 39, 97, 96, 66, 26, 34, 61,
... 80, 77, 30, 92, 88, 59, 64, 50, 27, 63, 77, 48, 28, 52, 78, 56, 61, 52,
... 35, 41, 53, 75, 93, 97, 58, 75, 73, 57, 73, 73, 46, 74, 31, 40, 100,
... 94, 73, 40, 25, 64, 96, 27, 33, 77, 62, 97, 25, 76, 33, 42, 99, 33, 54,
... 39, 99, 62, 99, 72, 90, 62, 78, 39, 92, 42, 58, 50, 41, 73, 27, 54, 37,
... 75, 34, 63, 80, 100, 83, 32, 65, 49, 82, 64, 42, 30]

>>> min(lst) #listedeki en küçük sayı

24

>>> max(lst) #listedeki en büyük sayı

100

```

## 9.8 Bölüm Soruları

1. Şu örnekte, kullanıcıdan aldığımız karakter dizisini neden `int(raw_input("sayı: "))` şeklinde en başta değil de *e/se* bloğu içinde sayıya dönüştürmeyi tercih ettiğimizi açıklayın.

```

#!/usr/bin/env python
# -*- coding: utf-8 -*-

a = 0

print """
Toplama işlemi için sayı girin:
(Programdan çıkmak için 'q' tuşuna basın)
"""

```

```
while True:
    sayi = raw_input("sayı: ")
    if sayi == "q":
        print "hoşçakalın!"
        break

    else:
        a += int(sayi)

    print "girdiğiniz sayıların toplamı: ", a
```

2. [http://wiki.ubuntu-tr.net/index.php/Ubuntu'da\\_S%C3%BCr%C3%BCm\\_%C4%B0simleri\\_ve\\_Anlam%C4%ADresinde\\_bulunan\\_Ubuntu\\_s%C3%BCr%C3%BCm\\_isimlerini\\_bir\\_liste\\_haline\\_getirin\\_ve\\_ekrana\\_şuna\\_benzer\\_bir\\_çıkıtı\\_verin:](http://wiki.ubuntu-tr.net/index.php/Ubuntu'da_S%C3%BCr%C3%BCm_%C4%B0simleri_ve_Anlam%C4%ADresinde_bulunan_Ubuntu_s%C3%BCr%C3%BCm_isimlerini_bir_liste_haline_getirin_ve_ekrana_şuna_benzer_bir_çıkıtı_verin:)

```
1) Warty Warthog
2) Hoary Hedgehog
...
```

3. Bir program yazarak kullanıcıdan 10 adet sayı girmesini isteyin. Kullanıcının aynı sayıyı birden fazla girmesine izin vermeyin. Programın sonunda, kullanıcının girdiği sayıları tek ve çift sayılar olarak ikiye ayırın ve her bir sayı grubunu ayrı ayrı ekrana basın.

4. Aşağıdaki programı, liste ögesini kullanıcıdan alacak şekilde yeniden yazın. Eğer kullanıcının aradığı öge listede yoksa kendisine bir uyarı mesajı gösterin ve başka bir öge arayabilme şansı verin:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

meyveler = ["elma", "erik", "elma", "çilek",
            "karpuz", "kavun", "su", "elma"]

sira = 0
liste = []
while sira < len(meyveler):
    try:
        oge = meyveler.index("elma", sira)
    except ValueError:
        pass
    sira += 1
    if not oge in liste:
        liste.append(oge)

for nmr in liste:
    print "aranan öge %s konumunda bulundu!"%nmr
```

5. Liste metotları içinde sadece bizi ilgilendiren şu metotları ekrana döken bir program yazın:

```
append
count
extend
index
insert
pop
remove
reverse
sort
```

---

# Demetler

---

Demetler, bir önceki bölümde incelediğimiz listelere çok benzer. Ama listeler ile aralarında çok temel bir fark vardır. Listeler üzerinde oynamalar yapabiliriz. Yani öğe ekleyebilir, öğe çıkarabiliriz. Demetlerde ise böyle bir şey yoktur.

Demeti şu şekilde tanımlıyoruz:

```
>>> demet = "Ali", "Veli", 49, 50
```

Gördüğünüz gibi, yaptığımız bu iş değişken tanımlamaya çok benziyor. İstersek demetin öğelerini parantez içinde de gösterebiliriz:

```
>>> demet2 = ("Ali", "Veli", 49, 50)
```

Parantezli de olsa parantezsiz de olsa yukarıda tanımladıklarımızın ikisi de “demet” sınıfına giriyor. İsterseniz bu durumu teyit edelim:

```
>>> type(demet)
<type 'tuple'>
>>> type(demet2)
<type 'tuple'>
```

*tuple* kelimesi Türkçe’de demet anlamına gelir...

Peki boş bir demet nasıl oluşturulur? Çok basit:

```
>>> demet = ()
```

Peki tek öğeli bir demet nasıl oluşturulur? O kadar basit değil. Aslında basit ama biraz tuhaf:

```
>>> demet = ("su",)
```

Gördüğünüz gibi, tek öğeli bir demet oluşturabilmek için öğenin yanına bir virgül koyuyoruz! Hemen teyit edelim:

```
>>> type(demet)
<type 'tuple'>
```



O virgülü koymazsak ne olur?

```
>>> demet2 = ("su")
```

demet2'nin tipini kontrol edelim:

```
>>> type(demet2)
<type 'str'>
```

Demek ki, virgülü koymazsak demet değil, alelade bir karakter dizisi oluşturmuş oluyoruz.

Yukarıda anlattığımız şekilde bir demet oluşturma işine demetleme (*packing*) adı veriliyor. Bunun tersini de yapabiliriz. Buna da demet çözme deniyor (*unpacking*).

Önce demetleyelim:

```
>>> aile = "Anne", "Baba", "Kardesler"
```

Şimdi demeti çözelim:

```
>>> a, b, c = aile
```

Bu şekilde komut satırına `print a` yazarsak, *Anne* ögesi; `print b` yazarsak *Baba* ögesi; `print c` yazarsak *Kardesler* ögesi ekrana yazdırılacaktır. Demet çözme işleminde dikkat etmemiz gereken nokta, eşittir işaretinin sol tarafında demetteki öge sayısı kadar değişken adı belirlememiz gerektiridir.

Önceki bölümde gördüğümüz *enumerate()* fonksiyonunu hatırlıyorsunuz. Bu fonksiyonu şöyle bir örnek içinde kullanabileceğimizi de biliyorsunuz:

```
>>> liste = dir(list)
>>> for a, b in enumerate(liste):
...     print a, b
```

Bu da aslında bir demet çözme işlemidir. Bildiğiniz gibi *enumerate()* fonksiyonu bir demet üretir. Bunu hemen teyit edelim:

```
>>> liste = dir(list)
>>> for i in enumerate(liste):
...     print type(i)
...
<type 'tuple'>
```

Gördüğünüz gibi elde ettiğimiz şey bir demet. Dolayısıyla `print a` dediğimiz zaman bu demetin ilk ögesini, `print b` dediğimiz zaman da ikinci ögesini ekrana yazdırmış oluyoruz.

En başta da dediğimiz gibi, demetler listelere çok benzer. "*Peki, listeler varken bu demetler ne işe yarar?*" diye sorduğunuzu duyar gibiyim.

Bir defa, demetler listelerin aksine değişiklik yapmaya müsait olmadıklarından listelere göre daha güvenlidirler. Yani yanlışlıkla değiştirmek istemediğiniz veriler içeren bir liste hazırlamak istiyorsanız demetleri kullanabilirsiniz. Ayrıca demetler listelere göre daha hızlı çalışır. Dolayısıyla bir program içinde sonradan değiştirmeniz gerekmeyecek verileri gruplamak için liste yerine demet kullanmak daha mantıklıdır.

## 10.1 Demetlerin Metotları

Demetler üzerinde değişiklik yapmak mümkün olmadığı için bu veri tipi metot bakımından da bir hayli fakirdir. Şu komutlardan herhangi biri yardımıyla demetlerin metotlarını görebileceğimizi biliyoruz:

```
>>> demet = ()
>>> dir(demet)
>>> dir(tuple)
>>> dir(())
```

Bu komutlar bize şu çıktıyı verir:

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

Burada bizi ilgilendirenler başında ve sonunda “\_” işareti barındırmayanlar. Yani şunlar:

```
count
index
```

### 10.1.1 Demet Öğelerinin Sırasını Bulmak

Tıpkı listelerde olduğu gibi, demetlerde de sıra bulma işlemleri için *index()* metodunu kullanıyoruz. Örneğin:

```
>>> demet = ("gül", "papatya", "kasımpatı", "erguvan")
>>> demet.index("gül")
0
```

Demek ki *gül* adlı öğe demetin 0. sırasında imiş... Yine tıpkı listelerde olduğu gibi bu metot sadece ilk bulduğu öğeyi dikkate alır. Burada da parantez içinde ikinci bir değer kullanabilirsiniz:

```
>>> demet.index("gül", 2)
```

### 10.1.2 Demet Öğelerinin Sayısını Bulmak

Python’da bir demet içindeki öğelerin sayısını bulmak için, tıpkı listelerde olduğu gibi, *count()* adlı bir metottan yararlanıyoruz. Bu metot bir öğenin demet içinde kaç kez geçtiğini bildirir:

```
>>> demet = ("erik", "çilek", "karpuz", "erik", "kavun", "karpuz", "çilek")
>>> demet.count("erik")
2
```

## 10.2 Bölüm Soruları

1. Listelerin aksine demetler üzerinde değişiklik yapılamamasının ne gibi avantajları olabileceği üzerinde düşünün.
2. Demetlerin bütün metotlarını numaralandıran ve her metottaki karakter sayısını gösteren bir program yazın. Programınız şöyle bir çıktı vermeli:

No	Metot Adı	Metot Uzunluğu
0	__add__	7
1	__class__	9
2	__contains__	12
3	__delattr__	11
4	__doc__	7
5	__eq__	6

3. Bir önceki soruda, numaralandırmayı 0 yerine 1'den başlatmak için ne yapmanız gerekir?
4. Diyelim ki elimizde şöyle bir liste var:

```
liste = ["elma", "armut", "elma", "kiraz",  
        "çilek", "kiraz", "elma", "kebab"]
```

Bu listedeki her bir öğenin, listede kaç kez geçtiğini söyleyen bir program yazın. Programınız tam olarak şöyle bir çıktı vermeli:

```
elma öğesi listede 3 kez geçiyor!  
armut öğesi listede 1 kez geçiyor!  
kiraz öğesi listede 2 kez geçiyor!  
çilek öğesi listede 1 kez geçiyor!  
kebab öğesi listede 1 kez geçiyor!
```

# Sözlükler

Şu ana kadar Python’da iki önemli veri tipinden bahsettik. Bunlar listeler ve demetler idi. Şimdi de sözlük adlı veri tipinden söz edeceğiz. Sözlükler Python’un en önemli veri tiplerinden bir tanesidir. Bu veri tipini öğrendikten sonra pek çok şeyi rahatlıkla yapabildiğinizi göreceksiniz, ufkunuzun genişlediğini hissedeceksiniz.

O halde lafı daha fazla uzatmadan yola koyulalım.

## 11.1 Sözlük Oluşturmak

Sözlükleri kullanabilmek için yapmamız gereken ilk iş sözlüğü oluşturmak olacaktır. Python’da sözlükleri şöyle oluşturuyoruz:

```
>>> sozluk = {}
```

Gördüğünüz gibi, sözlüklerin ayırt edici işareti küme parantezleridir. Yukarıda boş bir sözlük oluşturduk. Gelin isterseniz bunu test edelim:

```
>>> type(sozluk)
<type 'dict'>
```

Hatırlarsanız, listeleri tanımlayan sözcük “list”, demetleri tanımlayan sözcük ise “tuple” idi. Gördüğünüz gibi sözlükleri tanımlayan sözcüğümüz de “dict”...

Yukarıda boş bir sözlük oluşturduk. Dilerseniz şimdi de öğeleri olan bir sözlük oluşturalım:

```
>>> sozluk = {"elma": "meyve", "domates": "sebze", 1: "sayi"}
```

Sözlükler görünüş açısından öteki veri tiplerinden biraz farklıdır. Liste ve demet gibi veri tiplerinde her bir öğeyi birbirinden virgül ile ayırıyorduk. Aslında sözlüklerde de durum böyledir. Yani sözlüklerde de öğeleri birbirlerinden virgül ile ayırıyoruz. Ama gördüğünüz gibi, öğelerin biçiminde bazı farklılıklar var.

Sözlükler, **anahtar-değer** çiftlerinden oluşan bir veri tipidir. Burada mesela, *elma* bir “anahtar”, *meyve* ise bu anahtarın “değeri”dir. Aynı şekilde *sebze* değerinin anahtarı *domates*, *sayi* değerinin anahtarı ise *1*’dir. Dolayısıyla Python’da sözlük; “anahtar” ve “değer” arasında bağ kuran bir veri tipidir diyoruz.

Bu durumu daha net anlayabilmek için bir örnek daha verelim:

```
>>> sozluk = {"Adana": "01", "İstanbul": "34", "İzmir": "35"}
```

Bu sözlük üç öğeden oluşuyor. Bunu şu şekilde teyit edebilirsiniz:

```
>>> len(sozluk)
3
```

Bu sözlüğün her öğesi bir “anahtar-değer” çiftidir. Mesela *Adana* anahtar, *01* ise bu anahtarın değeridir.

Bir örnek daha verelim. Mesela sözlükleri kullanarak bir telefon defteri yazalım:

```
>>> telefon_defteri = {"Ahmet": "0533 123 45 67",
... "Salih": "0532 321 54 76",
... "Selin": "0533 333 33 33"}
```

Burada kodlarımızın sağa doğru biçimsiz bir biçimde uzamaması için virgülden sonra ENTER tuşuna basarak öğeleri tanımlamaya devam ettiğimize dikkat edin. Sağa doğru çok fazla uzamış olan kodlar hem görüntü açısından hoş değildir, hem de görüş alanını dağıttığı için okumayı zorlaştırır.

İsterseniz bir de yukarıdaki kodların metin düzenleyici içinde nasıl görüneceğine bakalım:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

telefon_defteri = {"Ahmet": "0533 123 45 67",
                   "Salih": "0532 321 54 76",
                   "Selin": "0533 333 33 33"}

print telefon_defteri
```

Kodlarımızı daha okunaklı bir hale getirmek için sözlüğü uygun yerlerinden bölerek alt satıra geçiriyoruz. Unutmayın, kod bir kez yazılır bin kez okunur. Okunaklı ve anlaşılır kodlar yazmak herkesin hayrınadır. Neyse, biz konumuza dönelim.

Sözlük tanımlarken dikkat etmemiz gereken birkaç nokta var. Bunlardan birincisi öğeleri belirlerken küme parantezlerini kullanıyor olmamız. İkincisi karakter dizilerinin yanısıra sayıları da tırnak içinde gösteriyor olmamız. İsterseniz sayıları tırnaksız kullanırsanız ne olacağını deneyerek görebilirsiniz. Ancak eğer gireceğiniz sayı boşluklu değilse ve 0 ile başlamıyorsa bu sayıyı tırnaksız da yazabilirsiniz. Üçüncüsü iki nokta üst üste ve virgüllerin nerede, nasıl kullanıldığına da dikkat etmeliyiz.

Sözlükleri nasıl oluşturacağımızı öğrendik. Şimdi gelelim bu sözlükleri nasıl kullanacağımıza...

## 11.2 Sözlük Öğelerine Erişmek

Bir sözlük oluşturduktan sonra, tabii ki bu sözlüğün öğelerine erişmek isteyeceksiniz. Bir sözlüğün tamamını ekrana dökmek için yapmamız gereken şey belli:

```
>>> sozluk = {"Python": "programlama dili",
... "İngilizce": "dil", "elma": "meyve"}

>>> print sozluk
```

`print` komutunu kullanarak bir sözlüğün tamamını ekrana dökülebiliyoruz. Peki ya biz bu sözlüğün öğelerine tek tek erişmek istersek ne yapacağız?

Hatırlarsanız liste ve demetlerin öğelerine tek tek şu şekilde erişiyorduk:

```
>>> liste = ["Ali", "Ahmet", "Mehmet"]
>>> liste[0]
'Ali'
>>> demet = ("Ali", "Ahmet", "Mehmet")
>>> demet[1]
'Ahmet'
```

Liste ve demetlerde öğeleri sıralarına göre çağırabiliyoruz. Çünkü liste ve demetler sıralı veri tipleridir. Yani liste ve demetlerdeki öğelerin her birinin bir sırası vardır. Ancak sözlükler öyle değildir. Sözlüklerde herhangi bir sıra kavramı bulunmaz. Mesela şu örneğe bakalım:

```
>>> a = {"a": 1, "b": 2, "c": 3}
>>> print a
{'a': 1, 'c': 3, 'b': 2}
```

Gördüğümüz gibi, öğeler tanımladığımız sırada görünmüyor. Sözlüklerde herhangi bir sıra kavramı olmadığı için şöyle bir girişim başarısızlığa uğrayacaktır:

```
>>> a[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 0
```

Sözlükler sırasız bir veri tipi olduğu için, sözlük öğelerini sıralarına göre değil, isimlerine göre çağırıyoruz.

Şuna bir bakalım:

```
>>> telefon_defteri = {"Ahmet": "0533 123 45 67",
... "Salih": "0532 321 54 76",
... "Selin": "0533 333 33 33"}
>>> telefon_defteri["Ahmet"]
'0533 123 45 67'
>>> telefon_defteri["Salih"]
'0532 321 54 76'
```

Gördüğümüz gibi, bu komutlar *Ahmet* ve *Salih* adlı anahtarların karşısında hangi değer varsa onu ekrana yazdırıyor. Dikkat edin, sözlükten öge çağırırken küme parantezlerini değil, köşeli parantezleri kullanıyoruz. Bu arada aklınızda bulunsun, sözlük içindeki öğeleri anahtara göre çağırıyoruz, değere göre değil. Yani iki nokta üst üste işaretinin solundaki ifadeleri kullanıyoruz öğeleri çağırırken, sağındakileri değil...

## 11.2.1 Sözlüklere Öğe Ekleme

Sözlüklere öğe eklemek son derece kolay bir işlemdir. Diyelim ki elimizde şöyle boş bir sözlük var:

```
>>> sozluk = {}
```

Bu sözlüğe öğe eklemek için şu yapıyı kullanıyoruz:

```
>>> sozluk[anahtar] = deger
```

Dilerseniz bu yapıyı somutlaştıracak bir örnek verelim:

```
>>> sozluk["Ad"] = "Ahmet"
```

Burada “Ad” bir anahtar, “Ahmet” ise bir değerdir. Yani sözlüklere öğe eklemek için hem bir anahtar, hem de bir değer belirtmemiz gerekiyor. Mesela bu sözlüğe bir öğe daha ekleyelim:

```
>>> sozluk["Soyad"] = "Su"
```

Şimdi sözlüğümüzü yazdıralım:

```
>>> print sozluk
```

```
{'Ad': 'Ahmet', 'Soyad': 'Su'}
```

Gördüğümüz gibi öğelerimiz sözlüğe eklenmiş. Bu yapıyı ve *for* döngüsünü kullanarak sözlüklere birden fazla öğeyi bir çırpıda ekleyebilirsiniz:

```
>>> liste = dir(list)
```

```
>>> sozluk = {}
```

```
>>> for anahtar, deger in enumerate(liste):  
...     sozluk[anahtar] = deger
```

```
>>> print sozluk
```

```
{0: '__add__', 1: '__class__', 2: '__contains__', 3: '__delattr__',  
4: '__delitem__', 5: '__delslice__', 6: '__doc__', 7: '__eq__',  
8: '__format__', 9: '__ge__', 10: '__getattr__', 11: '__getitem__',  
12: '__getslice__', 13: '__gt__', 14: '__hash__', 15: '__iadd__',  
16: '__imul__', 17: '__init__', 18: '__iter__', 19: '__le__',  
20: '__len__', 21: '__lt__', 22: '__mul__', 23: '__ne__', 24: '__new__',  
25: '__reduce__', 26: '__reduce_ex__', 27: '__repr__', 28: '__reversed__',  
29: '__rmul__', 30: '__setattr__', 31: '__setitem__', 32: '__setslice__',  
33: '__sizeof__', 34: '__str__', 35: '__subclasshook__', 36: 'append',  
37: 'count', 38: 'extend', 39: 'index', 40: 'insert', 41: 'pop',  
42: 'remove', 43: 'reverse', 44: 'sort'}
```

Gördüğümüz gibi `dir(list)` çıktısındaki öğeleri tek tek numaralandırıp bunları sözlüğe ekledik. Böylece artık liste metotlarını numara vererek çağırabiliriz:

```
>>> sozluk[1]
```

```
'__class__'
```

```
>>> sozluk[10]
```

```
'__getattribute__'  
  
>>> sozluk[20]  
  
'__len__'  
  
>>> sozluk[36]  
  
'append'  
  
>>> sozluk[40]  
  
'insert'
```

Yukarıdaki örnekte, sözlük anahtarları birer sayı olduğu için çıktıda öğeler sıralı olarak görünüyor. Bu, Python'daki sözlüklerin iç tasarımından kaynaklanan tesadüfi bir durumdur. Ne olursa olsun, sözlüklerde sıra kavramına güvenerek iş yapmamak gerekir. Mesela şu örneğe bakın:

```
>>> kayitlar = {}
```

Burada öncelikle boş bir sözlük oluşturduk. Bu sözlüğe tek tek öğe ekleyeceğiz:

```
>>> kayitlar["Ad"] = "Ahmet"  
>>> kayitlar["Soyad"] = "Okan"  
>>> kayitlar["Meslek"] = "Mimar"
```

Böylece sözlüğümüze üç farklı öğe eklemiş olduk. Şimdi bu sözlüğü ekrana dökelim:

```
>>> kayitlar  
{ 'Soyad': 'Okan', 'Ad': 'Ahmet', 'Meslek': 'Mimar' }
```

Gördüğünüz gibi, öğe sıralaması bozulmuş...

Bu arada, eğer kendi kendinize denemeler yapmışsanız, sözlüklerin Türkçe karakterleri düzgün gösteremediğini farketmişsinizdir. Örneğin:

```
>>> stok = {"Çilek": "100 kilo",  
... "Şeker": "5 kilo",  
... "Çay": "10 kilo",  
... "Kaşık": "100 adet"}  
  
>>> print stok  
{ '\x80ay': '10 kilo', '\x9eeker': '5 kilo',  
'Ka\x9f\x8dk': '100 adet', '\x80ilek': '100 kilo' }
```

Gördüğünüz gibi, Türkçe karakterlerin hiçbiri düzgün görüntülenemiyor. Hatırlarsanız buna benzer durumlarla listelerde de karşılaşmıştık. Orada bu sorunu çözmek için *for* döngüsünden yararlanmıştık. Sözlükler için de aynı yöntemi kullanabiliriz:

```
>>> for i in stok:  
...     print i  
  
Çay  
Şeker  
Kaşık  
Çilek
```



Bu sözlükteki öğelere, herhangi bir Türkçe karakter sorunu yaşamadan, normal bir şekilde erişebiliyoruz:

```
>>> print stok["Çay"]
>>> print stok["Şeker"]
```

## 11.3 Sözlük Öğelerini Değiştirmek

Bir önceki bölümde sözlüklere nasıl öğe ekleyeceğimizi gördük. Buna göre şöyle bir örnek verebiliyoruz:

```
>>> telefon_defteri = {}
>>> telefon_defteri["Zekiye"] = "0544 444 01 00"
>>> print telefon_defteri
{'Zekiye': '0544 444 01 00'}
```

Peki sözlüğümüzdeki bir öğenin değerini değiştirmek istersek ne yapacağız? Onu da şöyle yapıyoruz

```
>>> telefon_defteri["Zekiye"] = "0555 555 55 55"
```

Böylece sözlükteki Zekiye anahtarının değerini değiştirmiş olduk.

Buradan anladığımız şu: Bir sözlüğe yeni bir öğe eklerken de, varolan bir öğeyi değiştirirken de aynı komutu kullanıyoruz. Demek ki bir öğeyi değiştirirken aslında öğeyi değiştirmiyor, silip yerine yenisini koyuyoruz.

Bu arada, sözlük öğelerinde büyük-küçük harfin önemli olduğuna dikkat edin:

```
>>> telefon_defteri["zekiye"] = "0555 555 55 55"
>>> telefon_defteri
{'zekiye': '0555 555 55 55', 'Zekiye': '0555 555 55 55'}
```

"Zekiye" ve "zekiye" aynı görünse de aslında Python açısından bunların ikisi tamamen birbirinden farklıdır...

## 11.4 Sözlük Öğelerini Silmek

Eğer bir öğeyi sözlükten silmek istersek şu komutu kullanıyoruz:

```
>>> del telefon_defteri["Salih"]
```

Eğer biz sözlükteki bütün öğeleri silmek istersek şu komut kullanılıyor:

```
>>> telefon_defteri.clear()
```

Böylece boş bir sözlük elde etmiş olduk. Eğer sözlüğü tamamen ortadan kaldırmak isterseniz yine *del* komutundan yararlanabilirsiniz:

```
>>> del telefon_defteri
```

Yukarıdaki örneklerden birinde gördüğümüz `clear()` ifadesi, Python sözlüklerinin metotlarından biridir. Sözlüklerin bunun dışında başka metotları da vardır. Dilerseniz şimdi bu metotları inceleyelim.

## 11.5 Sözlüklerin Metotları

Demetlerin aksine, sözlükler üzerinde değişiklik yapabiliriz. Dolayısıyla sözlükler de, tıpkı listeler gibi, metot bakımından zengin bir veri tipidir. Sözlüklerin metotlarını listelemek için şu yöntemlerden herhangi birini izleyebilirsiniz:

```
>>> dir(dict)
>>> dir({})
>>> d = {}
>>> dir(d)
```

Bu yöntemlerin herhangi birini takip ettiğinizde şöyle bir çıktı elde edeceksiniz:

```
['__class__', '__cmp__', '__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', 'has_key', 'items',
 'iteritems', 'iterkeys', 'itervalues', 'keys', 'pop', 'popitem',
 'setdefault', 'update', 'values']
```

Burada bizi ilgilendirenler, her zamanki gibi şunlar olacaktır:

```
>>> for i in dir(dict):
...     if "_" not in i[0]:
...         print i
clear
copy
fromkeys
get
has_key
items
iteritems
iterkeys
itervalues
keys
pop
popitem
setdefault
update
values
```

Şimdi bu metotların en önemlilerini inceleyeceğiz.

### 11.5.1 keys() ve values()

Sözlük metotları arasında en önemlileri bu *keys()* ve *values()* adlı metotlardır. Kabaca söylemek gerekirse *keys()* metodu bir sözlükteki anahtarları, *values()* metodu ise sözlükteki değerleri verir.

Mesela:

```
>>> telefon_defteri = {"Ahmet": "0533 123 45 67",
... "Salih": "0532 321 54 76", "Selin": "0533 333 33 33"}

>>> print telefon_defteri.keys()

['Ahmet', 'Salih', 'Selin']

>>> print telefon_defteri.values()

['0533 123 45 67', '0532 321 54 76', '0533 333 33 33']
```

### 11.5.2 clear(), pop() ve popitem()

*clear()* metodu bir sözlüğün bütün öğelerini siler:

```
>>> telefon_defteri = {"Ahmet": "0533 123 45 67",
... "Salih": "0532 321 54 76", "Selin": "0533 333 33 33"}

>>> telefon_defteri.clear()

>>> telefon_defteri

{}
```

Gördüğünüz gibi, sözlük tamamen boşaldı.

*pop()* metodu bir sözlükteki öğeleri adlarına göre siler ve ekrana basar:

```
>>> telefon_defteri = {"Ahmet": "0533 123 45 67",
... "Salih": "0532 321 54 76", "Selin": "0533 333 33 33"}

>>> telefon_defteri.pop("Ahmet")

'0533 123 45 67'
```

Gördüğünüz gibi, sözlükteki *Ahmet* anahtarını sildik. Bu anahtar silinirken, bunun değeri de ekrana basıldı.

*popitem()* metodu ise bir sözlükteki rastgele bir anahtarı siler ve silinen anahtarın değerini ekrana basar:

```
>>> telefon_defteri.popitem()

('Selin', '0533 333 33 33')
```

Bu silme şekli tamamen tesadüfidir. Hangi öğenin silineceğini kontrol edemezsiniz.

### 11.5.3 items() ve iteritems()

*items()* metodu bir sözlük içindeki anahtar ve değerleri listeler:

```
>>> telefon_defteri.items()

[('Selin', '0533 333 33 33'), ('Ahmet', '0533 123 45 67'),
 ('Salih', '0532 321 54 76')]
```

Gördüğünüz gibi, burada sözlük içindeki bütün anahtar ve değerleri bir liste içinde yer alıyor. Her anahtar ve değer çifti de birer demet biçiminde.

Yukarıda bir demet dizisi elde ettiğimize göre, bu demetleri şu şekilde çözebiliriz (Bu işleme İngilizce’de “unpacking” adı veriliyor):

```
>>> for k, v in telefon_defteri.items():
...     print k, v
```

```
Selin 0533 333 33 33
Ahmet 0533 123 45 67
Salih 0532 321 54 76
```

*iteritems()* metodu da bir sözlük içindeki anahtar ve değerleri listeler. Ancak *items()* metodu ile *iteritems()* metodu arasında bazı kullanım farkları vardır. Örneğin:

```
>>> telefon_defteri.iteritems()

<dictionary-itemiterator object at 0x00EC2AB0>
```

Gördüğünüz gibi, *iteritems()* metodu listeyi çıktı olarak vermedi. Bu durum *iteritems()* metodunun çok önemli bir özelliğidir. *items()* metodu, anahtar-değer çiftlerinden oluşan listeyi derhal meydana getirip ekrana basar. *iteritems()* metodu ise anahtar-değer çiftlerinden oluşan listeyi meydana getirdiğini bildiren bir “nesne” üretir. *iteritems()* metodu ile oluşturduğunuz listeyi siz istediğiniz zaman ekrana dökebilirsiniz:

```
>>> for anahtar, deger in telefon_defteri.items():
...     print anahtar, deger
...
Selin 0533 333 33 33
Ahmet 0533 123 45 67
Salih 0532 321 54 76
```

*iteritems()* metodu, bu özelliği sayesinde çok büyük listeler üzerinde daha performanslı çalışacaktır.

### 11.5.4 iterkeys() ve itervalues()

Bu iki metot, biraz önce gördüğümüz *keys()* ve *values()* metotlarıyla aynı işi yapar. Bu metot çifti arasındaki fark, *items()* ve *iteritems()* metotları arasındaki fark gibidir.

### 11.5.5 copy()

Bu metot bir sözlüğü kopyalamamızı sağlar:

```
>>> yeni_rehber = telefon_defteri.copy()
```

### 11.5.6 get() ve has\_key()

Eğer amacınız bir anahtarın sözlük içinde varolup olmadığını denetlemekse *has\_key()* meto-  
dundan yararlanabilirsiniz:

```
>>> telefon_defteri.has_key("Veli")

False

>>> telefon_defteri.has_key("Selin")

True
```

Buna benzer bir şeyi şöyle de yapabileceğimizi görmüştük:

```
>>> telefon_defteri["Selin"]

'0533 333 33 33'

>>> telefon_defteri["Veli"]

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Veli'
```

Gördüğünüz gibi, eğer aranan anahtar sözlükte varsa o anahtarın değeri ekrana basılıyor. Ama eğer aranan anahtar sözlükte yoksa bir hata mesajı alıyoruz. Normalde bu hata mesajını al-  
mamak için *try... except:* bloklarından yararlanabiliriz:

```
>>> try:
...     telefon_defteri["Veli"]
... except KeyError:
...     print "Aranan öge sözlükte yok!"
...
Aranan öge sözlükte yok!
```

Ancak Python'da bunu yapmanın çok daha kolay bir yolu var. Şimdi şu senaryoyu inceleyin:

Diyelim ki bir hava durumu programı yazmak istiyoruz. Tasarımıza göre kullanıcı bir şehir  
adı girecek. Program da girilen şehre özgü hava durumu bilgilerini ekrana yazdıracak. Bunu  
yapabilmek için, daha önceki bilgilerimizi de kullanarak şöyle bir şey yazabiliriz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

soru = raw_input("Şehrinizin adını tamamı küçük "
"harf olacak şekilde yazınız: ")

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.")
```

Ama yukarıdaki yöntemin, biraz meşakkatli olacağı açık. Sadece üç şehir için hava durumu bilgilerini sorgulayacak olsak mesele değil, ancak onlarca şehri kapsayacak bir program üretmekse amacımız, yukarıdaki yöntem yerine daha pratik bir yöntem uygulamak gayet yerinde bir tercih olacaktır. İşte bu noktada programcının imdadına Python'daki sözlük veri tipi ve bu veri tipinin `get()` adlı metodu yetişecektir. Yukarıdaki kodların yerine getirdiği işlevi, şu kodlarla da gerçekleştirebiliriz:

```
#!/usr/bin/env python  
# -*- coding: utf-8 -*-  
  
soru = raw_input("Şehrinizin adını tamamı küçük "  
"harf olacak şekilde yazınız: ")  
  
cevap = {"istanbul": "gök gürültülü ve sağanak yağışlı",  
         "ankara": "açık ve güneşli", "izmir": "bulutlu"}  
  
print cevap.get(soru, "Bu şehre ilişkin havadurumu "  
"bilgisi bulunmamaktadır.")
```

Gördüğünüz gibi, ilk önce normal biçimde, kullanıcıya sorumuzu soruyoruz. Ardından da “anahtar-değer” çiftleri şeklinde şehir adlarını ve bunlara karşılık gelen hava durumu bilgilerini bir sözlük içinde depoluyoruz. Daha sonra, sözlük metodlarından biri olan `get()` metodunu seçiyoruz. Bu metod bize sözlük içinde bir değer var olup olmadığını denetleme imkânının yanı sıra, adı geçen değer sözlük içinde varolmaması durumunda kullanıcıya gösterilecek bir mesaj seçme olanağı da sunar. Python sözlüklerinde bulunan bu `get()` metodu bizi *else* veya *try-except* blokları kullanarak hata yakalamaya uğraşma zahmetinden de kurtarır.

Burada `print cevap.get(soru, "Bu şehre ilişkin hava durumu bilgisi bulunmamaktadır.")` satırı yardımıyla `soru` adlı değişkenin değerinin sözlük içinde var olup varolmadığını sorguluyoruz. Eğer kullanıcının girdiği şehir adı sözlüğümüz içinde bir anahtar olarak tanımlanmışsa, bu anahtarın değeri ekrana yazdırılacaktır. Eğer kullanıcının girdiği şehir adı sözlüğümüz içinde bulunmuyorsa, bu defa kullanıcıya *Bu şehre ilişkin hava durumu bilgisi bulunmamaktadır.* biçiminde bir mesaj gösterilecektir.

*if* deyimleri yerine sözlüklerden yararlanmanın, yukarıda bahsedilen faydalarının dışında bir de şu yararları vardır:

1. Öncelikle sözü geçen senaryo için sözlükleri kullanmak programcıya daha az kodla daha çok iş yapma olanağı sağlar.
2. Sözlük programcının elle oluşturacağı “if-elif-else” bloklarından daha performanslıdır ve bize çok hızlı bir şekilde veri sorgulama imkânı sağlar.
3. Kodların daha az yer kaplaması sayesinde programın bakımı da kolaylaşacaktır.
4. Tek tek “if-elif-else” blokları içinde şehir adı ve buna ilişkin hava durumu bilgileri tanımlamaya kıyasla sözlük içinde yeni anahtar-değer çiftleri oluşturmak daha pratiktir.

## 11.6 Bölüm Soruları

1. Basit bir Türkçe-İngilizce sözlük programı yazın. Yazdığınız programda kullanıcı Türkçe bir kelime sorup, bu kelimenin İngilizce karşılığını alabilmeli.

2. Rakamla girilen sayıları yazıyla gösteren bir program yazın. Mesela kullanıcı “5” sayısını girdiğinde programımız “beş” cevabını vermeli.

3. Python sözlüklerinde sıra kavramı yoktur. Yani bir sözlüğe girdiğiniz değerler çıktıda sıralı olarak görünmeyebilir. Ancak bir sözlükte anahtarlar sayı olduğunda Python bu sayıları sıraya dizmektedir. Python’un sayı değerli anahtarlara neden böyle davrandığını araştırın. Anahtarların birer sayı olduğu sözlüklerle bazı denemeler yaparak, çıktıda görünen öge sırasının hangi durumlarda bozulduğunu bulmaya çalışın. Mesela şu iki sözlüğün çıktılarını öge sıralaması açısından karşılaştırın:

```
a = {3: "üç", 2: "iki", 7: "yedi", 10: "on"}

b = {3: "üç", 2: "iki", 8: "sekiz", 9: "dokuz",
     5: "beş", 6: "altı", 4: "dört", 1: "bir",
     7: "yedi", 10: "on"}
```

4. Şu örnekte neden i yerine i[0] yazdığımızı açıklayın:

```
>>> for i in dir(dict):
...     if "_" not in i[0]:
...         print i
```

5. Şu programı, tekrar tekrar çalışacak şekilde yeniden yazın:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

soru = raw_input("Şehrinizin adını tamamı küçük "
"harf olacak şekilde yazınız: ")

cevap = {"istanbul": "gök gürültülü ve sağanak yağışlı",
         "ankara": "açık ve güneşli", "izmir": "bulutlu"}

print cevap.get(soru, "Bu şehre ilişkin havadurumu "
"bilgisi bulunmamaktadır.")
```

Kullanıcı “q” tuşuna basarak programdan çıkabilmeli.

6. Aşağıdaki sözlüğün hem anahtarlarını hem de değerlerini ekrana basın:

```
>>> stok = {"Çilek": "100 kilo",
... "Şeker": "5 kilo",
... "Çay": "10 kilo",
... "Kaşık": "100 adet"}
```

Burada her anahtar ve değer şu biçimde görünmeli:

```
Depoda 10 kilo 'Çay' mevcuttur.
Depoda 5 kilo 'Şeker' mevcuttur.
Depoda 100 adet 'Kaşık' mevcuttur.
Depoda 100 kilo 'Çilek' mevcuttur.
```

---

# Kümeler

---

Bu bölümde, bir dizi halinde incelediğimiz veri tiplerinin sonuncusu olan kümeleri inceleyeceğiz. Bu bölümün sonunda küme denen şeyin ne olduğunu ve ne işe yaradığını öğreneceksiniz.

İngilizce’de kümeye “set” diyorlar. Python’da nispeten yeni sayılabilecek bir araç olan kümeler 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 işlerimizi halletmemizi sağlayabilir.

Kümeler, matematikten bildiğimiz “küme” kavramının sahip olduğu bütün özellikleri taşır. Yani kesişim, birleşim ve fark gibi özellikler Python’daki setler için de geçerlidir.

## 12.1 Küme Oluşturmak

Kümelerin bize sunduklarından faydalanabilmek için elbette öncelikle bir küme oluşturmamız gerekiyor. Küme oluşturmak çok kolay bir işlemdir. Örneğin boş bir kümeyi şöyle oluşturuyoruz:

```
>>> bos_kume = set()
```

Listeler, demetler ve sözlüklerin aksine kümelerin ayırt edici bir işareti yoktur. Küme oluşturmak için `set()` fonksiyonundan yararlanıyoruz.

Yukarıda boş bir küme oluşturduk. İçinde öge de barındıran kümeleri ise şu şekilde oluşturuyoruz:

```
>>> kume = set(["elma", "armut", "kebab"])
```

Böylelikle, içinde öge barındıran 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", "kebab"]
```

```
>>> 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", "kebab")
>>> 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
```

Peki sözlükleri kullanarak küme oluşturabilir miyiz? Evet, neden olmasın?

```
>>> 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'}
```

Bir de şuna bakalım:

```
>>> kardiz = "Python Programlama Dili"
>>> kume = set(kardiz)
>>> print kume

set(['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.

```
>>> liste = ["elma", "armut", "elma", "kebab", "şeker", "armut",
... "çilek", "ağaç", "şeker", "kebab", "şeker"]

>>> for i in set(liste):
...     print i
...
ağaç
elma
şeker
kebab
çilek
armut
```

Gördüğünüz gibi, liste içinde birden fazla bulunan öğeler, Python'daki kümeler yardımıyla teke indirilebiliyor.

Öğrendiğimiz bu bilgi sayesinde, daha önce gördüğümüz *count()* metodunu da kullanarak, şöyle bir kod yazabiliriz:

```
>>> liste = ["elma", "armut", "elma", "kiraz",
... "çilek", "kiraz", "elma", "kebab"]

>>> for i in set(liste):
...     print "%s listede %s kez geçiyor!"%(i, liste.count(i))

elma listede 3 kez geçiyor!
kiraz listede 2 kez geçiyor!
armut listede 1 kez geçiyor!
kebab listede 1 kez geçiyor!
çilek listede 1 kez geçiyor!
```

Burada *set(liste)* ifadesini kullanarak, liste öğelerini eşsiz ve benzersiz bir hale getirdik.

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.

## 12.2 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()* fonksiyonlarını kullanıyorsak, kümeler için de *set()* adlı fonksiyondan yararlanacağız:

```
>>> dir(set)

['__and__', '__class__', '__contains__', '__delattr__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattr__', '__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 ve sözlükler gibi, değiştirilebilir bir veri tipidir.

### 12.2.1 clear metodu

Kümelerle ilgili olarak inceleyeceğimiz ilk metot *clear()*. Bu metodu daha önce sözlükleri çalışırken de görmüştük. Sözlüklerde bu metodun görevi sözlüğün içeriğini boşaltmak idi. Burada da aynı vazifeyi görür:

```
>>> km = set("adana")

>>> for i in km:
...     print i
...
a
d
n

>>> km.clear()

>>> km
set([])
```

Burada önce “km” adlı bir küme oluşturduk. Daha sonra da *clear()* metodunu kullanarak bu kümenin bütün öğelerini sildik. Artık elimizde boş bir sözlük var.

### 12.2.2 copy metodu

Listeler ve sözlükleri incelerken *copy()* adlı bir metot öğrenmiştik. Bu metot aynı zamanda kümelerle birlikte de kullanılabilir. Üstelik işlevi de aynıdır:

```
>>> km = set("kahramanmaraş")
>>> yedek = km.copy()
>>> yedek
set(['a', 'r', 'h', 'k', 'm', '\x9f', 'n'])
>>> km
set(['a', 'h', 'k', 'm', 'n', 'r', '\x9f'])
```

Burada bir şey dikkatinizi çekmiş olmalı. “km” 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...

### 12.2.3 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", "kebab"])
>>> kume.add("çilek")
>>> print kume
set(['elma', 'armut', 'kebab', '\x87ilek'])
```

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)
... 
```

```
>>> kume
set([1, 2, 3, 'elma', 'kebab', '\x87ilek', '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.

### 12.2.4 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)

set([1, 5])
```

Demek ki k1'in k2'den farkı buymuş. Peki k2'nin k1'den farkını bulmak istersek ne yapacağız?

```
>>> k2.difference(k1)

set([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.

### 12.2.5 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)

set([2])

>>> print k1
```

```
set([2])
>>> print k2
set([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 öge idi. Dolayısıyla *difference\_update()* metodunu uyguladığımızda k1'in öğelerinin silinip yerlerine 2 adlı ögenin geldiğini görüyoruz.

### 12.2.6 discard metodu

Bir önceki bölümde öğrendiğimiz *add()* metodu yardımıyla, önceden oluşturduğumuz bir kümeye yeni öğeler ekleyebiliyorduk. Bu bölümde öğreneceğimiz *discard()* metodu ise kümeden öge silmemizi sağlayacak:

```
>>> hayvanlar = set(["kedi", "köpek", "at", "kuş", "inek", "deve"])
>>> hayvanlar.discard("kedi")
>>> print hayvanlar
set(['kuş', 'inek', 'deve', 'köpek', 'at'])
```

Eğer küme içinde bulunmayan bir öge 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 ögeyi silmeye çalıştığımızda hata vermemesi.

### 12.2.7 remove metodu

Bu metot da bir önceki bölümde gördüğümüz *discard()* metoduyla aynı işlevi yerine getirir. Eğer bir kümeden öge 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 getirir de aralarında önemli bir fark vardır. Hatırlarsanız *discard()* metoduyla, kümede olmayan bir ögeyi silmeye çalışırsak herhangi bir hata mesajı almayacağımızı söylemiştik. Eğer *remove()* metodunu kullanarak, kümede olmayan bir ögeyi 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.

## 12.2.8 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)

set([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

set([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:

```
# -*- coding: utf-8 -*-

tr = "şçöğüıŞÇÖĞÜİ"

parola = raw_input("Sisteme giriş için bir parola belirleyin: ")

if set(tr) & set(parola):
    print "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 "Parolanızda Türkçe harfler kullanmayın!"
```

Burada aslında şöyle bir şey demiş oluyoruz:

*Eğer set(tr) ve set(parola) kümelerinin kesişim kümesi boş değilse, kullanıcıya “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 = "şçöğüışçöğüı"
>>> parola = "çilek"
>>> set(tr) & set(parola)
set(['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 = "şçöğüışçöğüı"
>>> 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.

## 12.2.9 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)
set([1, 3])
>>> print k1
set([1, 3])
>>> print k2
set([1, 3, 5])
```

Gördüğünüz gibi, *intersection\_update()* metodu `k1`’in bütün öğelerini sildi ve yerlerine `k1` ve `k2`’nin kesişim kümesinin elemanlarını koydu.



### 12.2.10 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 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.

### 12.2.11 issubset metodu

Bu metod 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 metod yardımıyla öğrenebiliriz. Eğer bir küme başka bir kümenin alt kümesi ise bu metod 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.

### 12.2.12 issuperset metodu

Bu metod, 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.

### 12.2.13 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.

### 12.2.14 update metodu

Hatırlarsanız *add()* metodunu anlatırken şöyle bir örnek vermiştik:

```
>>> kume = set(["elma", "armut", "kebab"])

>>> yeni = [1, 2, 3]

>>> for i in yeni:
...     kume.add(i)
...

set([1, 2, 3, 'elma', 'kebab', '\x87ilek', '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", "kebab"])
>>> yeni = [1, 2, 3]
>>> kume.update(yeni)
>>> print kume
{1, 2, 3, 'elma', 'kebab', 'armut'}
```

Gördüğünüz gibi, *for* döngüsünü kullanmaya gerek kalmadan aynı sonucu elde edebildik.

### 12.2.15 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)
set([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)
set([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? İşte bu noktada yardımımıza *symmetric\_difference()* adlı metot yetişecek:

```
>>> a.symmetric_difference(b)
set([2, 4])
```

Böylece iki kümede de bulunmayan öğeleri aynı anda almış olduk.

### 12.2.16 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...

### 12.2.17 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", "kebab"])
>>> a.pop()
'elma'
```

Peki bu metot hangi ölçüte göre kümeden öğe siliyor? Herhangi bir ölçüt yok. Bu metot, küme öğelerini tamamen rastgele siliyor.

Böylelikle Python'da Listeler, Demetler, Sözlükler ve Kümeler konusunu bitirmiş olduk. Bu konuları sık sık tekrar etmek, hiç olmazsa arada sırada göz gezdirmek bazı şeylerin zihninizde yer etmesi açısından oldukça önemlidir.

## 12.3 Dondurulmuş Kümeler (Frozenset)

Daha önce de söylediğimiz gibi, kümeler üzerinde değişiklik yapabiliyoruz. Zaten kümelerin *add()* ve *remove()* gibi metotlarının olması bu durumu teyit ediyor. Ancak kimi durumlarda, öğeleri üzerinde değişiklik yapılamayan kümelere de ihtiyaç duyabilirsiniz. Hatırlarsanız listeler ve demetler arasında da buna benzer bir ilişki var. Demetler çoğu zaman, üzerinde değişiklik yapılamayan bir liste gibi davranır. İşte Python aynı imkanı bize kümelerde de sağlar. Eğer öğeleri üzerinde değişiklik yapılamayan bir küme oluşturmak isterseniz *set()* yerine *frozenset()* fonksiyonunu kullanabilirsiniz. Dilerseniz hemen bununla ilgili bir örnek verelim:

```
>>> dondurulmus = frozenset(["elma", "armut", "ayva"])
```

Dondurulmuş kümeleri bu şekilde oluşturuyoruz. Şimdi bu dondurulmuş kümenin metotlarına bakalım:

```
>>> dir(dondurulmus)
['__and__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattr__', '__gt__', '__hash__', '__init__',
 '__iter__', '__le__', '__len__', '__lt__', '__ne__', '__new__', '__or__',
```

```
'__rand__', '__reduce__', '__reduce_ex__', '__repr__', '__ror__', '__rsub__',  
'__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__',  
'__subclasshook__', '__xor__', 'copy', 'difference', 'intersection',  
'isdisjoint', 'issubset', 'issuperset', 'symmetric_difference', 'union']
```

Gördüğünüz gibi, *add()*, *remove()*, *update()* gibi, değişiklik yapmaya yönelik metotlar listede yok.

Dondurulmuş kümeler ile normal kümeler arasında işlev olarak hiçbir fark yoktur. Bu ikisi arasındaki fark, listeler ile demetler arasındaki fark gibidir.

## 12.4 Bölüm Soruları

1. Liste, demet, sözlük ve kümeleri karşılaştırdığınızda, bu veri tiplerini oluşturma bakımından kümelerle öteki veri tipleri arasında nasıl bir fark görüyorsunuz?
2. Bir küme içinde herhangi bir öğenin bulunup bulunmadığını nasıl sorgulayabileceğimizi gösteren bir örnek kod yazın.
3. Karakter dizilerini, demetleri ve listeleri kullanarak küme oluşturabiliyoruz. Ancak aynı şeyi sayıları kullanarak yapamıyoruz. Sayılardan küme oluşturmak istediğimizde şöyle bir hata mesajı veriyor bize Python:

```
>>> a = 10  
  
>>> set(a)  
  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: 'int' object is not iterable
```

Burada Python bize kabaca şöyle bir şey söylemiş oluyor: *"Sayılar üzerinde döngü kurulamaz!"*. Aldığınız bu hata mesajının anlamı üzerinde düşünün. Bu hatayı başka hangi durumlarda alırsınız?

4. Kullanıcıya renk adları soran bir program yazın. Kullanıcıdan gelen renk adlarını programın sonunda kullanıcıya bildirin. Ancak kullanıcının girdiği her renk adı kendisine bir kez bildirilmeli, yani renk adları tekrar etmemeli.

5. Demetler bölümünde şöyle bir soru sormuştuk:

Diyelim ki elimizde şöyle bir liste var:

```
liste = ["elma", "armut", "elma", "kiraz",  
        "çilek", "kiraz", "elma", "kebab"]
```

Bu listedeki her bir öğenin, listede kaç kez geçtiğini söyleyen bir program yazın. Programınız tam olarak şöyle bir çıktı vermeli:

```
elma öğesi listede 3 kez geçiyor!  
armut öğesi listede 1 kez geçiyor!  
kiraz öğesi listede 2 kez geçiyor!  
çilek öğesi listede 1 kez geçiyor!  
kebab öğesi listede 1 kez geçiyor!
```

Kümeler bölümünde öğrendiğiniz bilgileri de dikkate alarak bu programı yeniden yazın.

---

# Fonksiyonlar

---

Şimdi şöyle bir düşünün: Diyelim ki çalıştığınız işyerinde her gün bir yerlere dilekçe gönderiyorsunuz. Aslında farklı yerlere gönderdiğiniz bu dilekçeler temel olarak aynı içeriğe sahip. Yani mesela dilekçeyi Mehmet Bey'e göndereceğiniz zaman yazıya Mehmet Bey'in adını; Ahmet Bey'e göndereceğiniz zaman ise Ahmet Bey'in adını yazıyorsunuz. Ayrıca tabii dilekçe üzerindeki tarih bilgilerini de güne göre düzenliyorsunuz. Mantıklı bir insan, yukarıdaki gibi bir durumda elinde bir dilekçe taslağı bulundurur ve sadece değiştirmesi gereken kısımları değiştirip dilekçeyi hazır hale getirir. Her defasında aynı bilgileri en baştan yazmaz. Dilerseniz anlattığımız bu durumu Python programlama dili ile temsil etmeye çalışalım:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

print """Sayın Mehmet Bey,

19.12.2009 tarihinde yaptığımız başvurunun sonuçlandırılması
hususunda yardımlarınızı rica ederiz.

Saygılarımızla,
Orçun Kunek"""
```

Bu dilekçe Mehmet Bey'e gidiyor. İsterseniz bir tane de Ahmet Bey'e yazalım...

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

print """Sayın Ahmet Bey,

15.01.2010 tarihinde yaptığımız başvurunun sonuçlandırılması
hususunda yardımlarınızı rica ederiz.

Saygılarımızla,
Orçun Kunek"""
```

Burada dikkat ederseniz iki dilekçenin metin içeriği aslında aynı. Sadece bir-iki yer değişiyor. Bu kodları bu şekilde yazmak oldukça verimsiz bir yoldur. Çünkü bu sistem hem programcıya çok vakit kaybettirir, hem fazlasıyla kopyala-yapıştır işlemi gerektirir, hem de bu kodların bakımını yapmak çok zordur. Eğer dilekçe metni üzerinde bir değişiklik yapmak isterseniz program içindeki ilgili kodları tek tek bulup düzeltmeniz gerekir. Yani mesela yukarıdaki iki dilekçenin aynı program içinde olduğunu varsayarsak, dilekçe metnindeki bir hatayı düzeltmek

istediğimizde aynı düzeltmeyi birkaç farklı yerde yapmamız gerekir. Örneğin yukarıdaki dilekçe metninde bir yazım hatası var. İlk satırda “tarihinde” yazacağımıza “tariginde” yazmışız... Tabii biz Mehmet Bey’e yazdığımız dilekçenin metnini Ahmet Bey’e yazacağımız dilekçeyi hazırlarken kopyala-yapıştır yaptığımız için aynı hata Mehmet Bey’e gidecek dilekçede de var. Şimdi biz bu yazım hatasını düzeltmek istediğimizde, bütün dilekçe metinlerini tek tek gözden geçirmemiz gerekecektir. Hele bir de dilekçe sayısı çok fazlaysa bu işlemin ne kadar zor olacağını bir düşünün. Tabii bu basit durum için, her metin düzenleyicide bulunan “bul-değiştir” özelliğini kullanabiliriz. Ama işler her zaman bu kadar kolay olmayabilir. Bu bakım ve taslaklama işini kolaylaştıracak bir çözüm olsa ve bizi aynı şeyleri tekrar tekrar yazmaktan kurtarsa ne kadar güzel olurdu, değil mi? İşte Python buna benzer durumlar için bize “fonksiyon” denen bir araç sunar. Biz de bu bölümde bu faydalı aracı nasıl kullanacağımızı öğrenecek ve bu aracı olabildiğince ayrıntılı bir şekilde incelemeye çalışacağız.

## 13.1 Fonksiyonları Tanımlamak

Python’da fonksiyonları kullanabilmek için öncelikle fonksiyonları **tanımlamamız** gerekiyor. Fonksiyon tanımlamak için `def` adlı bir parçacıktan yararlanacağız. Python’da fonksiyonları şöyle tanımlıyoruz:

```
def fonksiyon_adi():  
    ...
```

Gördüğünüz gibi, önce `def` parçacığını, ardından da fonksiyonumuzun adını yazıyoruz. Fonksiyon adı olarak istediğiniz her şeyi yazabilirsiniz. Ancak fonksiyon adı belirlerken, fonksiyonun ne işe yaradığını anlatan kısa bir isim belirlemeniz hem sizin hem de kodlarınızı okuyan kişilerin işini bir hayli kolaylaştıracaktır. Yalnız fonksiyon adlarında Türkçe karakter kullanmanız gerekiyor. Ayrıca fonksiyonları tanımlarken en sona parantez ve iki nokta üst üste işaretlerini de koymayı unutuyoruz.

Böylece ilk fonksiyonumuzu tanımlamış olduk. Ama henüz işimiz bitmedi. Bu fonksiyonun bir işe yarayabilmesi için bunun altını doldurmamız gerekiyor. Fonksiyon tanımının iki nokta üst üste işareti ile bitmesinden, sonraki satırın girintili olması gerektiğini tahmin etmişsinizdir. Gelin isterseniz biz bu fonksiyonun altını çok basit bir şekilde dolduralım:

```
def fonksiyon_adi():  
    print "merhaba fonksiyon!"
```

Böylece eksiksiz bir fonksiyon tanımlamış olduk. Burada dikkat etmemiz gereken en önemli şey, `def fonksiyon_adi():` satırından sonra gelen kısmın girintili yazılmasıdır.

Şimdi isterseniz bu fonksiyonu nasıl kullanabileceğimizi görelim:

```
#!/usr/bin/env python  
# -*- coding: utf-8 -*-  
  
def fonksiyon_adi():  
    print "merhaba fonksiyon!"
```

Bu kodları bir metin düzenleyiciye kaydedip çalıştırdığımızda hiç bir çıktı elde edemeyiz. Çünkü biz burada fonksiyonumuzu sadece tanımlamakla yetindik. Henüz bu fonksiyonun işletilmesini sağlayacak kodu yazmadık. Şimdi bu fonksiyonun hayat kazanmasını sağlayacak kodları girebiliriz:

```
#!/usr/bin/env python  
# -*- coding: utf-8 -*-
```

```
def fonksiyon_adi():
    print "merhaba fonksiyon!"

fonksiyon_adi()
```

Burada fonksiyonun çalışmasını sağlayan şey, en son satırdaki `fonksiyon_adi()` kodudur. Bu satırı ekleyerek, daha önce tanımladığımız fonksiyonu çağırmış oluyoruz. Bu satıra teknik olarak fonksiyon çağırısı (*function call*) adı verilir.

Yukarıdaki kodları çalıştırdığımızda ekrana *merhaba fonksiyon!* satırının yazdırıldığını göreceğiz.

Tebrikler! Böylece ilk eksiksiz fonksiyonunuzu hem tanımlamış, hem de çağırmış oldunuz. Dilerseniz, bu konunun en başında verdiğimiz dilekçe örneğini de bir fonksiyon haline getirelim:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def dilekce_gonder():
    print """\
    Sayın Mehmet Bey,

    19.12.2009 tarihinde yaptığımız başvurunun sonuçlandırılması
    hususunda yardımlarınızı rica ederiz.

    Saygılarımızla,
    Orçun Kunek"""

dilekce_gonder()
```

Elbette bu örnek Python'daki fonksiyonların bütün yeteneklerini ortaya koymaktan aciz. Üstelik bu fonksiyon, en başta bahsettiğimiz sorunları da çözemiyor henüz. Ama ne yapalım... Şu ana kadar öğrendiklerimiz ancak bu kadarını yapmamıza müsaade ediyor. Biraz sonra öğreneceklerimiz sayesinde fonksiyonlarla çok daha faydalı ve manalı işler yapabileceğiz.

Yeni bir bölüme geçmeden önce isterseniz fonksiyonlarla ilgili olarak buraya kadar öğrendiğimiz kısmı biraz irdeleyelim:

1. Python'da fonksiyonlar bizi aynı şeyleri tekrar tekrar yazma zahmetinden kurtarır.
2. Fonksiyonlar bir bakıma bir taslaklama sistemi gibidir. Biraz sonra vereceğimiz örneklerde bu durumu daha net olarak göreceğiz.
3. Python'da fonksiyonları kullanabilmek için öncelikle fonksiyonu tanımlamamız gerekir. Bir fonksiyonu tanımlamak için *def* adlı parçacıktan yararlanıyoruz.
4. Python'da bir fonksiyon tanımı şöyle bir yapıya sahiptir:

```
def fonksiyon_adi():
    ...
```

5. Fonksiyon adlarını belirlerken Türkçe karakter kullanmıyoruz. Fonksiyonlarımıza vereceğimiz adların olabildiğince betimleyici olması herkesin hayrınadır.
6. Elbette bir fonksiyonun işlevli olabilmesi için sadece tanımlanması yetmez. Ayrıca tanımladığımız fonksiyonun bir de gövdesinin olması gerekir. Fonksiyon gövdesini girintili olarak yazıyoruz. Dolayısıyla Python'da bir fonksiyon temel olarak iki kısımdan oluşur. İlk kısım fonksiyonun tanımlandığı başlık kısmı; ikinci kısım ise fonksiyonun içeriğini oluşturan gövde kısmıdır. Başlık ve gövde dışında kalan her şey fonksiyonun da dışındadır.
7. Bir fonksiyon, gövdedeki girintili kısmın bittiği yerde biter. Örneğin şu bir fonksiyondur:



```
def selamla():  
    print "merhaba!"  
    print "nasılsın?"
```

Bu fonksiyon `def selamla():` satırıyla başlar, `print "nasılsın?"` satırıyla biter.

8. Fonksiyonların işlevli olabilmesi için bu fonksiyonlar tanımlandıktan sonra çağrılmalıdır. Örneğin yukarıdaki fonksiyonu şöyle çağırıyoruz:

```
def selamla():  
    print "merhaba!"  
    print "nasılsın?"  
  
selamla()
```

Dediğimiz gibi, bu fonksiyon `def selamla():` satırıyla başlar, `print "nasılsın?"` satırıyla biter. Fonksiyon çağırısı dediğimiz `selamla()` satırı bu fonksiyonun dışındadır. Çünkü bu satır `selamla()` fonksiyonunun gövdesini oluşturan girintili kısmın dışında yer alıyor.

Eğer bu söylediklerimiz size biraz kafa karıştırıcı gelmişse, hiç endişe etmenize gerek yok. Tam olarak ne demek istediğimizi biraz sonra gayet net bir şekilde anlamanızı sağlayacak örnekler vereceğiz.

Dilerseniz bu bölümü kapatmadan önce fonksiyonlarla ilgili birkaç basit örnek daha yaparak bu konuyu ısınmanızı sağlayalım:

```
#!/usr/bin/env python  
# -*- coding: utf-8 -*-  
  
def tek():  
    print "Girdiğiniz sayı bir tek sayıdır!"  
  
def cift():  
    print "Girdiğiniz sayı bir çift sayıdır!"  
  
sayi = raw_input("Lütfen bir sayı giriniz: ")  
  
if int(sayi) % 2 == 0:  
    cift()  
  
else:  
    tek()
```

Burada `tek()` ve `cift()` adlı iki fonksiyon tanımladık. `tek()` adlı fonksiyonun görevi ekrana *Girdiğiniz sayı bir tek sayıdır!* çıktısı; `cift()` adlı fonksiyonun görevi ise *Girdiğiniz sayı bir çift sayıdır!* çıktısı vermek.

Daha sonra Python'un `raw_input()` fonksiyonunu kullanarak kullanıcıdan bir sayı girmesini istiyoruz. Ardından da kullanıcı tarafından girilen bu sayının tek mi yoksa çift mi olduğunu denetliyoruz. Eğer sayı 2'ye tam olarak bölünüyorsa çifttir. Aksi halde bu sayı tektir.

Burada `cift()` ve `tek()` adlı fonksiyonları nasıl çağırdığımıza dikkat edin. Eğer kullanıcının girdiği sayı 2'ye tam olarak bölünüyorsa, yani bu sayı çiftse, daha önce tanımladığımız `cift()` adlı fonksiyon devreye girecektir. Yok, eğer kullanıcının verdiği sayı 2'ye tam olarak bölünmüyorsa o zaman da `tek()` adlı fonksiyon devreye girer.

Bu kodlarda özellikle fonksiyonların nerede başlayıp nerede bittiğine dikkat edin. Daha önce de dediğimiz gibi, bir fonksiyon `def` parçacığıyla başlar, gövdesindeki girintili alanın sonuna kadar devam eder. Girintili alanın dışında kalan bütün kodlar o fonksiyonun dışındadır. Mesela

yukarıdaki örnekte *tek()* ve *cift()* birbirinden bağımsız iki fonksiyondur. Bu fonksiyonlardan sonra gelen sayı değişkeni de fonksiyon alanının dışında yer alır.

## 13.2 Fonksiyonlarda Parametre Kullanımı

Yukarıdaki örneklerde gördüğünüz gibi, bir fonksiyon tanımlarken, fonksiyon adını belirledikten sonra bir parantez işareti kullanıyoruz. Bu parantez işaretleri, örneklerde de gördüğümüz gibi hiçbir bilgi içermeyebilir. Yani bu parantezler boş olabilir. Ancak yapacağımız işin niteliğine göre bu parantezlerin içine birtakım bilgiler yerleştirmemiz gerekebilir. İşte bu bilgilere parametre adı verilir. Zaten Python fonksiyonları da asıl gücünü bu parametrelerden alır. Şimdi şöyle bir örnek verdiğimizizi düşünelim:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def selamla():
    print "merhaba, benim adım istihza!"

selamla()
```

Burada fonksiyonumuzu tanımladıktan sonra en sondaki `selamla()` satırı yardımıyla fonksiyonumuzu çağırdık. Yani tanımladığımız fonksiyona hayat öpücüğü verdik.

Ancak yukarıda tanımladığımız fonksiyon oldukça kısıtlı bir kullanım alanına sahiptir. Bu fonksiyon ekrana yalnızca *merhaba, benim adım istihza!* çıktısını verebilir. Eğer fonksiyonların yapabildiği şey bundan ibaret olsaydı, emin olun fonksiyonlar hiçbir işimize yaramazdı. Ama şimdi öğreneceğimiz parametre kavramı sayesinde fonksiyonlarımıza takla atmayı öğreteceğiz. Gelin isterseniz çok basit bir örnekle başlayalım:

```
def selamla(isim):
    print "merhaba, benim adım %s!" %isim
```

Belki fark ettiniz, belki de fark etmediniz, ama burada aslında çok önemli bir şey yaptık. Fonksiyonumuza bir parametre verdik! Şimdiye kadar tanımladığımız fonksiyonlarda, fonksiyon tanımı hep boş bir parantezden oluşuyordu. Ancak bu defa parantezimizin içinde bir değişken adı görüyoruz. Bu değişkenin adı `isim`. Fonksiyonlar söz konusu olduğunda, parantez içindeki bu değişkenlere parametre adı verilir. Fonksiyonumuzu tanımlarken bir parametre belirttikten sonra bu parametreyi fonksiyonun gövdesinde de kullandık. İsterseniz şimdi tanımladığımız bu fonksiyonun bir işe yarayabilmesi için fonksiyonumuzu çağıralım:

```
selamla("istihza")
```

Kodları bir arada görelim:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def selamla(isim):
    print "merhaba, benim adım %s!" %isim

selamla("istihza")
```

Bu kodları çalıştırdığımızda şöyle bir çıktı alırız:

```
merhaba, benim adım istihza!
```

Burada *selamla()* adlı fonksiyonu *istihza* argümanı ile birlikte çağırdık. Böylece çıktıda *istihza* değerini aldık. Burada terminolojiyle ilgili ufak bir not düşelim: Fonksiyonlar **tanımlanırken** parantez içinde belirtilen değerlere “parametre” adı verilir. Aynı fonksiyon **çağrılırken** parantez içinde belirtilen değerlere ise “argüman” adı verilir. Ama her iki durum için de “parametre” adının kullanıldığını da görebilirsiniz bazı yerlerde... Neyse... Biz bu terminoloji işini bir kenara bırakıp yolumuza devam edelim.

Eğer *selamla()* adlı fonksiyonu farklı bir argüman ile çağırırsak elbette alacağımız çıktı da farklı olacaktır:

```
selamla("Ahmet Efendi")
```

Bu defa çıktımız şöyle:

```
merhaba, benim adım Ahmet Efendi!
```

Burada önemli olan nokta, *selamla()* fonksiyonunun bir adet parametreye sahip olmasıdır. Dolayısıyla bu fonksiyonu argümansız olarak veya birden fazla argümanla çağıramayız. Yani fonksiyonu şu şekillerde çağırmak hata almamıza yol açacaktır:

```
selamla("Ahmet", "Mehmet")
```

veya:

```
selamla()
```

Sanırım bu örnekler fonksiyonlardaki parametre kavramının ne olduğunu net bir biçimde ortaya koyuyor. Daha sonraki bölümlerde bu parametre kavramından bolca yararlanacağız. İlerde göreceğimiz örnekler ne kadar karmaşık olursa olsun, işin temeli aynen yukarıda anlattığımız gibidir. Eğer bu temeli iyi kavırsanız başka yerlerde göreceğiniz daha karmaşık örnekleri anlamakta zorlanmazsınız.

Dilerseniz bu anlattıklarımızla ilgili ufak bir örnek daha yapıp başka bir konuya geçelim...

Python’da, verilen sayıları toplayan *sum()* adlı bir fonksiyon olduğunu önceki derslerimizde öğrenmiştik. Bu fonksiyonu şöyle kullanıyorduk:

```
>>> sayilar = [45, 90, 43]
>>> sum(sayilar)

178
```

*sum()* fonksiyonu, kendisine argüman olarak verilen bir sayı listesinin öğelerini birbiriyle toplayıp sonucu bize bildiriyor. Ancak Python’da bu *sum()* fonksiyonuna benzer bir şekilde bir sayı listesini alıp, öğelerini birbiriyle çarpan hazır bir fonksiyon bulunmuyor. Python bize bu işlem için herhangi bir hazır fonksiyon sunmadığından, böyle bir durumda kendi yöntemimizi kendimiz icat etmek zorundayız. O halde hemen başlayalım. Diyelim ki elimizde şöyle bir sayı listesi var:

```
>>> sayilar = [45, 90, 43]
```

Soru şu: Acaba bu listedeki sayıları birbiriyle nasıl çarpabiliriz? Bunu yapmanın en kolay yolu, listedeki bütün sayıları 1’le çarpıp, bütün değerleri tek bir değişken içinde toplamaktır. Yani öncelikle değeri 1 olan bir değişken belirlememiz gerekiyor:

```
>>> a = 1
```

Daha sonra listedeki bütün sayıları *a* değişkeninin değeriyle çarpıp, bu değeri yine *a* değişkenine atayacağız:

```
>>> for i in sayilar:
...     a = a * i
```

Yukarıdaki kodları şöyle de yazabileceğimizi biliyorsunuz:

```
>>> for i in sayilar:
...     a *= i
```

Böylece listedeki bütün sayıların çarpımını gösteren değer a değişkenine atanmış oldu. İsterseniz bu a değişkeninin değerini yazdırıp sonucu kendi gözlerinizle görebilirsiniz:

```
>>> print a

174150
```

Gördüğünüz gibi, listedeki bütün sayıların çarpımı a değişkeninde tutuluyor.

Kodları topluca görelim:

```
>>> sayilar = [45, 90, 43]
>>> a = 1
>>> for i in sayilar:
...     a *= i
...
>>> print a

174150
```

Şimdi şöyle bir düşünün: Diyelim ki bir program yazıyorsunuz ve bu programın değişik yerlerinde, bir liste içindeki sayıları birbiriyle çarpmanız gerekiyor. Bunun için şöyle bir yol takip edebilirsiniz:

1. Önce bir sayı listesi tanımlarsınız,
2. Daha sonra, değeri 1 olan bir değişken tanımlarsınız,
3. Son olarak da listedeki bütün sayıları bu değişkenin değeriyle çarpıp, elde ettiğiniz değeri tekrar bu değişkene atarsınız,
4. Program içinde, gereken her yerde bu işlemleri tekrar edersiniz...

Bu yöntem, sizi aynı şeyleri sürekli tekrar etmek zorunda bıraktığı için oldukça verimsiz bir yoldur. İşte Python'daki fonksiyonlar böyle bir durumda hemen devreye girer. Mantıklı bir programcı, yukarıdaki gibi her defasında tekerleği yeniden icat etmek yerine, tekerleği bir kez icat eder, sonra gereken yerlerde icat ettiği bu tekerleği kullanır. Biz de yukarıdaki işlemleri içeren bir fonksiyonu tek bir kez tanımlayacağız ve program içinde gereken yerlerde bu fonksiyonu çağıracağız. Şimdi gelin yukarıdaki işlemleri içeren fonksiyonumuzu tanımlayalım:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def carp(liste):
    a = 1
    for i in liste:
        a *= i

    print a
```

Böylece taslağımızı oluşturmuş olduk. Artık bu fonksiyonu kullanarak istediğimiz bir sayı grubunu birbiriyle rahatlıkla çarpabiliriz. Bunun için yapmamız gereken tek şey carp() adlı fonksiyonu çağırarak:

```
carp([3, 5, 7])
```

Burada dikkat ederseniz, *carp()* fonksiyonuna argüman olarak verdiğimiz sayıları bir liste içine aldık. Çünkü *carp()* fonksiyonu tek bir parametre alıyor. Eğer bu fonksiyonu şu şekilde çağırırsanız hata alırsınız:

```
carp(3, 5, 7)
```

Çünkü burada *carp()* fonksiyonuna birden fazla argüman verdik. Halbuki fonksiyonumuz sadece tek bir argüman alıyor. Elbette dilerseniz önce bir sayı listesi tanımlayabilir, ardından da bu listeyi fonksiyona argüman olarak verebilirsiniz:

```
sayılar = [3, 5, 7]  
carp(sayılar)
```

Şimdi kodları topluca görelim:

```
#!/usr/bin/env python  
# -*- coding: utf-8 -*-  
  
def carp(liste):  
    a = 1  
    for i in liste:  
        a = a * i  
    print a  
  
sayılar = [3, 5, 7]  
carp(sayılar)
```

Bu kodları çalıştırdığınızda 105 sonucunu alırsınız.

Bu arada, yukarıdaki kodlarda *carp()* fonksiyonuna ait *liste* adlı parametrenin yalnızca temsili bir isimlendirme olduğuna dikkat edin. Program içinde daha sonra fonksiyonu çağırırken argüman olarak kullanacağınız değerın “liste” adını taşıma zorunluluğu yoktur. Mesela bizim örneğimizde *carp()* fonksiyonunu “sayılar” adlı bir liste ile çağırdık... Yani yukarıdaki fonksiyonu şöyle de yazabilirdik:

```
#!/usr/bin/env python  
# -*- coding: utf-8 -*-  
  
def carp(osman):  
    a = 1  
    for i in osman:  
        a = a * i  
    print a
```

Daha sonra da yine bu fonksiyonu şu şekilde çağırabiliriz:

```
sayılar = [3, 5, 7]  
carp(sayılar)
```

Elbette fonksiyon içinde belirttiğiniz parametrelerin anlamlı bir ada sahip olması herkes için en doğrusudur.

Fonksiyonların işimizi ne kadar kolaylaştırdığını görüyorsunuz. Yapmak istediğimiz işlemleri bir fonksiyon olarak tanımlıyoruz ve gerektiği yerde bu fonksiyonu çağırarak işimizi hallediyoruz. Eğer işlemlerde bir değişiklik yapmak gerekirse, tanımladığımız fonksiyonu yeniden düzenlememiz yeterli olacaktır. Eğer fonksiyonlar olmasaydı, fonksiyon içinde tek bir kez tanımladığımız işlemi programın farklı yerlerinde defalarca tekrar etmemiz gerekecekti. Üstelik

işlemlerde bir değişiklik yapmak istediğimizde de, bütün programı baştan sona tarayıp değişiklikleri tek tek elle uygulamak zorunda kalacaktık.

Dilerseniz en başta verdiğimiz dilekçe örneğine tekrar dönelim ve o durumu fonksiyonlara uyarlayalım:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def dilekce_gonder(kime, tarih, kimden):
    print """\
    Sayın %s,

    %s tarihinde yaptığımız başvurunun sonuçlandırılması
    hususunda yardımlarınızı rica ederiz.

    Saygılarımızla,
    %s""" %(kime, tarih, kimden)

dilekce_gonder("Mehmet Bey", "19.12.2009", "Orçun Kunek")
```

Gördüğünüz gibi, yukarıdaki fonksiyon işimizi bir hayli kolaylaştırıyor. Burada fonksiyonumuzu sadece bir kez oluşturuyoruz. Ardından dilekçeyi kime göndereceksek, uygun bilgileri kullanarak yeni bir fonksiyon çağrısı yapabiliriz. Mesela dilekçeyi Ahmet Bey'e göndereceksek şöyle bir satır yazmamız yeterli olacaktır:

```
dilekce_gonder("Ahmet Bey", "21.01.2010", "Erdener Topçu")
```

Ayrıca dilekçe metninde bir değişiklik yapmak istediğimizde sadece fonksiyon gövdesini düzenlememiz yeterli olacaktır.

## 13.3 İsimli ve Sıralı Argümanlar

Bir önceki bölümde fonksiyon parametrelerini nasıl kullanacağımızı öğrendik. Python'da fonksiyonlara istediğiniz sayıda parametre verebilirsiniz. Mesela şöyle bir fonksiyon tanımlayabilirsiniz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def kayit_ekle(isim, soyisim, sehir, meslek, tel, adres):
    kayit = {}

    kayit["%s %s" %(isim, soyisim)] = [sehir, meslek,
                                         tel, adres]

    print "Bağlantı bilgileri kayıtlara eklendi!\n"

    for k, v in kayit.items():
        print k
        print "-"*len(k)
        for i in v:
            print i

kayit_ekle("Orçun", "Kunek",
           "Adana", "Şarkıcı",
           "0322 123 45 67", "Baraj Yolu")
```

Yine burada, kodlarımızın çirkin bir görüntü oluşturmaması için öğeleri nasıl alt satıra geçirdiğimize dikkat edin. Eğer Python kodlarına duyarlı bir metin düzenleyici kullanıyorsanız (mesela IDLE) virgül işaretlerinden sonra ENTER tuşuna bastığınızda düzgün girinti yapısı otomatik olarak oluşturulacaktır.

Bu kodlarda öncelikle *kayit\_ekle()* adlı bir fonksiyon tanımladık. Bu fonksiyon toplam altı adet parametre alıyor. Bunlar; *isim*, *soyisim*, *sehir*, *meslek*, *tel* ve *adres*.

Tanımladığımız bu fonksiyonun gövdesinde ilk olarak kayıt adlı bir sözlük oluşturduk. Bu sözlük, ekleyeceğimiz bağlantıya dair bilgileri tutacak. Daha sonra, oluşturduğumuz bu sözlüğe öge ekliyoruz. Buna göre, fonksiyon parametrelerinden olan *isim* ve *soyisim*; *kayit* adlı sözlükteki “anahtar” kısmını oluşturacak. *sehir*, *meslek*, *tel* ve *adres* değerleri ise *kayit* adlı sözlükteki “değer” kısmını meydana getirecek.

Sözlüğü oluşturduktan sonra ekrana *Bağlantı bilgileri kayıtlara eklendi!* biçiminde bir mesaj yazdırıyoruz. Daha sonra da *kayit* adlı sözlüğün öğelerini belli bir düzen çerçevesinde ekrana yazdırıyoruz.

Bu işlemi nasıl yaptığımıza dikkat edin. Python sözlüklerinde *items()* adlı bir metot olduğunu sözlükler konusunu işlerken öğrenmiştik. Bu metot yardımıyla bir sözlük içinde bulunan bütün anahtar ve değer çiftlerini elde edebiliyorduk. Dilerseniz buna bir örnek vererek bu metodu tekrar hatırlayalım.

Diyelim ki elimizde şöyle bir sözlük var:

```
>>> sozluk = {"programlama dili": "Python",  
... "metin duzenleyici": "Kwrite"}
```

Şimdi *items()* metodunu bu sözlük üzerine uygulayalım:

```
>>> print sozluk.items()  
  
[('programlama dili', 'Python'),  
( 'metin duzenleyici', 'Kwrite')]
```

Gördüğünüz gibi, *sozluk* adlı sözlüğe ait bütün anahtar ve değerler bir liste içinde demetler halinde toplandı. Şimdi şöyle bir şey yazalım:

```
>>> for k, v in sozluk.items():  
...     print k, v
```

Buradan şöyle bir çıktı alırız:

```
programlama dili Python  
metin duzenleyici Kwrite
```

Bu şekilde bir çıktı elde ettikten sonra artık bu çıktıyı istediğimiz şekilde biçimlendirebileceğimizi biliyoruz.

*items()* metodunu hatırlattığımıza göre biz tekrar *kayit\_ekle()* fonksiyonumuzu incelemeye devam edebiliriz. Biraz önce anlattığımız gibi, *kayit\_ekle()* fonksiyonu içindeki `for k, v in kayit.items():` satırında *k* değişkeni *kayit* adlı sözlükteki anahtarları, *v* değişkeni ise aynı sözlükteki değerleri temsil ediyor. Böylece sözlükteki anahtar ve değer çiftlerini birbirinden ayırmış olduk.

Bu işlemi yaptıktan sonra, öncelikle *k* değişkenini ekrana yazdırıyoruz. Yani *kayit* adlı sözlükteki anahtar kısmını almış oluyoruz. Bu kısım, fonksiyondaki *isim* ve *soyisim* parametrelerinin değerini gösteriyor.. `print "-"*len(k)` satırı ise, bir önceki satırda ekrana yazdırdığımız

isim ve soyismin altına, isim ve soyisim değerlerinin uzunluğu kadar çizgi çekmemizi sağlıyor. Böylece isim ve soyismi, fonksiyondaki öteki bilgilerden görsel olarak ayırmış oluyoruz.

En son olarak da kayıt adlı sözlüğün “değer” kısmındaki öğeleri tek tek ekrana yazdırıyoruz.

Fonksiyonumuzu başarıyla tanımladıktan sonra sıra geldi bu fonksiyonu çağırmaya...

Fonksiyonumuzu sırasıyla, *Orçun, KuneK, Adana, Şarkıcı, 0322 123 45 67* ve *Baraj Yolu* argümanlarıyla birlikte çağırıyoruz. Böylece bu kodları çalıştırdığımızda şöyle bir çıktı elde ediyoruz:

```
Bağlantı bilgileri kayıtlara eklendi!
```

```
Orçun KuneK
-----
Adana
Şarkıcı
0322 123 45 67
Baraj Yolu
```

İsterseniz, *kayit\_ekle()* fonksiyonundaki parametreleri kendiniz yazmak yerine kullanıcıdan almayı da tercih edebilirsiniz. Mesela şöyle bir şey yazabilirsiniz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def kayit_ekle(isim, soyisim, sehir, meslek, tel, adres):
    kayit = {}

    kayit["%s %s" %(isim, soyisim)] = [sehir, meslek,
                                         tel, adres]

    print "\nBağlantı bilgileri kayıtlara eklendi!\n"

    for k, v in kayit.items():
        print k
        print "-"*len(k)
        for i in v:
            print i

isi = raw_input("isim: ")
soy = raw_input("soyisim: ")
seh = raw_input("şehir: ")
mes = raw_input("meslek: ")
tel = raw_input("telefon: ")
adr = raw_input("adres: ")

kayit_ekle(isi, soy, seh, mes, tel, adr)
```

Yukarıdaki fonksiyonları kullanırken dikkat etmemiz gereken çok önemli bir nokta var. *kayit\_ekle()* adlı fonksiyonu kullanırken argüman olarak vereceğimiz değerlerin sırası büyük önem taşıyor. Yani bu değerleri, fonksiyon tanımındaki sıraya göre yazmamız gerek. Buna göre *kayit\_ekle()* fonksiyonunu çağırırken, ilk argümanımızın isim, ikincisinin soyisim, üçüncüsünün şehir, dördüncüsünün meslek, beşincisinin telefon, altıncısının ise adres olması gerekiyor. Aksi halde, bizim yukarıda verdiğimiz örnekte çok belli olmasa da, fonksiyondan alacağımız çıktı hiç de beklediğimiz gibi olmayabilir. Ancak takdir edersiniz ki, bu kadar fazla sayıda parametrenin sırasını akılda tutmak hiç de kolay bir iş değil. İşte bu noktada Python'daki isimli argümanlar (*keyword arguments*) devreye girer ve bizi büyük bir dertten kurtarır. Nasıl mı? İsterseniz yukarıda verdiğimiz örnekten yararlanalım:



```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def kayit_ekle(isim, soyisim, sehir, meslek, tel, adres):
    kayit = {}

    kayit["%s %s" %(isim, soyisim)] = [sehir, meslek,
                                         tel, adres]

    print "\nBağlantı bilgileri kayıtlara eklendi!\n"

    for k, v in kayit.items():
        print k
        print "-"*len(k)
        for i in v:
            print i

kayit_ekle(isim = "Abdurrahman",
           soyisim = "Çelebi",
           meslek = "Öğretmen",
           tel = "0212 123 45 67",
           sehir = "İstanbul",
           adres = "Çeliktepe")
```

Gördüğünüz gibi, *kayit\_ekle()* adlı fonksiyonumuzun argümanlarını isimleriyle birlikte çağırıyoruz. Böylece argümanları sıra gözetmeden kullanma imkânımız oluyor. Bizim örneğimizde bütün parametreler karakter dizilerinden oluştuğu için, isimli parametre kullanmanın faydası ilk bakışta pek belli olmuyor. Ama özellikle sayılar ve karakter dizilerini karışık olarak içeren fonksiyonlarda yukarıdaki yöntemin faydası daha belirgindir. Mesela şu örneğe bakalım:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def terfi_ettir(kisi, e_poz, y_poz, e_maas, z_orani):
    print ("%s, %s pozisyonundan %s pozisyonuna "
           "terfi etmiştir!" %(kisi, e_poz, y_poz))

    print ("Bu kişinin %s TL olan maaşı %s TL'ye "
           "yükseltilmiştir!" %(e_maas, e_maas +
                                (e_maas * z_orani / 100)))

terfi_ettir("Ahmet Öncel",
           "İş Geliştirme Uzmanı",
           "İş Geliştirme Müdürü",
           3500,
           25)
```

İşte bu örnekte, parametre/argüman sıralamasının önemi ortaya çıkar. Eğer burada mesela *Ahmet Öncel* argümanı ile *3500* argümanının yerini değiştirirseniz programınız hata verecektir. Çünkü bu fonksiyonda biz *3500* sayısını kullanarak aritmetik bir işlem yapıyoruz. Eğer *3500*'ün olması gereken yerde bir sayı yerine karakter dizisi olursa aritmetik işlem yapılamaz... Bu arada yukarıdaki fonksiyonun sağa doğru çok fazla yayılarak çirkin bir kod görüntüsü vermemesi için satırları nasıl alta kaydığımızı dikkat edin.

Yukarıdaki örneği, isimli argümanları kullanarak yazarsak sıralama meselesini dert etmemize gerek kalmaz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def terfi_ettir(kisi, e_poz, y_poz, e_maas, z_orani):
    print ("%s, %s pozisyonundan %s pozisyonuna "
            "terfi etmiştir!" %(kisi, e_poz, y_poz))

    print ("Bu kişinin %s TL olan maaşı %s TL'ye "
            "yükseltilmiştir!" %(e_maas, e_maas +
                                (e_maas * z_orani / 100)))

terfi_ettir(e_maas = 3500,
            e_poz = "İş Geliştirme Uzmanı",
            kisi = "Ahmet Öncel",
            y_poz = "İş Geliştirme Müdürü",
            z_orani = 25)
```

Teknik olarak söylemek gerekirse Python fonksiyonlarında öge sıralamasının önem taşıdığı argümanlara sıralı argüman (*positional argument*) adı verilir. Python fonksiyonlarında sıralı argümanlardan bolca yararlanılır. Ancak argüman sayısının çok fazla olduğu durumlarda isimli argümanları kullanmak da işinizi bir hayli kolaylaştırabilir.

## 13.4 Varsayılan Değerli Argümanlar

Önceki derslerimizden *range()* fonksiyonunu biliyorsunuz. Gelin isterseniz bu fonksiyona şimdi biraz daha yakından bakalım.

Hatırlarsanız *range()* fonksiyonunu şöyle kullanıyorduk:

```
>>> range(10)

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

Bu kullanım bize 0'dan 10'a kadar olan sayıların listesini veriyor. Dikkat ederseniz, biz yukarıda *range()* fonksiyonunu tek bir argüman ile çağırdık. Ancak biz dilersek yukarıdaki fonksiyonu ikinci bir argümanla daha çağırabiliriz:

```
>>> range(4, 10)

[4, 5, 6, 7, 8, 9]
```

Bu şekilde de 4'ten 10'a kadar olan sayıların listesini elde ediyoruz. *range()* fonksiyonu bunların dışında üçüncü bir argüman daha alır:

```
>>> range(4, 10, 2)

[4, 6, 8]
```

Bu üçüncü argüman, *range()* fonksiyonunun sayıları atlayarak göstermesini sağlıyor.

Gördüğünüz gibi *range()* fonksiyonu şu parametrelerden oluşuyor:

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

*range()* fonksiyonunu kullanabilmek için bu parametreler içinde sadece "bitiş\_değeri" adlı parametreye bir argüman vermemiz yeterli olacaktır:

```
range(10)
```

Öteki iki parametreyi ise boş geçebiliyoruz, çünkü o parametrelerin birer varsayılan değeri var. Bu yüzden o parametreleri belirtmesek de oluyor. O parametrelerin varsayılan değerlerinin şöyle olduğunu biliyorsunuz:

```
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.

İşte biz de kendi yazdığımız fonksiyonlarda böyle varsayılan değerler belirleyebiliriz. Mesela şu örneğe bir bakalım:

```
#!/usr/bin/env python  
# -*- coding: utf-8 -*-  
  
def kuvvet(sayi, kvt=2):  
    print sayi ** kvt
```

Burada, kendisine verilen bir sayının kuvvetini hesaplayan bir fonksiyon yazdık. Bu fonksiyonun varsayılan değerli bir parametresi var (`kvt`). Fonksiyonu şu şekilde çağırıyoruz:

```
kuvvet(12)
```

Fonksiyonu böyle çağırdığımızda otomatik olarak 12 sayısının 2. kuvveti hesaplanacaktır. Çünkü `kvt` parametresinin varsayılan değeri 2'dir. Ama eğer biz 12 sayısının farklı bir kuvvetini hesaplamak istersek aynı fonksiyonu şu şekilde çağırabiliriz:

```
kuvvet(12, 3)
```

Böylece 12 sayısının 3. kuvvetini hesaplamış olduk.

Gördüğümüz gibi, `kuvvet()` adlı fonksiyon toplam 2 parametreden oluşuyor. Bu fonksiyonu kullanabilmek için `sayi` parametresini mutlaka belirtmemiz gerekiyor. `kvt` parametresini belirtmesek de olur. Çünkü bunun varsayılan bir değeri var. Dolayısıyla biz fonksiyonu tek argümanla çağırdığımızda Python `kvt` parametresinin değerini 2 kabul edecektir. Eğer bizim istediğimiz şey herhangi bir sayının farklı bir kuvvetini hesaplamaksa bizim bunu açık açık belirtmemiz gerekiyor.

Gelin isterseniz bununla ilgili bir başka örnek daha verelim:

```
def bol(bolen, bolunen, hassas=True):
    sonuc = bolen / float(bolunen)

    if hassas == True:
        print float(sonuc)

    if hassas == False:
        print int(sonuc)
```

Burada *bol()* adlı bir fonksiyon tanımladık. Bu fonksiyon toplam 3 parametre alıyor: *bolen*, *bolunen* ve *hassas*. Bu üç parametreden sonuncusu varsayılan bir değere sahip. Dolayısıyla bizim bu fonksiyonu en az iki argüman ile çağırmamız gerekiyor:

```
bol(10, 3)
```

Bu programı çalıştırdığımızda şu sonucu alırız:

```
3.33333333333
```

Gördüğünüz gibi, sonuç içinde ondalık kısım da görünüyor. Eğer tamsayı şeklinde bir sonuç elde etmek istersek fonksiyonumuzu şöyle çağıracağız:

```
bol(10, 3, False)
```

Bu şekilde şöyle bir sonuç alırız:

```
3
```

Dediğimiz gibi, fonksiyonumuza baktığımız zaman, *bol()* fonksiyonunun üç adet parametre aldığını görüyoruz. Ama sonuncu parametreye varsayılan bir değer atadığımız için, fonksiyonumuzu kullanırken sadece iki argüman vermemiz yeterli oluyor. Eğer üçüncü parametreyi belirtmezsek, bu parametrenin değeri True varsayılacaktır. Parametrenin değeri True kabul edildiği için de fonksiyonumuz hesaplamada hassas bir sonuç verir. Eğer biz bu tür bir hassasiyet istemezsek üçüncü parametre olan *hassas*'ı False değeri ile çağırıyoruz.

Bu arada, yukarıdaki fonksiyonu şu şekilde tanımlayabileceğimizi de biliyoruz:

```
def bol(bolen, bolunen, hassas=True):
    sonuc = bolen / float(bolunen)

    if hassas:
        print float(sonuc)

    if not hassas:
        print int(sonuc)
```

Burada `if hassas:` ifadesi `if hassas == True:` ile eş anlamlıdır. `if not hassas:` ise `if hassas == False` ile aynı anlama gelir.

Varsayılan değerli fonksiyonlar tanımlarken dikkat etmemiz gereken önemli bir kural var. Bu tür fonksiyonlarda varsayılan değer, parametre sıralamasında en sonda gelmesi gerekiyor. Yani şöyle bir şey yazamayız:

```
def bol(hassas = True, bolen, bolunen):
    ...
```

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.

## 13.5 İstenen Sayıda Sıralı Argüman Kullanımı

Fonksiyonları anlatırken şöyle bir örnek vermiştik:

```
def selamla(isim):  
    print "merhaba, benim adım %s!" %isim
```

Bu fonksiyon, kendisine argüman olarak verilen herhangi bir ismi, mesela Ahmet'i, *merhaba, benim adım Ahmet!* şeklinde ekrana döküyordu. Eğer biz hem ismin hem de soyismin ekrana dökülmesini istersek şöyle bir şey yazabiliriz:

```
def selamla(isim, soyisim):  
    print "merhaba, ben ", isim, soyisim
```

Bu fonksiyonu şu şekilde çağırıyoruz:

```
selamla("Fırat", "Özgül")
```

Bu kodları çalıştırdığımızda şöyle bir çıktı alırız:

```
merhaba, ben Fırat Özgül
```

Yazdığımız bu fonksiyona istediğimiz sayıda parametre ekleyebileceğimizi biliyoruz. Mesela:

```
def sicile_ekle(isim, soyisim, tc_kimlik,  
               dog_yeri, dog_tarihi, adres):  
    ...
```

Bu parametreler böyle uzayıp gider... Örneğin yukarıdaki fonksiyon 6 adet parametre alıyor. Dolayısıyla biz bu fonksiyonu 6'dan az veya fazla sayıda argüman ile çağıramayız. Aksi halde Python bize hata mesajı gösterir. Peki ya biz kendimizi herhangi bir sayıyla sınırlamak istemezsek ne olacak? Yani mesela biz bir fonksiyon tanımlarken parametre sayısını önceden belirtmesek de istediğimiz kadar argümanla fonksiyonumuzu çağırabilsek olmaz mı? Örneğin şöyle bir şey yapamaz mıyız?

```
fonksiyon_adi(arg1, arg2, arg3, arg4, arg5, ...)
```

Elbette yapabiliriz. Şimdi şu kodları dikkatlice inceleyin:

```
#!/usr/bin/env python  
# -*- coding: utf-8 -*-  
  
def birlestir(*argumanlar):  
    a = ""  
    for i in argumanlar:  
        a += i + " "  
    print a
```

```
birlestir("Ahmet", "Öz", "Mersin",  
          "Mühendis", "0533 123 45 67")
```

Gördüğünüz gibi, bu fonksiyonun tam olarak kaç adet parametre aldığını belirtmedik. `"*arg"` gibi özel bir yapıdan faydalanarak bu fonksiyonun sınırsız sayıda argüman ile çağrılmasına olanak tanıdık.

Gelin isterseniz bir örnek daha yaparak durumu biraz daha netleştirelim. Hatırlarsanız yukarıda şöyle bir fonksiyon yazmıştık:

```
#!/usr/bin/env python  
# -*- coding: utf-8 -*-  
  
def carp(liste):  
    a = 1  
    for i in liste:  
        a *= i  
  
    print a
```

Bu fonksiyonu şu şekilde çağırıyorduk:

```
liste = [2, 3, 4]  
carp(liste)
```

Böylece liste içindeki bütün sayıları birbiriyle çarpabiliyorduk. Bu fonksiyonun özelliği yalnızca tek bir parametre almasıdır. Dolayısıyla bu fonksiyonu çağırırken 1'den az veya çok sayıda argüman kullanamıyoruz. Biz bu sınırlamayı aşmak için Python'daki listelerden yararlandık. Biraz önce öğrendiğimiz `"*arg"` yapısını kullanarak aynı fonksiyonu şöyle de yazabiliriz:

```
#!/usr/bin/env python  
# -*- coding: utf-8 -*-  
  
def carp(*liste):  
    a = 1  
    for i in liste:  
        a *= i  
  
    print a
```

Burada parametre sayısını kısıtlamadığımız için artık fonksiyonumuzu rahatlıkla şöyle çağırabiliriz:

```
carp(2, 3, 4, 5, 6, 7, 8)
```

Fonksiyonumuz, kendisine argüman olarak verilen bütün değerleri birbiriyle çarpacaktır...

Dilerseniz bu `"*arg"` yapısına biraz daha yakından bakalım. Şimdi şöyle bir şey yazdığımızı düşünün:

```
def fonk(*arg):  
    print arg
```

Bu fonksiyonu istediğiniz sayıda argümanla çağırabilirsiniz:

```
fonk("Ahmet", "Mehmet", "Veli", "Oğuz", "Ozan")
```

Bu fonksiyonu çalıştırdığınızda şöyle bir çıktı alırsınız:

```
('Ahmet', 'Mehmet', 'Veli', 'Oğuz', 'Ozan')
```

Gördüğünüz gibi, çıktımız bir demet. Fonksiyon çıktısını bu şekilde elde edeceğinizi bildikten sonra bu çıktıyı istediğiniz şekilde biçimlendirebilirsiniz. Mesela:

```
def fonk(*arg):  
    for sıra, isim in enumerate(arg):  
        print "%s --> %s" %(sıra, isim)  
  
fonk("Ahmet", "Mehmet", "Veli", "Oğuz", "Ozan")
```

Bu fonksiyonu çalıştırdığımızda şöyle bir çıktı elde ediyoruz:

```
0 --> Ahmet  
1 --> Mehmet  
2 --> Veli  
3 --> Oğuz  
4 --> Ozan
```

Python'daki bu "\*" işareti özel görevi olan bir araçtır. Bu işaret, bir dizi içindeki (bu demet olabilir, liste olabilir, vb.) bütün öğeleri tek tek fonksiyona uygular. Bu dediğimiz şeyi daha iyi anlayabilmek için şöyle bir örnek verelim:

```
def fonk(arg1, arg2, arg3):  
    print arg1, arg2, arg3
```

Bu basit fonksiyonu şu şekilde çağırabiliriz:

```
fonk("Ahmet", "Mehmet", "Ali")
```

Eğer istersek bu fonksiyonu, "\*" işaretini de kullanarak şu şekilde de çağırabiliriz:

```
ls = ["Ahmet", "Mehmet", "Ali"]  
  
fonk(*ls)
```

Eğer aynı fonksiyonu şu şekilde çağırırsanız hata alırsınız:

```
ls = ["Ahmet", "Mehmet", "Ali"]  
  
fonk(ls)
```

Burada "\*" işareti, listenin her bir öğesinin tek tek fonksiyona uygulanmasını sağlıyor.

Yukarıda şöyle bir örnek verdiğimizizi hatırlıyorsunuz:

```
#!/usr/bin/env python  
# -*- coding: utf-8 -*-  
  
def kayit_ekle(isim, soyisim, sehir, meslek, tel, adres):  
    kayit = {}  
  
    kayit["%s %s" %(isim, soyisim)] = [sehir, meslek,  
                                         tel, adres]  
  
    print "Bağlantı bilgileri kayıtlara eklendi!\n"  
  
    for k, v in kayit.items():  
        print k  
        print "-"*len(k)
```

```
        for i in v:
            print i

kayit_ekle("Orçun", "Kunek",
          "Adana", "Şarkıcı",
          "0322 123 45 67", "Baraj Yolu")
```

Biraz önce öğrendiğimiz bilgiyi kullanarak aynı fonksiyonu şu şekilde de yazabiliriz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def kayit_ekle(isim, soyisim, sehir, meslek, tel, adres):
    kayit = {}

    kayit["%s %s" %(isim, soyisim)] = [sehir, meslek,
                                         tel, adres]

    print "Bağlantı bilgileri kayıtlara eklendi!\n"

    for k, v in kayit.items():
        print k
        print "-"*len(k)
        for i in v:
            print i

isimler = ["Orçun", "Kunek", "Adana", "Şarkıcı",
           "0322 123 45 67", "Baraj Yolu"]

kayit_ekle(*isimler)
```

Normalde `kayit_ekle()` fonksiyonu 6 argüman alıyor. Biz burada bütün bu argümanları içeren bir listenin bütün öğelerini tek tek fonksiyona uyguluyoruz. Bunu yapmamızı sağlayan şey de işte biraz önce öğrendiğimiz "\*" işaretidir...

Yukarıdaki örnekte, `isimler` listesini oluşturan öğeleri kullanıcıdan da alabileceğimizi biliyoruz. Hatta bunun için daha önce şöyle bir kod da yazmıştık:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def kayit_ekle(isim, soyisim, sehir, meslek, tel, adres):
    kayit = {}

    kayit["%s %s" %(isim, soyisim)] = [sehir, meslek,
                                         tel, adres]

    print "\nBağlantı bilgileri kayıtlara eklendi!\n"

    for k, v in kayit.items():
        print k
        print "-"*len(k)
        for i in v:
            print i

isi = raw_input("isim: ")
soy = raw_input("soyisim: ")
seh = raw_input("şehir: ")
mes = raw_input("meslek: ")
```



```
tel = raw_input("telefon: ")
adr = raw_input("adres: ")

kayit_ekle(isi, soy, seh, mes, tel, adr)
```

Eğer istersek, biraz önce öğrendiğimiz bilgileri kullanarak bu kodları şu şekilde de kısaltabiliriz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def kayit_ekle(isim, soyisim, sehir, meslek, tel, adres):
    kayit = {}

    kayit["%s %s" %(isim, soyisim)] = [sehir, meslek,
                                         tel, adres]

    print "\nBağlantı bilgileri kayıtlara eklendi!\n"

    for k, v in kayit.items():
        print k
        print "-"*len(k)
        for i in v:
            print i

ls = []
sorular = ["isim: ", "soyisim: ", "şehir: ",
           "meslek: ", "tel: ", "adr: "]

for i in sorular:
    ls.append(raw_input(i))

kayit_ekle(*ls)
```

Burada önce `ls` adlı boş bir liste oluşturduk. Bu liste, kullanıcıdan `raw_input()` fonksiyonu yardımıyla alınan cevapları tutacak. Ardından da sorular adlı başka bir liste oluşturduk. Bu da kullanıcıya soracağımız soruları tutuyor. Daha sonra ise `raw_input()` fonksiyonunu bir `for` döngüsü içine sokarak, kullanıcıya sorduğumuz bütün soruların cevabını tek tek `ls` adlı listeye ekliyoruz. Son olarak da bu `ls` listesindeki bütün öğeleri tek tek `kayit_ekle()` adlı fonksiyona uyguluyoruz.

## 13.6 İstenen Sayıda İsimli Argüman Kullanımı

Bir önceki bölümde fonksiyonları kullanarak, istediğimiz sayıda isimsiz argümanı nasıl verebileceğimizi öğrenmiştik. Bu bölümde ise istediğimiz 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:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def adres_defteri(isim, soyisim, telefon):
    defter = {}

    defter["isim"] = isim
```

```
defter["soyisim"] = soyisim
defter["telefon"] = telefon

for k, v in defter.items():
    print("%s\t:\t%s" %(k, v))

adres_defteri(isim="Fırat",
              soyisim="Özgül",
              telefon="02122121212")
```

Burada öncelikle defter adlı bir sözlük tanımladık. Adres bilgilerini bu sözlük içine kaydedeceğiz. 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("%s\t:\t%s" %(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 ogeler in defter.items():
    print ogeler

('soyisim', 'Özgül')
('adres', 'İstanbul')
('telefon', '02122121212')
('eposta', 'firatozgul@firtzgl.com')
('cep', '05994443322')
('isim', 'Fırat')
```

Gördüğümüz gibi, yukarıdaki kod bize defter adlı sözlüğün “anahtar” ve “değer” çiftlerini 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("%s\t:\t%s" %(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:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

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 "%s\t:\t%s" %(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:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

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 sözlüklere nasıl öğe ekleyebileceğimizi anlattığımızı hatırlıyorsunuz:

```
sozluk = {}

sozluk["ayakkabı"] = 2
sozluk["elbise"] = 1
sozluk["gömlek"] = 4
```

Bu örnekte sozluk 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 ek-  
leme imkanı tanıyor bize... Dediğim gibi, bu parametreler bir sözlük olduğu için yukarıdaki  
örneği şu şekilde de yazabilirsiniz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

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 "%s\t:%s" % (k, v)

sozluk = {"eposta": "firatozgul@frtzgl.com",
         "adres": "İstanbul",
         "cep": "05994443322"}

adres_defteri(isim="Fırat",
              soyisim="Özgül",
              telefon="02122121212",
              **sozluk)
```

Burada sözlüğü önceden tanımladığımıza ve bunu fonksiyonu çağırırken doğrudan argüman olarak eklediğimize dikkat edin. Ayrıca *sozluk* argümanını fonksiyona yazarken yine çift yıldızlı yapıyı kullanmayı da unutmuyoruz.

## 13.7 Gömülü Fonksiyonlar (Built-in Functions)

Python'da en büyük nimetlerden biri de bu dilin yapısında bulunan gömülü fonksiyonlardır (*built-in functions*). Peki, bir fonksiyonun gömülü olması ne anlama gelir? "gömülü" demek,

“Python’un içinde yer alan” demektir. Yani gömülü fonksiyonlar, Python programlama dilinin içinde yer alan, hazır fonksiyonlardır. Mesela daha önce öğrendiğimiz ve sık sık kullandığımız `range()` fonksiyonu gömülü bir fonksiyondur. Bu fonksiyon Python’un içinde, kullanılmaya hazır bir şekilde bekler. Bu fonksiyonun işlevini yerine getirebilmesi için tanımlanmasına gerek yoktur. Python geliştiricileri bu fonksiyonu tanımlamış ve dilin içine “gömmüşlerdir”. Mesela `len()`, `enumerate()` ve `sum()` fonksiyonları da birer gömülü fonksiyondur.

Python’daki gömülü fonksiyonların listesine <http://docs.python.org/library/functions.html> adresinden erişebilirsiniz. Bir program yazarken, özel bir işlevi yerine getirmeniz gerektiğinde yukarıdaki adresi mutlaka kontrol edin. Bakın bakalım sizden önce birisi tekerleği zaten icat etmiş mi? Örneğin tamsayıları (integer) ikili sayılara (*binary*) çevirmeniz gerekiyorsa, oturup bu işlemi yapan bir fonksiyon tanımlamaya çalışmanız boş bir çaba olur. Bunun yerine hali-hazırda tanımlanıp dilin içine gömülmüş olan `bin()` fonksiyonunu kullanabilirsiniz. Yukarıdaki adreste bunun gibi onlarca gömülü fonksiyon göreceksiniz.

Peki, gömülü fonksiyonlarla, bizim kendi yazdığımız fonksiyonlar arasında tam olarak ne fark vardır? Bir defa, gömülü fonksiyonlar oldukça hızlı ve verimlidir. Çünkü bu fonksiyonlar Python geliştiricileri tarafından özel olarak optimize edilmiştir.

İkincisi (ve belki de en önemlisi), bu fonksiyonlar her an hazır ve nazırdır. Yani bu fonksiyonları kullanabilmek için özel bir şey yapmanıza gerek yoktur. İstedığınızde veya bu fonksiyonlar lazım olduğunda doğrudan kullanabilirsiniz bu fonksiyonları. Ama bizim tanımladığımız fonksiyonlar böyle değildir. Kendi yazdığımız fonksiyonları kullanabilmek için, bu fonksiyonu içeren modülü öncelikle içe aktarmamız (`import`) gerekir. Şu son söylediğim şeyin kafanızı karıştırmasına izin vermeyin. Yeri ve zamanı geldiğinde “modül” ve “içe aktarmak” kavramlarından söz edeceğiz.

## 13.8 global Deyimi

Bu bölümde *global* adlı bir deyimden söz edeceğiz. İsterseniz *global*’in ne olduğunu anlatmaya çalışmak yerine doğrudan bir örnekle işe başlayalım. Diyelim ki şöyle bir şey yazdık:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def fonk():
    a = 5
    print a

fonk()
```

Burada her zamanki gibi `fonk()` adlı bir fonksiyon tanımladık ve daha sonra bu fonksiyonu çağırdık. Sonuç olarak bu fonksiyonu çalıştırdığımızda 5 çıktısını aldık.

Gördüğünüz gibi, fonksiyon içinde `a` adlı bir değişkenimiz var. Şimdi şöyle bir şey yazarak bu `a` değişkenine ulaşmaya çalışalım:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def fonk():
    a = 5
    print a

fonk()
print "a'nın değeri: ", a
```

Bu kodları çalıştırdığımızda şöyle bir hata alırız:

```
Traceback (most recent call last):
  File "deneme.py", line 7, in <module>
    print a
NameError: name 'a' is not defined
```

Bu hata mesajı bize `a` diye bir değişken olmadığını söylüyor. Halbuki biz `fonk()` adlı fonksiyon içinde bu `a` değişkenini tanımlamıştık, değil mi? O halde neden Python `a` değişkenini bulamadığından yakınıyor? Hatırlarsanız bu bölümün en başında, bir fonksiyonun sınırlarının ne olduğundan söz etmiştik. Buna göre yukarıdaki `fonk()` adlı fonksiyon `def fonk():` ifadesiyle başlıyor, `print a` ifadesiyle bitiyor. Bu fonksiyonun etkisi bu alanla sınırlıdır. Python'da buna isim alanı (namespace) adı verilir. Herhangi bir fonksiyon içinde tanımlanan her şey o fonksiyonun isim alanıyla sınırlıdır. Yani mesela yukarıdaki fonksiyon içinde tanımladığımız `a` değişkeni yalnızca bu fonksiyon sınırları dâhilinde geçerlidir. Bu alanın dışına çıkıldığında `a` değişkeninin herhangi bir geçerliliği yoktur. O yüzden Python yukarıdaki gibi bir kod yazdığımızda `a` değişkenini bulamayacaktır. İsterseniz bu durumu daha iyi anlayabilmek için yukarıdaki örneği şöyle değiştirelim:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def fonk():
    a = 5
    print a

fonk()
a = 10

print "a'nın değeri: ", a
```

Bu kodları çalıştırdığımızda ise şöyle bir çıktı alırız:

```
5
a'nın değeri: 10
```

Buradaki ilk `5` sayısı fonksiyon içindeki `a`'nın değerini gösteriyor. Alt satırdaki `10` değeri ise `a`'nın fonksiyon sınırları dışındaki değerini... Gördüğümüz gibi, `a` değişkenini fonksiyon dışında da kullanabilmek için bu değişkeni dışarıda tekrar tanımlamamız gerekiyor. Peki, biz fonksiyon içindeki `a` değişkenine fonksiyon dışından da erişemez miyiz? Elbette erişebiliriz. Ama bunun için *global* adlı bir deyimden yararlanmamız gerekir. Dilerseniz yukarıda ilk verdiğimiz örnek üzerinden giderek bu *global* deyimini anlamaya çalışalım:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def fonk():
    a = 5
    print a

fonk()
print "a'nın değeri: ", a
```

Kodları bu şekilde yazdığımızda Python'un bize bir hata mesajı göstereceğini biliyoruz. Şimdi bu kodlara şöyle bir ekleme yapalım:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

```
def fonk():
    global a
    a = 5
    print a

fonk()
print "a'nın değeri: ", a
```

Burada yaptığımız şey, fonksiyonu tanımladıktan sonra fonksiyon gövdesinin ilk satırına `global a` diye bir şey eklemekten ibaret. Bu ifade fonksiyon içindeki `a` adlı değişkenin “global” olduğunu, yani fonksiyonun kendi sınırları dışında da geçerli bir değer olduğunu söylüyor. Bu kodları çalıştırdığımızda şöyle bir çıktı alıyoruz:

```
5
a'nın değeri: 5
```

Gördüğünüz gibi, *global* deyimini bir fonksiyon içindeki değerlere fonksiyon dışından da erişmemize yardımcı oluyor...

Şimdi şöyle bir şey yazdığımızı düşünün:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

a = 10

def fonk():
    a = 5
    return a

print "a'nın fonksiyon içindeki değeri", fonk()

print "a'nın fonksiyon dışındaki değeri: ", a
```

Buradaki *return* deyimine takılmayın biraz sonra bunun tam olarak ne işe yaradığını göreceğiz. Biz yalnızca yukarıdaki kodların çıktısına odaklanalım.

Burada öncelikle bir `a` değişkeni tanımladık. Bu değişkenin değeri `10`. Ardından *fonk()* adlı bir fonksiyon oluşturduk. Bu fonksiyon içinde değeri `5` olan bir `a` değişkeni daha tanımladık. Fonksiyon dışında ise, iki adet çıktı veriyoruz. Bunlardan ilki fonksiyon içindeki `a` değişkeninin değerini gösteriyor. İkincisi ise fonksiyon dışındaki `a` değişkeninin değerini... Yani bu kodları çalıştırdığımızda şöyle bir çıktı elde ediyoruz:

```
a'nın fonksiyon içindeki değeri 5
a'nın fonksiyon dışındaki değeri: 10
```

Burada fonksiyon içinde ve dışında aynı adda iki değişken tanımlamamıza rağmen, Python'daki “isim alanı” kavramı sayesinde bu iki değişkenin değeri birbirine karışmıyor. Ama bir de şu kodlara bakın:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

a = 10

def fonk():
    global a
    a = 5
    return a
```

```
print "a'nın fonksiyon içindeki değeri", fonk()
print "a'nın fonksiyon dışındaki değeri: ", a
```

Burada bir önceki kodlara ilave olarak global `a` satırını yazdık. Bu defa çıktımız şöyle oldu:

```
a'nın fonksiyon içindeki değeri 5
a'nın fonksiyon dışındaki değeri: 5
```

Gördüğünüz gibi, *global* deyimi fonksiyon dışındaki `a` değişkenini sildi. Şimdi bu noktada kendimize şu soruyu sormamız lazım: Acaba bizim istediğimiz şey bu mu? Özellikle üzerinde birkaç farklı kişinin çalıştığı büyük projelerde böyle bir özellik ne tür sorunlara yol açabilir? Üzerinde pek çok farklı kişinin çalıştığı büyük projelerde *global* deyiminin büyük sıkıntılar doğurabileceği apaçık ortada. Siz programın bir yerine bir değişken tanımlamaya çalışırken, başka bir geliştirici bir fonksiyon içinde *global* deyimini kullanarak fonksiyon dışındaki aynı adlı değişkenlerin değerini birbirine katmış olabilir... İşte bu tür sıkıntılardan ötürü, her ne kadar faydalı bir araçmış gibi görünse de, *global* deyiminin sakıncaları faydalarından çoktur. O yüzden yazdığınız programlarda *global* deyiminden mümkün olduğunca uzak durmanızda fayda var.

## 13.9 return Deyimi

Hatırlarsanız bir önceki bölümde şöyle bir fonksiyon tanımlamıştık:

```
def fonk():
    a = 5
    return a
```

Dikkat ederseniz burada *return* diye bir şey kullandık. Bu kelime Türkçe'de "vermek, döndürmek, iade etmek" gibi anlamlara gelir. Buna göre yukarıdaki fonksiyon `a` değişkenini "döndürüyor". Peki bu ne demek? Açıklayalım:

Python'da her fonksiyonun bir "dönüş değeri" vardır. Yani Python'daki bütün fonksiyonlar bir değer döndürür. Burada "döndürmek"ten kastımız bir işlemin sonucu olarak ortaya çıkan değeri vermektir. Mesela, "*Bu fonksiyonun dönüş değeri bir karakter dizisidir.*" veya "*Bu fonksiyon bir karakter dizisi döndürür.*" dediğimiz zaman kastettiğimiz şey; bu fonksiyonun işletilmesi sonucu ortaya çıkan değer bir karakter dizisi olduğudur. Örneğin yukarıdaki fonksiyon `a` adlı değişkeni döndürüyor ve bu `a` değişkeni bir tamsayı olduğu için, fonksiyonumuzun dönüş değeri bir tamsayıdır. Peki, fonksiyonlar bir değer döndürüyor da ne oluyor? Yani bir fonksiyonun herhangi bir değer döndürmesinin kime ne faydası var? İsterseniz bunu daha iyi anlayabilmek için yukarıdaki örneği bir de şöyle yazalım:

```
def fonk():
    a = 5
    print a
```

Burada herhangi bir değer döndürmüyoruz. Yaptığımız şey bir `a` değişkeni belirleyip, *print* deyimini kullanarak bunu ekrana bastırmaktan ibaret. Bu arada şunu da belirtelim: Her ne kadar biz bir fonksiyonda açık açık bir değer döndürmesek de o fonksiyon otomatik olarak bir değer döndürecektir. Herhangi bir değer döndürmediğimiz fonksiyonlar otomatik olarak `None` diye bir değer döndürür. Bunu şu şekilde test edebiliriz:

```
print fonk()
```



Fonksiyonu bu şekilde çağırdığımızda şöyle bir çıktı aldığımızı göreceksiniz:

```
5
None
```

İşte burada gördüğümüz `None` değeri, `fonk()` adlı fonksiyonumuzun otomatik olarak döndürdüğü değerdir. Yukarıdaki fonksiyonu *print* olmadan da çağırabileceğimizi biliyorsunuz:

```
fonk()
```

Bu defa `a` değişkeninin değeri ekrana dökülecek, ancak `None` değerini göremeyeceğiz. Şimdi şu fonksiyona bakalım:

```
def fonk():
    a = 5
    return a

fonk()
```

Burada ise ekranda herhangi bir çıktı göremeyiz. Bunun nedeni, bu fonksiyonda herhangi bir “print” işlemi yapmıyor olmamızdır. Biz burada sadece `a` değişkenini döndürmekle yetiniyoruz. Yani ekrana herhangi bir çıktı vermiyoruz. Bu fonksiyondan çıktı alabilmek için fonksiyonu şöyle çağırmamız gerekir:

```
print fonk()
```

“Peki, bütün bu anlattıkların ne işe yarar?” diye sorduğunuzu duyar gibiyim...

Bir defa şunu anlamamız lazım: *print* ve *return* aynı şeyler değildir. *print* deyimi bir mesajın ekrana basılmasını sağlar. *return* deyimi ise herhangi bir değer döndürülmesinden sorumludur. Siz bir fonksiyondan bir değer döndürdükten sonra o değerle ne yapacağınız tamamen size kalmış. Eğer tanımladığınız bir fonksiyonda bir değer döndürmek yerine o değeri ekrana basarsanız (yani *print* deyimini kullanırsanız) fonksiyonun işlevini bir bakıma kısıtlamış olursunuz. Çünkü tanımladığınız bu fonksiyonun tek görevi bir değeri ekrana basmak olacaktır. Ama eğer o değeri ekrana basmak yerine döndürmeyi tercih ederseniz, fonksiyonu hem ekranda bir mesaj göstermek için hem de başka işler için kullanabilirsiniz. Bunu anlayabilmek için şöyle bir örnek verelim:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def ekrana_bas():
    a = 5
    print a

print "a değişkeninin değeri: %s" %ekrana_bas()
```

Bu kodları çalıştırdığımızda şöyle bir çıktı alırız:

```
5
a değişkeninin değeri: None
```

Görüyorsunuz, işler hiç de istediğimiz gibi gitmedi. Halbuki biz *a değişkeninin değeri: 5* gibi bir çıktı alacağımızı zannediyorduk.

Daha önce de dediğimiz gibi, bir fonksiyondan herhangi bir değer döndürmediğimizde otomatik olarak `None` değeri döndürüldüğü için çıktıda bu değeri görüyoruz. Ama eğer yukarıdaki fonksiyonu şu şekilde tanımlasaydık işimiz daha kolay olurdu:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def ekrana_bas():
    a = 5
    return a

print "a değişkeninin değeri: %s" %ekrana_bas()
```

Gördüğünüz gibi, bu defa istediğimiz çıktıyı aldık. Bir de şu örneğe bakın:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def sayi_isle():
    sor = input("bir sayı girin: ")
    return sor

sayi = sayi_isle()

print "girdiğiniz sayı: %s" %sayi

if sayi % 2 == 0:
    print "girdiğiniz sayı çifttir"
else:
    print "girdiğiniz sayı tektir"

print "girdiğiniz sayının karesi: %s" %sayi ** 2
print "girdiğiniz sayının küpü: %s" %sayi ** 3
```

Burada *sayi\_isle()* adlı fonksiyonda kullanıcıya bir sayı sorup bu sayıyı döndürüyoruz. Daha sonra fonksiyonu çağırırken, bu döndürdümüz değerle istediğimiz işlemi yapabiliyoruz. İsterseniz bu fonksiyonu bir de *return* yerine *print* ile tanımlamayı deneyin. O zaman ne demek istediğimi gayet net bir biçimde anlayacaksınız.

## 13.10 Fonksiyonlarda pass Deyimi

Hatırlarsanız hata yakalama konusunu işlerken *pass* adlı bir deyimden söz etmiştik. İsterseniz o deyimi tekrar hatırlayalım.

“Pass” kelimesi Türkçe’de “geçmek, aşmak” gibi anlamlara gelir. Python’da ise bu deyim herhangi bir işlem yapmadan geçeceğimiz durumlarda kullanılır. Peki, bu ne demek?

Şu örneğe bir bakalım:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def deneme():
    liste = []
    while True:
        a = raw_input("Giriniz: ")
        if a == "0":
            pass
        else:
            liste.append(a)
```

```
print liste
deneme()
```

Burada gördüğümüz gibi, eğer kullanıcı 0 değerini girerse, bu değer listeye eklenmeyecek, Python hiçbir şey yapmadan bu satırı atlayacaktır. İşte *pass* buna benzer durumlarda, “*hiçbir şey yapmadan yola devam et!*” anlamı katar kodlarımıza.

Dediğimiz gibi, *pass* deyiminin bu görevlerini daha önce öğrenmiştik. *pass* deyimini yukarıdaki durumlar dışında bir de şöyle bir durumda kullanabilirsiniz: Diyelim ki bir program yazıyorsunuz. Bu programda bir fonksiyon tanımlayacaksınız. Fonksiyonun isminin ne olacağına karar verdiğiniz, ama fonksiyon içeriğini nasıl yazacağınızı düşünmediniz. Eğer program içinde sadece fonksiyonun ismini yazıp bırakırsanız programınız çalışma sırasında hata verecektir. İşte böyle bir durumda *pass* deyimini imdadınıza yetişir. Bu deyimini kullanarak şöyle bir şey yazabilirsiniz:

```
def bir_fonksiyon():
    pass
```

Fonksiyon tanımlarken fonksiyon gövdesini boş bırakamazsınız. Çünkü dediğimiz gibi, eğer gövdeyi boş bırakırsanız programınız çalışmaz. Böyle bir durumda, yukarıda gösterdiğimiz gibi fonksiyonu tanımlayıp gövde kısmına da bir *pass* deyimini yerleştirebilirsiniz. Daha sonra gövde kısmına ne yazacağınıza karar verdiğinizde bu *pass* deyimini silebilirsiniz.

## 13.11 Fonksiyonların Belgelendirilmesi

Her zaman söylediğimiz gibi, kod bir kez yazılır, bin kez okunur. Bu yüzden yazdığınız kodların anlaşılır olması her şeyden önemlidir. Program yazarken, yazdığınız kodların işlevinin yanısıra anlaşılır olmasına da dikkat etmeniz gerekir. Program yazarları, yazdıkları kodların daha kolay anlaşılmasını sağlamak için bazı yardımcı araçlardan da faydalanır. Mesela önceki derslerimizde gördüğümüz yorum (*comment*) kavramı bu yardımcı araçlardan biridir.

Python programcılarının, okunaklılığı artırmak için kullanabileceği bir başka yardımcı araç da belgelendirme dizileridir (*docstring*). Dilerseniz hemen bununla ilgili küçük bir örnek verelim:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def fonk():
    """boş bir fonksiyondur. Hiçbir
    işe yaramaz..."""
    pass
```

Gördüğünüz gibi, belgelendirme dizisini fonksiyon tanımının hemen ardından getiriyoruz. Belgelendirme dizilerinde üç tırnak kullanmak adettendir.

Bir fonksiyona ait belgelendirme dizisine erişmek için şu yöntemi kullanıyoruz:

Mesela *sum()* fonksiyonunun belgelendirme dizisine ulaşalım:

```
>>> print sum.__doc__

sum(sequence[, start]) -> value

Returns the sum of a sequence of numbers (NOT strings) plus the value
of parameter 'start' (which defaults to 0). When the sequence is
empty, returns start.
```

Kendi yazdığımız fonksiyonların belgelendirme dizilerine de aynı şekilde ulaşabiliriz. Önce fonksiyonumuzu tanımlayalım:

```
>>> def fonk():
...     """boş bir fonksiyondur. Hiçbir
...     işe yaramaz..."""
...     pass
```

Şimdi de fonksiyonumuzun belgelendirme dizisine ulaşalım:

```
>>> print fonk.__doc__

boş bir fonksiyondur. Hiçbir
işe yaramaz...
```

Gelin isterseniz biraz daha anlamlı bir örnek verelim:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def not_ortalaması_hesapla(ogrenci_sayisi=5):
    """Not ortalaması hesaplayan bir fonksiyon.
    Bu fonksiyonun aldığı tek argüman
    kullanıcıya kaç kez not bilgisi sorulacağını
    gösterir."""

    deneme = 0
    toplam = []
    while deneme < ogrenci_sayisi:
        deneme += 1
        toplam.append(int(raw_input("%s. öğrencinin notu: "
                                     % deneme)))

    print "Toplam %s öğrenci var." % deneme
    print ("Bu öğrencilerin not ortalaması: %s"
           %(sum(toplam)/deneme))

not_ortalaması_hesapla()
```

Gördüğümüz gibi, belgelendirme dizileri bize yazdığımız bir fonksiyonun ne işe yaradığını anlatma imkanı sağlıyor. Belgelendirme dizileri özellikle yazdığımız kodları okuyan başkaları için kıymetlidir. Belgelendirme dizileri sayesinde, yazdığımız kodları okuyanlar, yazdığımız şeyin ne işe yaradığı hakkında fikir sahibi olacaktır.

Yukarıdaki kodlarda yer alan belgelendirme dizisini ekrana basmak için kodlarınıza şöyle bir satır eklemelisiniz:

```
print not_ortalaması_hesapla.__doc__
```

Bu arada, yazdığınız belgelendirme dizilerinin de Python'un girintileme kurallarına uygun olması gerektiğine dikkat edin.

Böylece fonksiyonlar konusunu tamamlamış olduk. Artık yeni ve çok önemli bir konu olan "Modüllere" başlayabiliriz. Ama tabii ki önce bölüm sorularımız...

## 13.12 Bölüm Soruları

1. Esasında siz, bu bölümde incelediğimiz fonksiyon konusuna hiç de yabancı sayılmazsınız. Bu bölüme gelinceye kadar pek çok fonksiyon öğrenmiştik. Mesela daha önceki derslerimizden hangi fonksiyonları hatırlıyorsunuz?
2. Python'daki `sum()` fonksiyonunun yaptığı işi yapan bir fonksiyon yazın. Yazdığınız fonksiyon bir liste içindeki sayıların toplamını verebilmeli.
3. Kullanıcıya isim soran bir program yazın. Bu program kullanıcının ismini ekrana dökrebilmeli. Ancak eğer kullanıcının girdiği isim 5 karakterden uzunsa, 5 karakterden uzun olan kısım ekrana basılmamalı, onun yerine "..." işareti gösterilmelidir. Örneğin kullanıcıdan alınan isim Abdullah ise programınız "Abdul..." çıktısını vermeli.
4. Bir önceki soruda yazdığınız programda kullanıcı, içinde Türkçe karakterler bulunan bir isim girdiğinde programınızda nasıl bir durum ortaya çıkıyor? Örneğin programınız "Işıl" ismine nasıl bir tepki veriyor? Sizce bunun sebebi nedir?
5. Yukarıdaki fonksiyonlarla ilgili şöyle bir örnek vermiştik:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def tek():
    print "Girdiğiniz sayı bir tek sayıdır!"

def cift():
    print "Girdiğiniz sayı bir çift sayıdır!"

sayi = raw_input("Lütfen bir sayı giriniz: ")

if int(sayi) % 2 == 0:
    cift()
else:
    tek()
```

Bu kodlara şöyle bir baktığınızda, aslında bu kodları şu şekilde de yazabileceğimizi farketmişsinizdir:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

sayi = raw_input("Lütfen bir sayı giriniz: ")

if int(sayi) % 2 == 0:
    print "Girdiğiniz sayı bir çift sayıdır!"
else:
    print "Girdiğiniz sayı bir tek sayıdır!"
```

Bu kodları böyle değil de fonksiyon içinde yazmamızın sizce ne gibi avantajları vardır?

6. Argüman ile parametre arasındaki farkı açıklayın.
7. Standart bir kurulum betiğini taklit eden bir program yazın. Örneğin programınız şu aşamaları gerçekleştirebilmeli:
  - Kullanıcıya, "Kurulumla hoşgeldiniz!" mesajı göstermeli,

- Kullanıcıya bir lisans anlaşması sunulmalı ve bu anlaşmanın şartlarını kabul edip etmediğini sormalı,
- Eğer kullanıcı lisans anlaşmasının şartlarını kabul ederse kurulumla devam etmeli, aksi halde kurulumdan çıkmalı,
- Kullanıcıya, “standart paket”, “minimum kurulum” ve “özel kurulum” olmak üzere üç farklı kurulum seçeneği sunulmalı,
- Kullanıcının seçimine göre, programda kurulu gelecek özelliklerin bazılarını etkinleştirmeli veya devre dışı bırakmalı,
- Programın sistem üzerinde hangi dizine kurulacağını kullanıcıya sorabilmeli,
- Kurulumdan hemen önce, programın hangi özelliklerle kurulmak üzere olduğunu kullanıcıya son kez bildirmeli ve bu aşamada kullanıcıya kurulumdan vazgeçme veya kurulum özelliklerini değiştirme imkanı vermeli,
- Eğer kullanıcı kurulum özelliklerini değiştirmek isterse önceki aşamaya geri dönebilmeli,
- Eğer kullanıcı, seçtiği özelliklerle kurulumu gerçekleştirmek isterse program kullanıcının belirlediği şekilde sisteme kurulabilmeli,
- Son olarak kullanıcıya kurulumun başarıyla gerçekleştirildiğini bildiren bir mesaj göstermeli.

Not: Bu adımları kendi hayal gücünüze göre değiştirebilir, bunlara yeni basamaklar ekleyebilirsiniz.

8. Fonksiyonlar konusunu anlatırken şöyle bir örnek vermiştik:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def tek():
    print "Girdiğiniz sayı bir tek sayıdır!"

def cift():
    print "Girdiğiniz sayı bir çift sayıdır!"

sayi = raw_input("Lütfen bir sayı giriniz: ")

if int(sayi) % 2 == 0:
    cift()
else:
    tek()
```

Bu programın zayıf yönlerini bulmaya çalışın. Sizce bu program hangi durumlarda çöker? Bu programın çökmesini engellemek için ne yapmak gerekir?

---

# Modüller

---

Bu bölümde Python'daki en önemli konulardan biri olan modüllerden söz edeceğiz. Ancak modülleri kullanabilmek için tabii ki öncelikle “modül” denen şeyin ne olduğunu anlamamız gerekiyor.

Şöyle düşünün: Diyelim ki bir program yazıyorsunuz. Yazdığınız bu programın içinde karakter dizileri, sayılar, değişkenler, listeler, demetler, sözlükler, kümeler ve fonksiyonlar var. Programınız da .py uzantılı bir metin dosyası içinde yer alıyor. İşte bütün bu öğeleri ve veri türlerini içeren .py uzantılı dosyaya modül adı verilir. Bu bilgiye göre, şimdiye kadar yazdığınız ve bundan sonra yazacağınız bütün Python programları aynı zamanda birer modüldür.

Peki, bu bilginin bize ne faydası var? Ya da şöyle soralım: Yazdığımız bir Python programının modül olması neden bu kadar önemli?

Hatırlarsanız bir önceki bölümde Python'daki fonksiyonlardan bahsetmiştik. Yine hatırlarsanız o bölümde *carp()* adlı bir fonksiyon da tanımlamıştık. Bu fonksiyonu kullanabilmek için ne yapmamız gerektiğini biliyorsunuz. *carp()* fonksiyonuna ihtiyacımız olduğunda bu fonksiyonu çağırmamız yeterli oluyor. Şimdi şöyle bir düşünelim: Biz bu *carp()* fonksiyonuna ihtiyacımız olduğunda fonksiyonu çağırmak yoluyla aynı program içinde kullanabiliyoruz. Peki ya aynı fonksiyona başka bir Python programında da ihtiyacımız olursa ne yapacağız? O fonksiyonu kopyalayıp öbür Python programına yapıştıracak mıyız? Tabii ki hayır! Kodları alıp oradan oraya kopyalamak programcılık tecrübeniz açısından hiç de verimli bir yöntem değildir. Üstelik doğası gereği “kopyala-yapıştır” tekniği hatalara oldukça açık bir yoldur. Biz herhangi bir Python programında bulunan herhangi bir fonksiyona (veya niteliğe) ihtiyaç duyduğumuzda o fonksiyonu (veya niteliği) programımıza “aktaracağız”. Peki, bunu nasıl yapacağız?

Dediğimiz gibi bütün Python programları aynı zamanda birer modüldür. Bu özellik sayesinde Python programlarında bulunan fonksiyon ve nitelikler başka Python programları içine aktarılabilirler. Böylece bir Python programındaki işlevsellikten, başka bir Python programında da yararlanabilirsiniz.

İşte bu bölümde, bütün bu işlemleri nasıl yapacağımızı öğreneceğiz. Dilerseniz lafı daha fazla dolandırmadan modüller konusuna hızlı bir giriş yapalım.

## 14.1 Modüllerin Çeşitleri

Python'da modüller çeşit çeşittir. Temel olarak Python'da üç çeşit modülden söz edilebilir:

1. Kendi Yazdığınız Modüller

2. Geliştiricilerin Yazdığı Modüller
3. Üçüncü Şahısların Yazdığı Modüller

**Kendi yazdığınız modüller**, adından da anlaşılacağı gibi, bir Python programcısı olarak oturup sizin yazdığınız modüllerdir.

**Geliştiricilerin yazdığı modüller**, sizin modüllerinizin aksine Python geliştiricileri tarafından yazılıp Python programlama diline entegre edilmiş olan ve bu bakımdan dilin bir parçası olan modüllerdir.

**Üçüncü şahısların yazdığı modüller**, genellikle internet üzerindeki kaynaklardan edinebileceğiniz, Python programcıları tarafından hazırlanıp herkesin kullanımına sunulmuş olan modüllerdir.

Biz bu bölümde, yukarıda saydığımız üç modül türünü de olabildiğince ayrıntılı bir şekilde incelemeye çalışacağız. Dilerseniz öncelikle kendi yazdığımız modüllerden başlayalım...

## 14.2 Kendi Yazdığınız Modüller

Yukarıda da tanımladığımız gibi, kendi yazdığınız modüller, birer Python programcısı olarak oturup sizin yazdığınız Python programlarıdır. Mesela bir program yazdınız ve adını da *deneme.py* koydunuz. Bu programın içeriği şöyle olsun:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def carp(liste):
    a = 1
    for i in liste:
        a = a * i
    print a
```

İşte *deneme.py* adlı bu Python programı bir modüldür ve bu modül, *carp()* adlı tek bir fonksiyondan oluşur. Elbette modüller tek bir fonksiyondan ibaret olmak zorunda değildir. Bir modül pek çok farklı fonksiyondan oluşabilir. Örneğin:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def carp(liste):
    a = 1
    for i in liste:
        a = a * i
    return a

def bol(bolen, bolunen, hassas=True):
    sonuc = bolen / float(bolunen)

    if hassas:
        return float(sonuc)

    if not hassas:
        return int(sonuc)

def topl(*args):
    toplam = 0
    for i in args:
```



```
        toplam += i

    return toplam

def cıkar(bir, iki):
    return iki - bir
```

Burada dört farklı fonksiyona sahip bir modül tanımladık. Dilerseniz bu modülü masaüstüne *aritmetik.py* adıyla kaydedelim.

Bu noktada *Modül nedir?* sorusunu tekrar sormakta fayda var. Bu bölümün en başında yaptığımız tanıma göre, *carp()*, *bol()*, *topla()* ve *cıkar()* fonksiyonlarını içeren yukarıdaki *aritmetik.py* adlı program bir modüldür. Bu modülün adı da *aritmetik*'tir. Python'da modüller (genellikle) *.py* uzantısına sahiptir. Ancak bir modülün adı söylenirken bu *.py* uzantısı es geçilir ve sadece isim kısmı dikkate alınır. Bu yüzden elinizdeki *aritmetik.py* adlı program *aritmetik* modülü olarak adlandırılacaktır.

Dediğimiz gibi, bu modül içinde toplam dört adet fonksiyon var. Biz bu fonksiyonları aynı modül içinde rahatlıkla kullanabiliriz. Mesela *cıkar()* fonksiyonuna ihtiyacımız olduğunda yapmamız gereken tek şey bu fonksiyonu çağırmaktır:

```
print cıkar(1545, 1432)
```

Veya *topla()* fonksiyonuna gereksinim duyarsak, yine bu fonksiyonu çağırmak çok kolaydır:

```
print topla(3, 15, 43, 23, 88)
```

Peki ya bu fonksiyonlara başka bir programda da ihtiyaç duyarsak ne olacak? Böyle bir durumda yapacağımız şey bu fonksiyonu o program içinden çağırmak olacak. Ancak bunu yapmanın belli kuralları var. Peki, nedir bu kurallar? İşte şimdi bu kuralların ne olduğunu inceleyeceğiz...

## 14.3 Modülleri İçe Aktarmak

Bu bölümün başında şöyle bir cümle sarfetmiştik:

*"Herhangi bir Python programında bulunan herhangi bir fonksiyona (veya niteliğe) ihtiyaç duyduğumuzda o fonksiyonu (veya niteliği) programımıza aktarabiliriz."*

Python'da bir modülü başka bir programa taşıma işlemine içe aktarma adı verilir. İngilizce'de ise bu işleme *import* deniyor.

Şimdi yukarıda *aritmetik.py* adıyla kaydettiğimiz dosyanın bulunduğu dizin içinde bir komut satırı açıp Python'un etkileşimli kabuğunu çalıştırın. Mesela eğer *aritmetik.py* dosyasını masaüstüne kaydettiyseniz bir komut satırı açın, *cd Desktop* komutuyla masaüstüne gelin ve orada *python* komutunu vererek etkileşimli kabuğu başlatın. Şimdi şu komutu verin:

```
>>> import aritmetik
```

Eğer hiçbir şey olmadan bir alt satıra geçildiyse modülünüzü başarıyla içe aktardınız demektir. Eğer şöyle bir hata çıktısıyla karşılaşıyorsanız, muhtemelen Python'u masaüstünün olduğu dizinde başlatamamışsınızdır:

```
>>> import aritmetik
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named aritmetik
```

import aritmetik komutunun başarılı olduğunu varsayarak yolumuza devam edelim...

Modülü içe aktardıktan sonra *dir()* adlı özel bir fonksiyondan yararlanarak, içe aktardığımız bu modül içindeki kullanılabilir fonksiyon ve nitelikleri görebileceğimizi biliyorsunuz:

```
>>> dir(aritmetik)
```

Bu komut bize şöyle bir çıktı verir:

```
['__builtins__', '__doc__', '__file__', '__name__',  
 '__package__', 'bol', 'carp', 'cikar', 'topla']
```

Burada bizi ilgilendiren kısım *bol*, *carp*, *cikar* ve *topla* adlı öğeler. Bu çıktıdan anlıyoruz ki, *aritmetik* adlı modülün içinde *bol*, *carp*, *cikar* ve *topla* adlı fonksiyonlar var ve biz bu fonksiyonları kullanma imkânına sahibiz. O halde gelin mesela bu modül içindeki *carp()* adlı fonksiyonu kullanabilmek için sırasıyla şöyle bir şeyler yazalım:

```
>>> liste = [45, 66, 76, 12]  
>>> aritmetik.carp(liste)
```

Bu komutlar şöyle bir çıktı verir:

```
2708640
```

Gördüğünüz gibi, *aritmetik* modülü içindeki *carp()* adlı fonksiyonu kullanarak “liste” içindeki sayıları birbiriyle çarptık. *aritmetik* modülünü nasıl içe aktardığımıza ve bu modülün içindeki bir fonksiyon olan *carp()* fonksiyonunu nasıl kullandığımıza çok dikkat edin. Önce modülümüzün adı olan “aritmetik”i yazıyoruz. Ardından bir nokta işareti koyup, ihtiyacımız olan fonksiyonun adını belirtiyoruz. Yani şöyle bir formül takip ediyoruz:

```
modül_adı.fonksiyon
```

Böylece modül içindeki fonksiyona erişmiş olduk. Yalnız burada asla unutmamamız gereken şey öncelikle kullanacağımız modülü `import modül_adı` komutuyla içe aktarmak olacaktır. Modülü içe aktarmazsak tabii ki o modüldeki fonksiyon veya niteliklere erişemeyiz.

Şimdi *aritmetik.py* adlı dosyanızı açıp dosyanın en başına şu kodu ekleyin:

```
pi_sayisi = 22/7.0
```

Yani *aritmetik.py* dosyasının son hali şöyle olsun:

```
#!/usr/bin/env python  
# -*- coding: utf-8 -*-
```

```
pi_sayisi = 22/7.0
```

```
def carp(liste):  
    a = 1  
    for i in liste:  
        a = a * i  
    return a  
  
def bol(bolen, bolunen, hassas=True):  
    sonuc = bolen / float(bolunen)  
  
    if hassas:  
        return float(sonuc)  
  
    if not hassas:
```

```
        return int(sonuc)

def topla(*args):
    toplam = 0
    for i in args:
        toplam += i

    return toplam

def cikar(bir, iki):
    return iki - bir
```

Şimdi tekrar komut satırına dönüp şu komutu verin:

```
dir(aritmetik)
```

Bu komut biraz öncekiyle aynı çıktıyı verecektir. Halbuki biz modülümüze `pi_sayisi` adlı bir değişken daha ekledik. O halde neden bu değişken listede görünmüyor?

Python'da bir modülü komut satırında içe aktardıktan sonra eğer o modülde bir değişiklik yaparsanız, o değişikliğin etkili olabilmesi için modülü yeniden yüklemeniz gerekir. Bu işlemi `reload()` adlı özel bir fonksiyon yardımıyla yapıyoruz:

```
>>> reload(aritmetik)
```

Bu komut şöyle bir çıktı verir:

```
<module 'aritmetik' from 'aritmetik.py'>
```

Bu çıktı modülün başarıyla yeniden yüklendiğini gösteriyor. Şimdi `dir(aritmetik)` komutunu tekrar verelim:

```
>>> dir(aritmetik)
```

Bu defa listede `bol`, `carp`, `cikar` ve `topla` öğeleriyle birlikte `pi_sayisi` öğesini de göreceksiniz. Dolayısıyla artık bu öğeye de erişebilirsiniz:

```
>>> aritmetik.pi_sayisi

3.1428571428571428
```

Eh, yarım yamalak da olsa pi sayısını elde etmiş olduk!..

Buraya kadar modülleri hep etkileşimli kabukta içe aktardık. Ama tabii ki önemli olan bir modülü başka bir python programı içinden çağırabilmektir. İsterseniz şimdi bir modülü başka bir Python programı içinden nasıl çağırabileceğimizi öğrenelim.

Şimdi masaüstünde `test.py` adlı bir dosya oluşturun ve Kwrite, Kate, Gedit veya IDLE adlı metin düzenleyicilerden birini kullanarak bu dosyayı açın. Dosyaya şu satırları yazın:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import aritmetik
```

İlk iki satır zaten demirbaş. Burada önemli olan satır `import aritmetik`. Bu satırla, yine masaüstünde kayıtlı bulunan `aritmetik.py` adlı programı, yani `aritmetik` modülünü içe aktardık. Bu modülü içe aktardığımıza göre, artık bu modülün içindeki bütün fonksiyon ve nitelikleri kullanabiliriz. O halde yazmaya devam edelim:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import aritmetik

print aritmetik.cikar(1455, 32)
```

Yazdığımız son satırda, *aritmetik* modülünün içinde yer alan *cikar()* adlı fonksiyonu kullanmış olduk. Böylece daha önce bir kez tanımladığımız *cikar()* adlı fonksiyona başka bir Python programında da ihtiyaç duyduğumuzda, bu fonksiyonu tekrar yazmak zorunda kalmadan, gerekli fonksiyonu barındıran modülü programımız içine aktararak (*import*) işimizi hallettik. Aynı şekilde *aritmetik* modülü içindeki öbür fonksiyonları ve nitelikleri de kullanabilirsiniz. Mesela modül içinde yer alan *pi\_sayisi* niteliğini de kullanabiliriz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import aritmetik

print aritmetik.cikar(1455, 32)

print "pi sayısının değeri: %s" %aritmetik.pi_sayisi
```

Şimdi bu *test.py* adlı programı çalıştırın ve nasıl bir sonuç aldığınızı inceleyin.

## 14.4 Modülleri İçe Aktarma Yöntemleri

Python’da programımız içinde kullanacağımız modülleri birkaç farklı yöntemle içe aktarabiliriz. Biz şimdiye kadar sadece *import modül\_adı* yöntemini öğrendik. Hemen kısaca bu yöntemleri inceleyelim:

### **import modül\_adı**

Bu yöntemle bir modülü, bütün içeriğiyle birlikte içe aktarabiliriz. Başka bir deyişle bir modülün içinde ne var ne yoksa programımız içine davet edebiliriz. Yukarıda kullandığımız da zaten bu yöntemdir.

### **from modül\_adı import \***

Bu yöntemle bir modül içinde adı “\_\_” ile başlayanlar hariç bütün fonksiyonları programımız içine aktarabiliriz. Yani bu yöntem de tıpkı yukarıda anlatılan yöntemde olduğu gibi, bütün fonksiyonları alacaktır. Yalnız “\_\_” ile başlayan fonksiyonlar hariç...

Eğer bir modülü bu yöntemi kullanarak içe aktarmışsanız, içe aktardığımız modülün nitelik ve fonksiyonlarına doğrudan nitelik veya fonksiyon adını kullanarak erişebilirsiniz. Örneğin *import modül\_adı* yöntemiyle içe aktardığımız modüllerin nitelik ve fonksiyonlarını şöyle kullanıyorduk:

```
>>> modül_adı.fonksiyon
```

*from modül\_adı import \** yöntemiyle içe aktardığımız modüllerin nitelik ve fonksiyonlarını ise şöyle kullanıyoruz:

```
>>> fonksiyon
```

Mesela yukarıda bahsettiğimiz *aritmetik* modülünü örnek alalım:

```
>>> from aritmetik import *
>>> liste = [2, 3, 4]
>>> print carp(liste)
```

24

Gördüğünüz gibi, bu defa `aritmetik.carp(liste)` gibi bir komut vermedik. `carp()` fonksiyonunu doğrudan kullanabildik. Bu yöntem oldukça pratiktir. Programcıya aynı işi daha az kodla yapma imkânı sağlar. Ancak bu yöntemin bazı sakıncaları vardır. Bunlara biraz sonra değineceğiz.

### **from modül\_adı import falanca, filanca**

Bu yöntem ise bir modülden *falanca* ve *filanca* adlı fonksiyonları çağırmanızı sağlayacaktır. Yani bütün içeriği değil, bizim istediğimiz fonksiyonları içe aktarmakla yetinecektir. Örneğin:

```
>>> from aritmetik import carp
```

Bu şekilde *aritmetik* modülünün yalnızca `carp()` fonksiyonunu içe aktarmış olduk. Bu şekilde `carp()` fonksiyonuna erişebiliriz:

```
>>> liste = [2, 3, 4]
>>> print carp(liste)
```

24

Ama `pi_sayisi` niteliğine erişemeyiz. Çünkü biz burada sadece `carp()` fonksiyonunu içe aktardık. Eğer `pi_sayisi` niteliğine de erişebilmek istersek modülümüzü şu şekilde içe aktarmamız gerekir:

```
>>> from deneme import carp, pi_sayisi
```

Bu şekilde hem `carp()` fonksiyonunu, hem de `pi_sayisi` niteliğini içe aktarmış olduk.

### **import modül\_adı as yeni\_isim**

Diyelim ki herhangi bir sebepten, modülün adını programınız içinde doğrudan kullanmak istemiyorsunuz. O zaman bu yöntemi kullanarak modüle farklı bir ad verebilirsiniz:

```
>>> import aritmetik as arit
>>> liste = [2, 3, 4]
>>> arit.carp(liste)
```

Mesela içe aktaracağınız modül adı çok uzunsa ve her defasında bu uzun ismi yazmak size zor geliyorsa bu yöntemi kullanarak modül adını kısaltabilirsiniz. Ayrıca programınızda zaten *aritmetik* adlı başka bir nitelik veya fonksiyon varsa bu ikisinin birbirine karışmasını engellemek için de bu yöntemi kullanmayı tercih edebilirsiniz.

Peki bu yöntemlerden hangisini kullanmak daha iyidir? Eğer ne yaptığınızdan tam olarak emin değilseniz veya o modülle ilgili bir belgede farklı bir yöntem kullanmanız önerilmiyorsa, anlatılan birinci yöntemi kullanmak her zaman daha güvenlidir (`import modül_adı`). Çünkü bu şekilde bir fonksiyonun nereden geldiğini karıştırma ihtimaliniz ortadan kalkar. Mesela `aritmetik.carp(liste)` gibi bir komuta baktığınızda `carp()` fonksiyonunun *aritmetik* adlı bir modül içinde olduğunu anlayabilirsiniz. Ama sadece `carp(liste)` gibi bir komutla

karşılaştığınızda bu fonksiyonun program içinde mi yer aldığını, yoksa başka bir modülden mi içe aktarıldığını anlayamazsınız. Ayrıca mesela programınız içinde zaten *carp()* adlı bir fonksiyon varsa, *aritmetik* adlı modülden *carp()* fonksiyonunu aldığınızda isim çakışması nedeniyle hiç istemediğiniz sonuçlarla karşılaşabilirsiniz. Buna bir örnek verelim. Komut satırında şöyle bir kod yazın:

```
>>> pi_sayisi = 46
```

Şimdi *aritmetik* adlı modülü şu yöntemle içe aktarın:

```
>>> from aritmetik import *
```

Bakın bakalım *pi\_sayisi* değişkeninin değeri ne olmuş?

```
>>> print pi_sayisi
```

```
3.1428571428571428
```

Gördüğünüz gibi, *aritmetik* modülündeki *pi\_sayisi* niteliği sizin kendi programınızdaki *pi\_sayisi* değişkenini silip attı. Herhalde böyle bir şeyin başınıza gelmesini istemezsiniz. O yüzden içeriğini bilmediğiniz modülleri içe aktarırken `import modül_adı` yöntemini kullanmak sizi büyük baş ağrılarından kurtarabilir.

## 14.5 Geliştiricilerin Yazdığı Modüller

Buraya kadar kendi kendimize nasıl Python modülü yazacağımızı ve yazdığımız bu modülleri nasıl kullanacağımızı gördük. Ama modül çeşitlerinden bahsederken de söylediğimiz gibi, Python sadece kendi modüllerimizi yazmamıza izin vermez. Bir de hazır yazılmış modüller vardır. Python programlama dili içinde çok sayıda modül kullanıma hazır bir şekilde bizi bekler. Python modüllerinin listesine <http://docs.python.org/modindex.html> adresinden erişebilirsiniz. Biz bu bölümde, *os* adlı bir modül üzerinden, size geliştiricilerin yazdığı modülleri tanıtmaya çalışacağız.

### 14.5.1 *os* Modülü

*os* adlı modül Python'daki en önemli modüllerden biridir. Bu bölümün en başında yaptığımız modül tanımını dikkate alacak olursak, aslında *os* modülü, bilgisayarımızda bulunan *os.py* adlı bir Python programıdır. Peki biz bu *os.py* programını nereden bulabiliriz. GNU/Linux sistemlerinde bu modül çoğunlukla */usr/lib/python2.6/* dizini içinde bulunur. Windows sistemlerinde ise bu modülü bulmak için *C:/Python26/Lib* adlı dizinin içine bakabilirsiniz.

*os.py* dosyasını açıp içine baktığınızda, aslında bu dosyanın birtakım fonksiyon ve niteliklerden oluşan alelade bir Python programı olduğunu göreceksiniz. Bu dosyanın sizin yazdığınız Python programlarından hiç bir farkı yoktur. Kendi yazdığınız modüller içindeki fonksiyonları, modülü içe aktarmak suretiyle nasıl kullanıyorsanız, *os* modülünü de aynen öyle içe aktarıp bunun içindeki fonksiyon ve nitelikleri kullanacaksınız.

*os* modülü bize, işletim sistemleriyle ilgili işlemler yapma olanağı sunar. Bu modüle ilişkin resmi (İngilizce) belgelere <http://docs.python.org/library/os.html> adresinden erişebilirsiniz. Modülün kendi belgelerinde belirtildiğine göre, bu modülü kullanan programların farklı işletim sistemleri üzerinde çalışma şansı daha fazladır. Bunun neden böyle olduğunu biraz sonra daha iyi anlayacaksınız.

Bu modülü, tıpkı kendi yazdığımız modülleri içe aktarıyormuşuz gibi aktaracağız:

```
>>> import os
```

Gördüğünüz gibi, kullanım olarak kendi yazdığımız bir modülü nasıl içe aktarıyorsak *os* modülünü de aynen öyle içe aktarıyoruz. Neticede bu modülü siz de yazmış olablirdiniz. Dolayısıyla, içeriği dışında, bu modülün sizin yazdığınız herhangi bir Python programından (başka bir söyleyişle “Python modülünden”) hiç bir farkı yoktur. Tabii bu modülün sizin yazdığınız modülden önemli bir farkı, komut satırını hangi dizin altında açmış olursanız olun *os* modülünü içe aktarabilmenizdir. Yani bu modülü kullanabilmek için *os.py* dosyasının bulunduğu dizine gitmenize gerek yok. Bunun neden böyle olduğunu biraz sonra açıklayacağız. O yüzden bunu şimdilik dert etmeyin. Neyse, biz konumuza dönelim...

Burada en önemli konu, bu modülü içe aktarmaktır. Bu modülün içindeki herhangi bir fonksiyonu ya da niteliği kullanabilmek için öncelikle modülü içe aktarmalıyız. Eğer bu şekilde modülü “import” etmezsek, bu modülle ilgili kodlarımızı çalıştırmak istediğimizde Python bize bir hata mesajı gösterecektir.

Bu modülü programımız içine nasıl davet edeceğimizi öğrendiğimize göre, *os* modülü içindeki fonksiyonlardan ve niteliklerden söz edebiliriz. Öncelikle, isterseniz bu modül içinde neler var neler yok şöyle bir listeleyelim.

Python komut satırında “>>>” işaretinden hemen sonra:

```
>>> import os
```

komutuyla *os* modülünü içe aktarıyoruz. Daha sonra şu komutu veriyoruz:

```
>>> dir(os)
```

İsterseniz daha anlaşılır bir çıktı elde edebilmek için bu komutu şu şekilde de verebilirsiniz:

```
>>> for icerik in dir(os):  
...     print icerik
```

Gördüğünüz gibi, bu modül içinde bir yığın fonksiyon ve nitelik var. Şimdi biz bu fonksiyonlar ve niteliklerden önemli olanlarını incelemeye çalışalım.

## name Niteliği

*dir(os)* komutuyla *os* modülünün içeriğini incelediğinizde orada *name* adlı bir nitelik olduğunu göreceksiniz. Bu nitelik, kullandığınız işletim sisteminin ne olduğu hakkında bilgi verir.

Basit bir örnekle başlayalım:

```
>>> print os.name
```

Bu komutu hangi işletim sisteminde verdiğinizle bağlı olarak aldığınız çıktı da farklı olacaktır. Mesela ben bu komutu GNU/Linux işletim sistemi üzerinde verdiğim için ‘posix’ çıktısı aldım.

*os* modülünde işletim sistemi isimleri için öntanımlı olarak şu ifadeler bulunur:

- GNU/Linux için “posix”,
- Windows için “nt”, “dos”, “ce”
- Macintosh için “mac”
- OS/2 için “os2”
- Risc Os için “riscos”

Eğer mesela siz bu komutu Windows XP üzerinde verdiyseniz ‘nt’ çıktısı almış olmalısınız... Öğrendiğimiz bu bilgiyi kullanarak şöyle basit bir şey yazabiliriz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import os

if os.name == "posix":
    a = raw_input("Linus Torvalds'a mesajınızı yazın:")

if os.name == "nt":
    a = raw_input("Bill Gates'e mesajınızı yazın:")
```

Bu basit örnekte öncelikle os adlı modülü bütün içeriğiyle birlikte programımıza aktardık. Daha sonra bu modül içindeki name niteliğinden yararlanarak, kullanılan işletim sistemini sorguladık. Buna göre bu program çalıştırıldığında, eğer kullanılan işletim sistemi GNU/Linux ise, kullanıcıdan “Linus Torvalds’a mesajını yazması” istenecektir. Eğer kullanılan işletim sistemi Windows ise, “Bill Gates’e mesaj yazılması istenecektir. Aynı komutları şu şekilde de yazabiliriz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from os import name

if name == "posix":
    a = raw_input("Linus Torvalds'a mesajınızı yazın:")

if name == "nt":
    a = raw_input("Bill Gates'e mesajınızı yazın:")
```

Dikkat ederseniz burada from os import name komutuyla, os modülü içindeki name niteliğini aldık yalnızca. Ayrıca program içinde kullandığımız os.name ifadesini de name şekline dönüştürdük. Çünkü from os import name komutuyla yalnızca name niteliğini çektiğimiz, aslında os modülünü çekmediğimiz için, os.name yapısını kullanırsak Python bize “os” isminin tanımlanmadığını söyleyecektir.

Bu name niteliğinin ne kadar faydalı bir şey olduğunu tahmin edersiniz. Eğer yazdığınız bir programda name niteliğini kullanarak işletim sistemi sorgulaması yaparsanız, yazdığınız programın birden fazla işletim sistemi üzerinde çalışma imkânı olacaktır. Çünkü bu sayede programınızın, kullanılan işletim sistemine göre işlem yapmasını sağlayabilirsiniz.

## system() Fonksiyonu

Bu fonksiyon, kullandığımız işletim sistemine ait sistem komutlarını çalıştırmamızı sağlar. Peki “sistem komutu” ne demek? Sistem komutu, üzerinde çalıştığımız işletim sisteminin bir parçası olan ve bu işletim sistemine ilişkin işlemleri yerine getirmemizi sağlayan komutlardır. Örneğin GNU/Linux’ta ls bir sistem komutudur ve bu komutun görevi o anda içinde bulunduğunuz dizindeki dosya ve klasörleri listelemektir. Bu komutun Windows’taki karşılığı ise dir komutudur.

İşte buna benzer sistem komutlarını Python yardımıyla çalıştırabilmek için os modülünün bize sunduğu system() adlı fonksiyondan yararlanacağız. Örneğin:

```
>>> os.system("ls")
```

veya:



```
>>> os.system("dir")
```

Gördüğünüz gibi, `system()` komutunu kullanmak çok basit. Yapmamız gereken tek şey parantez içinde, çalıştırılmasını istediğimiz sistem komutunu belirtmek...

Bir örnek daha verelim. Diyelim ki GNU/Linux'taki Kwrite programını kullanarak bir dosya açmak istiyoruz:

```
>>> os.system("kwrite dosyaad1.txt")
```

Ancak bir dosyayı açtırırken, o dosyayı açacak programı doğrudan belirtmek pek doğru bir yol değildir. Çünkü mesela Kwrite her GNU/Linux kullanıcısının bilgisayarında kurulu olmayabilir. Kwrite yerine mesela Gedit olabilir... Bu yüzden Python'da herhangi bir dosya veya klasörü açtırmak için en güvenilir yol `xdg-open` adlı sistem komutundan yararlanmak olacaktır. Bunu şöyle kullanabiliriz:

```
>>> os.system("xdg-open dosyaad1.txt")
```

Burada kullandığımız `xdg-open` komutu bütün GNU/Linux sistemlerinde çalışan bir sistem komutudur. Bu komutun görevi herhangi bir dosya veya klasörü sistemdeki varsayılan uygulamayla açmaktır. Yani mesela bir sistemde metin dosyalarını açan öntanımlı program Kwrite ise dosya Kwrite ile açılacaktır. Eğer metin dosyalarını açan öntanımlı program başka bir şeyse (mesela Gedit), dosya o programla açılacaktır. `xdg-open` komutu özellikle bir dizini açtırmaya çalışırken çok işe yarar. Çünkü GNU/Linux sistemlerinde dizin görüntüleyici uygulamanın hangisi olduğunu kestirmek çok güçtür. Sistemde Konqueror, Dolphin, Thunar ve benzeri dizin görüntüleyicilerden hangisinin bulunduğuna emin olamazsınız. `xdg-open` komutu bu noktada size yardımcı olacaktır. Örneğin `/usr/share` dizinini görüntülemek için şöyle bir kod yazabilirsiniz:

```
>>> os.system("xdg-open /usr/share")
```

Böylece, acaba kullanıcının bilgisayarında hangi dizin görüntüleyici uygulama kurulu, diye endişe etmenize gerek kalmaz. `xdg-open` komutu, sistemde hangi görüntüleyici varsa `/usr/share` dizinini onunla açacaktır.

Yukarıda Kwrite ile yaptığımız işlemi bir de Windows'ta Notepad programıyla yapalım:

```
>>> os.system("notepad dosyaad1.txt")
```

Tıpkı GNU/Linux'ta olduğu gibi, Windows'ta da bir işlemi yapacak uygulamayı doğrudan belirtmek iyi bir fikir değildir. GNU/Linux'taki `xdg-open` yerine `os` modülünün `startfile()` adlı fonksiyonundan yararlanabilirsiniz. Örneğin:

```
# -*- coding: cp1254 -*-  
  
import os  
  
dosya = "falancadosya"  
  
os.startfile(dosya)
```

`startfile()` fonksiyonu GNU/Linux'ta bulunmaz. Bu fonksiyon yalnızca Windows'ta geçerlidir ve GNU/Linux'taki `xdg-open` komutunun yaptığı şeye benzer bir işlevi vardır.

Yukarıdaki bilgileri kullanarak örneğin şu basit programı yazabiliriz:

```
#!/usr/bin/env python  
# -*- coding: utf-8 -*-
```

```
import os

dosya = "falanca.txt"

print "%s adlı dosya açılıyor..." %dosya

if os.name == "nt":
    os.startfile(dosya)

elif os.name == "posix":
    os.system("xdg-open %s" %dosya)
```

Bu program oldukça küçük bir kod parçasından ibaret olmasına rağmen, bize `os` modülüyle ilgili çok önemli bilgiler veriyor. Gelin isterseniz yukarıdaki kod parçasının ne işler çevirdiğini adım adım inceleyelim:

1. İlk iki satırın ne iş yaptığını söylememize gerek yok. Artık siz bu satırların ne olduğunu adınız gibi biliyorsunuz.
2. Üçüncü satır da artık bizim için oldukça tanıdık bir koddan oluşuyor. Bu komutla `os` modülünü bütün içeriğiyle birlikte programımızın içine aktarıyoruz.
3. Sonraki satırda “falanca.txt” adlı bir dosya tanımladık. Bu değişkenin yerine, bilgisayarınızda bulunan herhangi bir dosyanın adını yazabilirsiniz.
4. `print "%s adlı dosya açılıyor..." %dosya` satırıyla kullanıcıya *falanca.txt* dosyasının açılmak üzere olduğuna dair bilgi veriyoruz.
5. Eğer kullanıcının işletim sistemi Windows XP ise (`if os.name == "nt":`) `os` modülünün `startfile()` adlı fonksiyonunu kullanarak dosyayı açıyoruz (`os.startfile(dosya)`)
6. Eğer kullanıcının işletim sistemi GNU/Linux ise (`if os.name == "posix":`) `os` modülünün `system()` adlı fonksiyonunu kullanarak dosyayı açıyoruz (`os.system("xdg-open %s" %dosya)`)

Dediğimiz gibi, `system()` fonksiyonu her türlü sistem komutunu çalıştırmanıza izin verir. Mesela Windows'ta Python yardımıyla Program Files dizinini görüntüleyelim:

```
>>> os.system("explorer C:\\Program Files")
```

Burada kullandığımız bölü işaretinin iki adet ters bölü (\\) olduğuna dikkat ediyoruz.

Aynı şekilde, Internet Explorer programı ile bir web sitesini açmak için de `system()` fonksiyonundan yararlanabilirsiniz:

```
>>> os.system("start iexplore.exe http://www.istihza.com")
```

Burada da Windows'un start adlı sistem komutunu kullanarak IE ile istihza.com sitesini görüntüledik.

## listdir() Fonksiyonu

Python yardımıyla bir dizin içindeki dosyaları görüntülemek istersek, `system()` fonksiyonundan yardım alabileceğimizi biliyoruz. Bunun için GNU/Linux kullanıcıları şunu yazabilir:

```
>>> os.system("ls")
```

Windows kullanıcıları ise şunu:

```
>>> os.system("dir")
```

Ancak bu işlemi Python'da çok daha kolay ve esnek bir biçimde yapmanın bir yolu daha var. `os` modülü içinde yer alan `listdir()` adlı fonksiyon bize bir dizin içindeki dosyaları veya klasörleri listeleme imkânı veriyor. Mesela `/usr/bin` dizini içindeki bütün dosyaları listelemek için şöyle bir şey yazabiliriz:

```
>>> import os

>>> a = os.listdir("/usr/bin")

>>> print a
```

Yukarıdaki örnekte her zamanki gibi, modülümüzü `import os` komutuyla programımıza aktardık ilk önce. Ardından kullanım kolaylığı açısından `listdir()` fonksiyonunu `a` adlı bir değişkene atadık. Örnekte `listdir()` fonksiyonunun nasıl kullanıldığını görüyorsunuz. Örneğimizde `/usr/bin` dizini altındaki dosya ve klasörleri listeliyoruz. Burada parantez içinde tırnak işaretlerini ve yatık çizgileri nasıl kullandığımıza dikkat edin. En son da `print a` komutuyla `/usr/bin` dizinin içeriğini liste olarak ekrana yazdırıyoruz. Çıktının tipinden anladığımız gibi, elimizde olan şey, öğeleri yan yana dizilmiş bir liste. Eğer biz dizin içeriğinin böyle yan yana değil de alt alta dizildiğinde daha şık görüneceğini düşünüyorsak, kodlarımızı şu biçime sokabiliriz:

```
import os

a = os.listdir("/usr/bin")

for dosyalar in a:
    print dosyalar
```

Eğer dosyalarımıza numara da vermek istersek şöyle bir şey yazabiliriz:

```
import os

a = os.listdir("/usr/bin")
c = 0

for dosyalar in a:
    if c < len(a):
        c = c+1
        print c, dosyalar
```

Hatta daha önce öğrendiğimiz `enumerate()` fonksiyonunu kullanarak bu işlemi çok daha kısa bir yoldan halledebilirsiniz:

```
import os

a = os.listdir("/usr/bin")

for numara, dosya in enumerate(a, 1):
    print numara, dosya
```

Eğer amacınız, o anda içinde bulunduğunuz dizindeki dosyaları listelemekse `listdir()` fonksiyonunu şu şekilde kullanabilirsiniz:

```
>>> os.listdir(".")
```

Buradaki `"."` argümanı, o anda içinde bulunulan dizini temsil eder. Eğer bir üst dizinin içeriğini listelemek isterseniz tek nokta yerine iki nokta işaretini kullanabilirsiniz:

```
>>> os.listdir(".")
```

Bu komut o anda içinde bulunduğunuz dizine göre bir üst dizinin içeriğini ekrana liste olarak verecektir. Yani mesela o anda `/usr/local/bin` dizini içindeyseniz yukarıdaki komut `/usr/local` dizininin içeriğini listeleyecektir.

Bu fonksiyonu ve daha önce öğrendiğimiz `system()` fonksiyonunu kullanarak, bir dizin içindeki bütün dosyaları açan ve bu sebeple bilgisayarınızın bir süre yanıt vermemesine bile yol açabilecek korkunç bir program yazabilirsiniz! (Denemeyin!)

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

if os.name == "posix":
    for i in os.listdir("/usr/bin"):
        os.system("xdg-open %s%s" %("/usr/bin/", i))

elif os.name == "nt":
    for i in os.listdir("C:\\Program Files"):
        os.startfile("%s%s" %("C:\\Program Files\\", i))
```

Sırada yine önemli bir fonksiyon var...

## getcwd() Fonksiyonu

`os` modülü içinde yer alan bu fonksiyon bize o anda hangi dizin içinde bulunduğumuza dair bilgi verir. İsterseniz bu fonksiyonun tam olarak ne işe yaradığını bir örnek üzerinde görelim:

```
>>> os.getcwd()
```

Gördüğünüz gibi bu komut bize o anda hangi dizin içinde bulunduğumuzu söylüyor. Bu arada İngilizce bilenler için söyleyelim, buradaki “*cwd*”nin açılımı “*current working directory*”. Yani kabaca “mevcut çalışma dizini”... Daha açık ifade etmek gerekirse: “O anda içinde bulunduğumuz dizin”. Şöyle bir örnek vererek konuyu biraz açalım:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import os

mevcut_dizin = os.getcwd()

if mevcut_dizin == "/home/istihza/Desktop":
    for i in os.listdir(mevcut_dizin):
        print i
else:
    print ("Bu program yalnızca /home/istihza/Desktop "
           "dizininin içeriğini gösterebilir!")
```

Yukarıdaki örnekte öncelikle `os` modülünü içe aktardık. Daha sonra `mevcut_dizin` adında bir değişken tanımlayıp `getcwd()` fonksiyonunun kendisini bu değişkenin değeri olarak atadık. Ardından, “eğer `mevcut_dizin` `/home/istihza/Desktop` ise bu dizin içindeki dosyaları bize listele ve sonucu ekrana yazdır, yok eğer `mevcut_dizin` `/home/istihza/Desktop` değil ise, ‘bu program yalnızca `/home/istihza/Desktop` dizininin içeriğini gösterebilir,’ cümlesini göster” dedik. Burada dikkat ederseniz `if` deyiminden sonra `for` döngüsünü kullandık. Bu işlemi, ekran çıktısı daha düzgün olsun diye yaptık. Eğer böyle bir kaygımız olmasaydı,

```
if mevcut_dizin == "/home/istihza/Desktop":
```

satırının hemen altına:

```
print mevcut_dizin
```

yazıp işi bitirirdik.

Biz burada `getcwd()` fonksiyonu için basit örnekler verdik, ama eminim siz yaratıcılığınızla çok daha farklı ve kullanışlı kodlar yazabilirsiniz. Mesela kendi yazdığınız bir modülü içe aktarmak istediğinizde neden hata verdiğini anlamak için bu fonksiyondan yararlanabilirsiniz. Eğer bu fonksiyonun verdiği çıktı, içe aktarmaya çalıştığınız modülün bulunduğu dizinden farklıysa o modülü boşuna içe aktarmaya çalışıyorsunuz demektir!

Şimdi de `os` modülü içindeki başka bir fonksiyona değinelim.

### chdir() Fonksiyonu

Bu fonksiyon yardımıyla içinde bulunduğumuz dizini değiştirebiliriz. Diyelim ki o anda `/usr/share/apps` dizini içindeyiz. Eğer bir üst dizine, yani `/usr/share/` dizinine geçmek istiyorsak, şu komutu verebiliriz:

```
>>> import os
>>> os.chdir(os.pardir)
>>> print os.getcwd()
```

Hatırlarsanız bir üst dizini temsil etmek için `".."` işaretinden yararlanıyorduk. İşte `pardir` de bu işaretin yaptığı işi yapar. Buradaki `pardir` niteliği, İngilizce *"parent directory"* (bir üst dizin) ifadesinin kısaltması oluyor. `pardir` niteliği dışında, bir de `curdir` niteliği vardır. Bu sabiti kullanarak "mevcut dizin" üzerinde işlemler yapabiliriz. Tıpkı `".."` işaretinde olduğu gibi....:

```
>>> import os
>>> os.listdir(os.curdir)
```

Gördüğünüz gibi bu `curdir` niteliği `getcwd()` fonksiyonuna benziyor. Bunun dışında, istersek gitmek istediğimiz dizini kendimiz elle de belirtebiliriz:

```
>>> import os
>>> os.chdir("/var/tmp")
```

Böylece doğrudan `/var/tmp` dizinine ulaşmış olduk. Yukarıdaki komutun ardından şu komutu vererseniz bu durumu teyit edebilirsiniz:

```
>>> os.getwd()
'/var/tmp'
```

### mkdir() ve makedirs() Fonksiyonları

Bu iki fonksiyon yardımıyla dizin veya dizinler oluşturacağız. Mesela:

```
>>> import os
>>> os.mkdir("/test")
```

Bu kod / dizini altında “test” adlı boş bir klasör oluşturacaktır. Eğer bu kodu şu şekilde yazarsak, mevcut çalışma dizini içinde yeni bir dizin oluşacaktır:

```
>>> import os
>>> os.mkdir("test")
```

Yani, mesela mevcut çalışma dizini masaüstü ise bu “test” adlı dizin masaüstünde oluşacaktır. İsterseniz bu kodları şu şekle getirerek yeni oluşturulan dizinin nerede olduğunu da görebilirsiniz:

```
>>> import os
>>> print os.getcwd()
>>> os.mkdir("test")
```

Bundan başka, eğer isterseniz mevcut bir dizin yapısı içinde başka bir dizin de oluşturabilirsiniz. Yani mesela `/home/kullanıcı_adınız/` dizini içinde “deneme” adlı boş bir dizin oluşturabilirsiniz:

```
>>> import os
>>> os.mkdir("/home/istihza/deneme")
```

Peki diyelim ki iç içe birkaç tane yeni klasör oluşturmak istiyoruz. Yani mesela `/home/kullanıcı_adınız` dizini altında yeni bir “Programlar” dizini, onun altında da “Python” adlı yeni başka bir dizin daha oluşturmak istiyoruz.

Hemen deneyelim:

```
>>> import os
>>> os.mkdir("/home/istihza/Programlar/Python")
```

Ne oldu? Şöyle bir hata çıktısı elde ettik, değil mi?

```
>>> os.mkdir("/home/istihza/Programlar/Python")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: [Errno 2] No such file or directory:
'/home/istihza/Programlar/Python'
```

Demek ki bu şekilde çoklu dizin oluşturamıyoruz. İşte bu amaç için elimizde `makedirs()` fonksiyonu var. Hemen deneyelim yine:

```
>>> import os
>>> os.makedirs("/home/istihza/Programlar/Python")
```

Gördüğümüz gibi, `/home/kullanıcı_adınız/` dizini altında yeni bir “Programlar” dizini ve onun altında da yeni bir “Python” dizini oluştu. Buradan çıkan sonuç, demek ki `mkdir()` fonksiyonu bize yalnızca bir adet dizin oluşturma izni veriyor. Eğer biz birden fazla, yani çoklu yeni dizin oluşturmak istiyorsak `makedirs()` fonksiyonunu kullanmamız gerekiyor.

Küçük bir örnek daha verip bu bahsi kapatalım:

```
>>> import os
>>> print os.getcwd()
>>> os.makedirs("test/test1/test2/test3")
```

Tahmin ettiğiniz gibi bu kod mevcut çalışma dizini altında iç içe “test”, “test1”, “test2” ve “test3” adlı dizinler oluşturdu. Eğer “test” ifadesinin soluna “/” işaretini eklerseniz, bu boş dizinler kök dizini altında oluşacaktır.

### **rmkdir() ve removedirs() fonksiyonları**

Bu fonksiyonlar bize mevcut dizinleri silme olanağı tanıyor. Yalnız, burada hemen bir uyarı yapalım: Bu fonksiyonları çok dikkatli kullanmamız gerekiyor. Ne yaptığınızdan, neyi sildiğinizden emin değilseniz bu fonksiyonları kullanmayın! Çünkü Python bu komutu verdiğinizde tek bir soru bile sormadan silecektir belirttiğiniz dizini. Gerçi, bu komutlar yalnızca içi boş dizinleri silecektir, ama yine de uyaralım...

Hemen bir örnek verelim. Diyelim ki mevcut çalışma dizinimiz olan masaüstünde “TEST” adlı boş bir dizin var ve biz bu dizini silmek istiyoruz:

```
>>> import os
>>> os.rmdir("TEST")
```

Böylece “TEST” dizini silindi.

Bir de şu örneğe bakın:

```
>>> import os
>>> os.rmdir("/home/istihza/TEST")
```

Bu kod ise /home/kullanıcı\_adı dizini altındaki boş “TEST” dizinini silecektir.

Tıpkı *mkdir()* ve *makedirs()* fonksiyonlarında olduğu gibi, iç içe birden fazla boş dizini silmek istediğimizde ise *removedirs()* fonksiyonundan yararlanıyoruz:

```
>>> import os
>>> os.removedirs("test1/test2")
```

Yine hatırlatmakta fayda var: Neyi sildiğinize mutlaka dikkat edin...

Python’da dizinleri nasıl yöneteceğimizi, nasıl dizin oluşturup sileceğimizi basitçe gördük. Şimdi de bu “dizinleri yönetme” işini biraz irdелейelim. Şimdiye kadar hep bir dizin, onun altında başka bir dizin, onun altında da başka bir dizini nasıl oluşturacağımızı çalıştık. Peki, aynı dizin altında birden fazla dizin oluşturmak istersek ne yapacağız? Bu işlemi çok kolay bir şekilde şöyle yapabiliriz:

```
>>> import os
>>> os.makedirs("test1/test2")
>>> os.makedirs("test1/test3")
```

Bu kodlar mevcut çalışma dizini altında “test1” adlı bir dizin ile bunun altında “test2” ve “test3” adlı başka iki adet dizin daha oluşturacaktır. Peki, bu “test1”, “test2” ve “test3” ifadelerinin sabit değil de değişken olmasını istersek ne yapacağız. Şöyle bir şey deneyelim:

```
>>> import os
>>> test1 = "Belgelerim"
>>> test2 = "Hesaplamalar"
>>> test3 = "Resimler"
>>> os.makedirs(test1/test2)
>>> os.makedirs(test1/test3)
```

Bu kodları çalıştırdığımızda Python bize şöyle bir şey söyler:

```
>>> os.makedirs(test1/test3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s)
for /: 'str' and 'str'
```

Peki, neden böyle oldu ve bu hata ne anlama geliyor?

Kod yazarken bazı durumlarda “/” işareti programcıları sıkıntıya sokabilir. Çünkü bu işaret Python’da hem “bölme” işleci hem de “dizin ayracı” olarak kullanılıyor. Biraz önce yazdığımız kodda Python bu işareti “dizin ayracı” olarak değil “bölme işleci” olarak algıladı ve sanki “test1” ifadesini “test2” ifadesine bölmek istiyormuşuz gibi davrandı bize. Ayrıca kullandığımız *os.makedirs()* fonksiyonunu da gördüğü için ne yapmaya çalıştığımızı anlayamadı ve kafası karıştı. Peki, bu meseleyi nasıl halledeceğiz?

Bu meseleyi halletmek için kullanmamız gereken başka bir nitelik var Python’da...

## sep niteliği

Bu nitelik, işletim sistemlerinin dizin ayraçları hakkında bize bilgi veriyor. Eğer yazdığımız bir programın farklı işletim sistemleri üzerinde çalışmasını istiyorsak bu fonksiyon epey işimize yarayacaktır. Çünkü her işletim sisteminin dizin ayracı birbiriyle aynı değil. Bunu şu örnekle gösterebiliriz: Hemen bir Python komut satırı açıp şu komutları verelim:

```
>>> import os
>>> os.sep
'/'
```

Bu komutu GNU/Linux’ta verdiğimiz için komutun çıktısı “/” şeklinde oldu. Eğer aynı komutu Windows’ta verirsek sonuç şöyle olacaktır:

```
>>> import os
>>> os.sep
'\\'
```

Peki bu sep niteliği ne işe yarar? Yazdığımız kodlarda doğrudan dizin ayracı vermek yerine bu niteliği kullanırsak, programımızı farklı işletim sistemlerinde çalıştırırken, sistemin kendine özgü dizin ayracının kullanılmasını sağlamış oluruz. Yani mesela:



```
>>> import os
>>> os.makedirs("test/test2")
```

komutu yerine;

```
>>> import os
>>> os.makedirs("test" + os.sep + "test2")
```

komutunu kullanırsak programımızı farklı işletim sistemlerinde çalıştırırken herhangi bir aksaklık olmasını önlemiş oluruz. Çünkü burada sep niteliği, ilgili işletim sistemi hangisiyse ona ait olan dizin ayracının otomatik olarak yerleştirilmesini sağlayacaktır.

Bu sep niteliği ayrıca dizin adlarını “değişken” yapmak istediğimizde de bize yardımcı olacaktır. Hatırlarsanız yukarıda şöyle bir kod yazmıştık:

```
>>> import os
>>> test1 = "Belgelerim"
>>> test2 = "Hesaplamalar"
>>> test3 = "Resimler"
>>> os.makedirs(test1/test2)
>>> os.makedirs(test1/test3)
```

Yine hatırlarsanız bu kodu çalıştırdığımızda Python hata vermişti. Çünkü Python burada “/” işaretini bölme işleci olarak algılamıştı. İşte bu hatayı almamak için sep niteliğinden faydalanabiliriz. Şöyle ki:

```
>>> import os
>>> test1 = "Belgelerim"
>>> test2 = "Hesaplamalar"
>>> test3 = "Resimler"
>>> os.makedirs(test1)
>>> os.makedirs(os.sep.join([test1, test2]))
>>> os.makedirs(os.sep.join([test1, test3]))
```

Dikkat ederseniz, burada sep niteliğini *join()* adlı bir fonksiyon ile birlikte kullandık. (*join()* fonksiyonunu birkaç ders sonra daha ayrıntılı bir şekilde inceleyeceğiz). Yukarıdaki kod sayesinde doğrudan “/” işaretine bulaşmadan, başımızı derde sokmadan işimizi halledebiliriz. Ayrıca burada parantez ve köşeli parantezlerin nasıl kullanıldığına da dikkat etmemiz gerekiyor.

Yukarıda “test1”, “test2” ve “test3” değişkenlerinin adlarını doğrudan kod içinde verdik. Tabii eğer istersek *raw\_input()* fonksiyonuyla dizin adlarını kullanıcıya seçtirebileceğimiz gibi, şöyle bir şey de yapabiliriz:

```
import os

def dizinler(test1, test2, test3):
    os.makedirs(test1)
    os.makedirs(os.sep.join([test1, test2]))
    os.makedirs(os.sep.join([test1, test3]))
```

Dikkat ederseniz, burada öncelikle `os` modülünü çağırıyoruz. Daha sonra `dizinler()` adlı bir fonksiyon oluşturup parametre olarak “test1”, “test2” ve “test3” adlı değişkenler belirliyoruz. Ardından `os.makedirs(test1)` komutuyla “test1” adlı bir dizin oluşturuyoruz. Tabii bu “test1” bir değişken olduğu için adını daha sonradan biz belirleyeceğiz. Alttaki satırda ise `os.makedirs()` ve `os.sep.join()` komutları yardımıyla, bir önceki satırda oluşturduğumuz “test1” adlı dizinin içinde “test2” adlı bir dizin daha oluşturuyoruz. Burada `os.sep.join()` fonksiyonu “/” işaretiyle uğraşmadan dizinleri birleştirme imkânı sağlıyor bize. Hemen alttaki satırda da benzer bir işlem yapıp kodlarımızı bitiriyoruz. Böylece bir fonksiyon tanımlamış olduk. Şimdi bu dosyayı `deneme.py` adıyla masaüstüne kaydedelim. Böylelikle kendimize bir modül yapmış olduk. Şimdi Python komut satırını açalım ve şu komutları verelim:

```
>>> import deneme

>>> deneme.dizinler("Belgelerim", "Videolar", "Resimler")
```

Burada öncelikle `import deneme` satırıyla `deneme` adlı modülümüzü çağırdık. Daha sonra `deneme.dizinler` satırıyla bu modül içindeki `dizinler()` adlı fonksiyonu çağırdık. Böylelikle masaüstünde “Belgelerim” adlı bir klasörün içinde “Videolar” ve “Resimler” adlı iki klasör oluşturmuş olduk. Bu `os.sep.join()` ifadesi ile ilgili son bir şey daha söyleyip bu konuya bir nokta koyalım.

Şimdi Python komut satırını açarak şu kodları yazalım:

```
>>> import os

>>> os.sep.join(["Dizin1", "Dizin2"])
```

ENTER tuşuna bastığımızda, bu komutların çıktısı şöyle olur:

```
'Dizin1/Dizin2'
```

Aynı kodları Windows üzerinde verirsek de şu çıktıyı alırız:

```
'Dizin1\\Dizin2'
```

Gördüğünüz gibi farklı platformlar üzerinde, `os.sep` çıktısı birbirinden farklı oluyor. Bu örnek, `sep` niteliğinin, yazdığımız programların “taşınabilirliği” (*portability*), yani “farklı işletim sistemleri üzerinde çalışabilme kabiliyeti” açısından ne kadar önemli olabileceğini gösteriyor.

## 14.6 Üçüncü Şahısların Yazdığı Modüller

Buraya kadar kendi yazdığımız modülleri ve geliştiriciler tarafından yazılıp dilin içine entegre edilen modülleri ayrı ayrı gördük. Bunların dışında bir de üçüncü şahıslar tarafından yazılan modüller vardır. Bu modüller, yazarları tarafından genellikle internet üzerinden erişime sunulmuştur. Dolayısıyla bu modülleri, yazarlarının internet sitelerinden indirip bilgisayarınıza kurabilir, bu modülleri de herhangi bir Python modülü gibi kullanabilirsiniz. Piyasada pek çok üçüncü şahıs modülü bulunur. Bu modüller sayesinde, çok zor bazı işleri çok kolay bir şekilde

halledebiliriz. Biz bu bölümde *pyPdf* adlı bir örnek modül üzerinden, üçüncü şahıs modüllerini tanıtmaya çalışacağız.

Ancak bu noktada küçük bir uyarı yapalım. Aşağıda inceleyeceğimiz PyPdf modülünü anlatırken, henüz öğrenmediğimiz bazı şeylerle de karşılaşacaksınız. Anlayamadığınız noktalar olursa bunlara çok fazla takılmayın. Zira bütün o ayrıntıları sonraki derslerimizde inceleyeceğiz. Bizim burada amacımız, Python’da üçüncü şahıs modüllerinin neye benzediği ve bunların nasıl kullanılacağı hakkında yalnızca bir fikir vermekten ibarettir.

## 14.6.1 pyPdf Modülü

Bu bölümde inceleyeceğimiz örnek modülün adı *pyPdf*. Bu modül, Python’da pdf dosyaları ile ilgili temel işlemleri yapmamızı sağlıyor. Mathieu Fenniak tarafından yazılmış olan bu üçüncü şahıs modülünün anasayfasına <http://pybrary.net/pyPdf/> adresinden ulaşabilirsiniz.

*pyPdf* modülü bir üçüncü şahıs modülü olduğundan, bunu kullanabilmek için öncelikle bu modülü bilgisayarımıza kurmamız gerekiyor. Eğer bu modülü kurmazsak elbette kullanmamız mümkün olmaz. Gelin bir deneme yapalım:

```
>>> import pyPdf
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named pyPdf
```

Gördüğünüz gibi, bilgisayarımızda *pyPdf* modülü olmadığı için `import pyPdf` komutu hata veriyor. O yüzden öncelikle bu modülü bilgisayarımıza kuracağız.

Bu modül **GNU/Linux** dağıtımlarının paket depolarında bulunur. Eğer Ubuntu kullanıyorsanız, *pyPdf* modülünü şu komut yardımıyla sisteminize kurabilirsiniz:

```
sudo apt-get install python-pypdf
```

Eğer Ubuntu dışında bir GNU/Linux dağıtımı kullanıyorsanız, paket yöneticiniz yardımıyla “pypdf” şeklinde bir arama yapmanızı öneririm.

Kullandıkları dağıtımın paket depolarında *pyPdf* modülünü bulamayan arkadaşlarım şu adresten ilgili *tar.gz* dosyasını indirerek kaynaktan kurulum yapmayı tercih edebilirler: <http://pybrary.net/pyPdf/>

Bu sıkıştırılmış dosyayı bilgisayarımıza indirdikten sonra dosyayı açıp, orada şu komutu veriyoruz:

```
sudo python setup.py install
```

Eğer herhangi bir aksilik olmadıysa *pyPdf* modülü bu komutun ardından sistemimize kurulacaktır.

**Windows** kullanıcıları ise şu adresten *.exe* dosyasını indirip çift tıklayarak modülü bilgisayarlarına kurabilir: <http://pybrary.net/pyPdf/>

Gelin isterseniz biraz önce içe aktarırken hata aldığımız komutu tekrar vererek *pyPdf* modülünü doğru bir şekilde kurup kuramadığımızı denetleyelim:

```
>>> import pyPdf
```

Bu komutu verdiğinizde şöyle bir uyarı mesajı almış olabilirsiniz:

```
>>> import pyPdf
DeprecationWarning: the sets module is deprecated
from sets import ImmutableSet
```

*pyPdf* modülünün uyumlu olduğu son Python sürümü 2.5'tir. Bu modül Python'un 2.6 sürümünde de çalışıyor, ancak program çıktısında bazı uyarılar alıyoruz. Bu uyarılar, Python'un 3.x sürümünde kullanımdan kaldırılacak özelliklere ilişkindir. Ancak bu uyarılara rağmen bu modül Python 2.6 sürümünde de kullanılabilir. Yukarıdaki uyarı mesajı modülümüzün çalışmasını engellemez. *pyPdf* modülünün Python3.x sürümlerine uyumlu hale getirilmesi için çalışmalar da sürdürülmektedir.

Bu arada *pyPdf* modülünü içe aktarırken büyük-küçük harfe dikkat etmelisiniz...

*pyPdf* modülünü kurduğumuza göre artık bu modülü incelemeye başlayabiliriz.

## pdf Bilgilerine Ulaşmak

Başta da söylediğimiz gibi *pyPdf* modülünü kullanarak pdf belgelerine ait bazı bilgileri toplayabilirsiniz. Bu bilgiler neler olabilir? Mesela belgenin başlığı, belgenin yazarı, belgenin hangi yazılım ile oluşturulduğu, vb... Birkaç örnek verelim.

Öncelikle *pyPdf* modülünden bazı fonksiyonları içe aktarıyoruz:

```
from pyPdf import PdfFileWriter, PdfFileReader
```

`import` komutunu nasıl verdiğimizize dikkat edin. Daha önce modüllerin içe aktarılma yöntemlerini incelerken bu biçimi öğrendiğimizi hatırlıyor olmalısınız. *pyPdf* modülünün tamamına değil de, bunun içindeki sadece iki fonksiyona ihtiyaç duyduğumuz için yalnızca o iki fonksiyonu içe aktarıyoruz.

Şimdi de okuyacağımız pdf dosyasını tanımlıyoruz:

```
belge = "diveintopython.pdf"
```

Ben örnek olarak kendime *diveintopython.pdf* adlı bir pdf dosyası seçtim. Siz elbette başka bir dosya ile çalışabilirsiniz.

Yukarıda tanımladığımız belge adlı pdf dosyasını okumak üzere açıyoruz:

```
kaynak = PdfFileReader(open(belge, "rb"))
```

Burada belge adlı pdf dosyasını "rb" kipinde açtığımıza dikkat edin. pdf dosyaları ikili (*binary*) düzende oldukları için, bunları okurken "rb" kipini kullanmamız gerekir.

Daha önce de dediğimiz gibi, bu konuyu anlatırken henüz yabancı olduğumuz kavramlarla karşılaşyoruz. Ama bunları çok fazla dert edinmeyin. Siz şimdilik sadece üçüncü şahıs modüllerin neye benzediği üzerine yoğunlaşmaya çalışın. Neyse... Biz incelememize devam edelim.

Şimdi de *pyPdf* modülü içinde bulunan *getDocumentInfo()* fonksiyonunun `title` adlı niteliğini kullanarak, belge adlı pdf dosyamızın başlığını alalım:

```
print (u"%s adlı dosyanın başlığı şudur: %s"
      %(belge, kaynak.getDocumentInfo().title))
```

Burada, Türkçe karakterlerin düzgün görüntülenememesi gibi bir sorunla karşılaşmamak için, "u" harfini kullanarak karakter dizimizi "unicode" haline getiriyoruz. "unicode" kavramından birkaç ders sonra daha ayrıntılı olarak bahsedeceğiz.

İsterseniz yazdığımız kodları topluca görelim:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from pyPdf import PdfFileWriter, PdfFileReader

belge = "diveintopython.pdf"

kaynak = PdfFileReader(open(belge, "rb"))

print (u"%s adlı dosyanın başlığı şudur: %s"
      %(belge, kaynak.getDocumentInfo().title))
```

Daha önce de dediğimiz gibi, eğer sisteminizdeki Python sürümü 2.6 ise, yukarıdaki kodları çalıştırdığınızda şuna benzer uyarılar alabilirsiniz:

```
istihza@istihza:~/Desktop$ python deneme.py
DeprecationWarning: the sets module is deprecated
  from sets import ImmutableSet
```

Bu uyarılar programımızın çalışmasını engellemez. Çünkü bunlar birer hata değil, sadece uyarıdır. Bu uyarıya göre, pyPdf kütüphanesi içinde kullanılmış olan bazı modüller, Python'un mevcut sürümünde çalışıyor olsa bile, bir üst sürümde artık kullanılmayacak. Eğer siz bu uyarıları almak istemiyorsanız, programımızı şöyle yazabilirsiniz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import warnings

warnings.simplefilter("ignore", DeprecationWarning)

from pyPdf import PdfFileWriter, PdfFileReader

belge = "diveintopython.pdf"

kaynak = PdfFileReader(open(belge, "rb"))

print (u"%s adlı dosyanın başlığı şudur: %s"
      %(belge, kaynak.getDocumentInfo().title))
```

Burada *warnings* adlı bir başka modülü içe aktardığımıza dikkat edin. Bu modül geliştiricilerin yazdığı modüllerden biridir. Dolayısıyla bilgisayarımıza herhangi bir program kurmaya gerek kalmadan doğrudan programlarımız içinde kullanılabilir. Bu modülün *simplefilter()* adlı fonksiyonunu kullanarak "DeprecationWarning" olarak ifade edilen, "tedavülden kalkacak özellikler" hakkındaki uyarıları etkisizleştiriyoruz. Bu kodları çalıştırdığımız zaman şöyle bir çıktı elde edeceğiz:

```
diveintopython.pdf adlı dosyanın başlığı şudur: Dive Into Python
```

Her pdf belgesinin başlığı olmayabilir. Dolayısıyla eğer sizin çalıştığınız belgenin bir başlığı yoksa, başlık yerine None değerini alacaksınız.

*pyPdf* modülü yardımıyla bir belgenin başlığı dışında, yazarının kim olduğunu da öğrenebiliriz. Bunun için *title* yerine *author* niteliğini kullanıyoruz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

```
import warnings

warnings.simplefilter("ignore", DeprecationWarning)

from pyPdf import PdfFileWriter, PdfFileReader

belge = "falanca.pdf"

kaynak = PdfFileReader(open(belge, "rb"))

print (u"%s adlı dosyanın yazarı şudur: %s"
      %(belge, kaynak.getDocumentInfo().author))
```

Tıpkı title niteliğinde olduğu gibi, eğer pdf belgesinde yazar bilgisi geçmiyorsa, None değerini alırız.

Bir pdf belgesinin hangi yazılım ile oluşturulduğunu öğrenmek için ise creator niteliğini kullanıyoruz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import warnings

warnings.simplefilter("ignore", DeprecationWarning)

from pyPdf import PdfFileWriter, PdfFileReader

belge = "falanca.pdf"

kaynak = PdfFileReader(open(belge, "rb"))

print (u"%s adlı dosya şu yazılımla oluşturulmuştur: %s"
      %(belge, kaynak.getDocumentInfo().creator))
```

Bu kodları çalıştırdığımızda şuna benzer çıktılar alabiliriz:

Scribus 1.3.3.12, Writer, Adobe InDesign CS3 (5.0.4), vb...

Bu çıktılar, elimizdeki pdf belgesinin hangi yazılımın hangi sürümü kullanılarak oluşturulduğunu gösterir. Eğer belge başka bir formattan pdf formatına dönüştürülmüşse, özgün formatı oluşturmak için kullanılan yazılım görünecektir. Mesela bir belge önce OpenOffice ile yazılıp daha sonra OpenOffice kullanılarak pdf formatına dönüştürülmüşse çıktıda OpenOffice görünecektir. (OpenOffice çıktıda “Writer” olarak görünür.)

creator niteliğine benzer bir şekilde, pdf belgesini oluşturan yazılımı görüntüleyen bir başka metod da producer adlı niteliktir. Bu niteliği şu şekilde kullanıyoruz (Sadece ilgili kod parçasını gösteriyoruz)

```
kaynak.getDocumentInfo().producer
```

Örneğin, creator niteliği ile “Scribus 1.3.3.12” çıktısını almışsak, producer niteliği bize şu çıktıyı verebilir:

Scribus PDF Library 1.3.3.12

Veya creator niteliği bize Adobe InDesign CS3 (5.0.4) çıktısını veriyse, producer niteliği şuna benzer bir çıktı verebilir:

Adobe PDF Library 8.0

Ya da creator niteliği ile “Writer” çıktısı elde etmişsek, producer niteliğiyle şunu elde edebiliriz:

OpenOffice.org 2.3

Daha önce de belirttiğimiz gibi, her pdf dosyası yukarıdaki özelliklerin hepsini barındırmayabilir. Yani mesela her pdf belgesinin bir başlığı (title) olmayabilir... Böyle bir pdf belgesine title niteliğini uyguladığımızda None çıktısı alırız.

*getDocumentInfo()* fonksiyonunun en önemli metotlarını gördüğümüze göre, *pyPdf* modülünün başka bir özelliğini incelemeye başlayabiliriz.

## pyPdf ile pdf Belgelerinden Sayfa Almak

*pyPdf* modülünü kullanarak bir pdf belgesinden istediğimiz sayfaları alabiliriz. Örneğin, eğer bir pdf belgesinin ilk sayfasını almak istiyorsak şöyle bir işlem yapmamız gerekir:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from pyPdf import PdfFileReader, PdfFileWriter

kaynak = PdfFileReader(open("falanca.pdf", "rb"))
nesne = PdfFileWriter()
hedef = open("hedef.pdf", "wb")

nesne.addPage(kaynak.getPage(0))

nesne.write(hedef)
hedef.close()
```

Daha önce bahsettiğimiz gibi, bu kodları çalıştırdığınızda bazı zararsız uyarılar alacaksınız. İsterseniz yukarıda gösterdiğimiz şekilde bu uyarıları kapatabilirsiniz. Yukarıdaki programı bir dosyaya kaydedip çalıştırdığınızda, çalışma dizininiz içinde *hedef.pdf* adında bir pdf belgesi oluşacak. Bu yeni pdf belgesini açıp baktığınızda, içeriğinin *falanca.pdf* adlı belgenin ilk sayfası olduğunu göreceksiniz. Bu demek oluyor ki, yukarıdaki kodlar yardımıyla bir pdf dosyasının ilk sayfasını alıp başka bir pdf dosyası oluşturabiliyoruz...

Gelin isterseniz bu kodları biraz açıklayalım:

Burada öncelikle *pyPdf* modülünden, pdf dosyasını okumamıza yarayan *PdfFileReader()* fonksiyonunu ve pdf dosyasından okuduğumuz parçaları yazmamızı sağlayan *PdfFileWriter()* fonksiyonunu içe aktarıyoruz. Daha sonra, `kaynak = PdfFileReader(open("falanca.pdf", "rb"))` satırıyla kaynak dosyamızı tanımlıyoruz. Burada okuyacağımız kaynak pdf dosyamızın adı *falanca.pdf*. Bu dosyayı “rb” kipiyle açıyoruz. (Bu kiplerin ne demek olduğunu bir sonraki bölümde öğreneceğiz) Ardından, *falanca.pdf* dosyasından okuduğumuz verileri hedef dosyaya yazmadan önce, *PdfFileWriter()* fonksiyonunu kullanarak, yazacağımız verileri bir “*pyPdf.pdf.PdfFileWriter* nesnesi” haline getiriyoruz. Eğer yukarıdaki kodları şu şekilde yazacak olursanız:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import warnings

warnings.simplefilter("ignore", DeprecationWarning)

from pyPdf import PdfFileReader, PdfFileWriter
```

```
kaynak = PdfFileReader(open("falanca.pdf", "rb"))
nesne = PdfFileWriter()
print nesne
```

Şöyle bir çıktı alırsınız:

```
<pyPdf.pdf.PdfFileWriter object at 0xb7cdea0c>
```

Demek ki, aslında kaynak değişkeni içinde depoladığımız “şey” bir nesnedir. Biz bu nesneyi alıp doğruca hedef dosyamıza yazacağız. Tabii ki önce hedef dosyamızı tanımlamamız gerekiyor. Bunu da yukarıdaki kodlardaki şu satırla hallettik:

```
hedef = open("hedef.pdf", "wb")
```

Burada *hedef.pdf* adlı dosyamızı yine “ikili” düzende (*binary*) açtığımıza dikkat edin.

Artık bir önceki adımda oluşturduğumuz “pyPdf.pdf.PdfFileWriter nesnesini” hedef dosyamıza yazdırabiliriz. Bu işlemi yapan satırlarımız şunlar:

```
nesne.addPage(kaynak.getPage(0))
nesne.write(hedef)
hedef.close()
```

Burada öncelikle, `kaynak.getPage(0)` satırı yardımıyla kaynak dosyasının sıfırncı (yani ilk) sayfasını alıyoruz. `pyPdf` modülünün `addPage()` fonksiyonunu kullanarak, aldığımız bu sıfırncı sayfayı bir önceki satırda oluşturduğumuz nesnenin içine ekliyoruz. Sondan bir önceki satır yardımıyla da, içeri doldurduğumuz bu nesneyi hedef adlı dosyaya yazdırıyoruz. Son satırımız ise hedef dosyayı kapatmamızı sağlıyor.

## Pdf Belgelerinin Sayfa Sayısını Öğrenmek

`pyPdf` modülü yardımıyla, elimizdeki pdf belgelerinin sayfa sayısını da rahatlıkla sorgulayabiliriz. Bu işlemi yapmak için `pyPdf` modülünün `getNumPages()` fonksiyonundan yararlanacağız:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import warnings

warnings.simplefilter("ignore", DeprecationWarning)

from pyPdf import PdfFileReader

kaynak = PdfFileReader(open("python.pdf", "rb"))

print kaynak.getNumPages()
```

Gördüğünüz gibi, `getNumPages()` fonksiyonunu kullanarak bir pdf belgesinin sayfa sayısını öğrenmek son derece kolay.

Böylelikle Python’la temel pdf işlemlerini yapmamızı sağlayan `pyPdf` modülünü kısaca tanımış olduk. Bu modül aynı zamanda `HARMAN` programında da kullanılmıştır. Bu programı ve kaynak kodlarını incelemek için [http://istihza.com/harman/icindekiler\\_harman.html](http://istihza.com/harman/icindekiler_harman.html) adresini ziyaret edebilirsiniz.

Bu bölümde pek çok yeni şey öğrendik. Öğrendiklerimizin bir kısmı henüz bize çok fazla şey ifade etmemiş olabilir. Ama birkaç ders sonra burada anlatılan her şeyin zihninizde berraklaştığını göreceksiniz.



## 14.7 Modüllerin Yolu

Kendi yazdığımız modülleri anlatırken şöyle bir problemden söz etmiştik: Kendimiz bir modül yazdığımızda, bu modülü düzgün bir şekilde içe aktarabilmemiz için o modülün mevcut çalışma dizini içinde yer alıyor olması gerekiyor. Yani eğer içe aktarmaya çalıştığımız modül, o anda bulunduğumuz dizin içinde değilse ImportError adlı bir hata alıyoruz. Ancak böyle bir problem, mesela os modülünde bulunmuyor. Biz o anda hangi dizin altında bulunursak bulunalım, her halükarda os modülünü rahatlıkla içe aktarabiliyoruz. İşte bu bölümde biz bu durumun nedenini anlatmaya çalışacağız.

Python bir modülü içe aktarmaya çalışırken belli dizinlerin içine bakar. Eğer aranan modül o dizinlerin içinde yoksa Python yukarıda bahsettiğimiz ImportError hatasını verecektir. Peki Python bir modülü ararken hangi dizinlerin içine bakar?

Python bir modülü ararken ilk olarak mevcut çalışma dizinine bakar. Yani içe aktarmaya çalıştığımız modülü ilk olarak, komut satırının açıldığı dizin içinde bulmaya çalışır. Peki ya aranan modül mevcut çalışma dizini içinde yoksa?

Böyle bir durumda Python tabii ki hemen pes etmez. Çünkü bakması gereken başka dizinler de vardır...

Python'da sys adlı bir modül bulunur. (Bu modüle ilişkin bilgiyi "Modül Dizini" başlıklı bölümümüzde bulabilirsiniz.) Bu modülün path adlı bir niteliği vardır. İşte Python bir modülü ararken bu niteliğin gösterdiği dizinlerin içine bakar. Yani:

```
>>> import sys
>>> sys.path
['', '/usr/local/lib/python2.6/dist-packages/setuptools-0.6c11-py2.6.egg',
'/usr/local/lib/python2.6/dist-packages/docutils-0.6-py2.6.egg',
'/usr/local/lib/python2.6/dist-packages/Jinja2-2.4.1-py2.6.egg',
'/usr/local/lib/python2.6/dist-packages/Pygments-1.3.1-py2.6.egg',
'/usr/local/lib/python2.6/dist-packages/Sphinx-1.0b2-py2.6.egg',
'/usr/lib/python2.6', '/usr/lib/python2.6/plat-linux2',
'/usr/lib/python2.6/lib-tk', '/usr/lib/python2.6/lib-old',
'/usr/lib/python2.6/lib-dynload', '/usr/lib/python2.6/dist-packages',
'/usr/lib/python2.6/dist-packages/PIL', '/usr/lib/python2.6/dist-packages/gst-0.10',
'/usr/lib/pymodules/python2.6', '/usr/lib/python2.6/dist-packages/gtk-2.0',
'/usr/lib/pymodules/python2.6/gtk-2.0', '/usr/local/lib/python2.6/dist-packages']
```

Elbette kullandığınız işletim sistemine bağlı olarak sizde bu komutun çıktısı farklı olacaktır.

İşte Python bir modülü içe aktarmaya çalıştığımız zaman bu çıktıda görünen dizinlerin içini tek tek kontrol edecek ve aradığımız modülün bu dizinlerden herhangi birinin içinde olup olmadığına bakacaktır. Eğer aradığımız modül bu dizinlerden herhangi birinin içindeyse ne ala! Ama eğer modül bu dizinlerin hiç birinde yoksa işte o zaman biraz önce bahsettiğimiz o ImportError hatasını alırız.

Gördüğünüz gibi, sys.path çıktısı aslında basit bir listeden ibarettir. Dolayısıyla listeler üzerinde yapabileceğiniz her şeyi bu sys.path çıktısı üzerinde de yapabilirsiniz. Mesela bu listeye yeni dizinler ekleyebilirsiniz:

```
>>> sys.path.append("/herhangi/bir/dizin")
```

Ancak unutmayın ki, sys.path listesine sonradan eklediğiniz dizinler kalıcı değildir. Yazdığınız program sona erdiğinde sys.path listesi de eski haline dönecektir.

## 14.8 Bölüm Soruları

1. Modüller konusunu anlatırken şöyle bir örnek vermiştik:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import aritmetik

print aritmetik.cikar(1455, 32)

print "pi sayısının değeri: %s" %aritmetik.pi_sayisi
```

Bu programda, *aritmetik* modülünün içe aktarılması sırasında bir hata ortaya çıkabileceğini göz önüne alarak bu programı yeniden yazın.

2. Dediğimiz gibi, bütün Python programları temelde birer modüldür. Ancak modül ve alelade bir Python programı arasında bazı farklar da vardır. Modüller konusunu anlatırken söylediklerimizi de dikkate alarak, modülleri alelade Python programlarından ayıran özelliklerin neler olabileceğini tartışın. Bir Python programının gerçek anlamda bir modül olabilmesi ve ciddi olarak bir işe yarayabilmesi için modülü yazarken sizce hangi noktalara dikkat etmek gerekir?

3. Python modüllerini kaç farklı şekilde içe aktarabiliyoruz? Bu içe aktarma biçimleri arasındaki farklar nelerdir? Bunlar içinde sizce en mantıklısı ve en kullanışlısı hangisidir?

4. Modüller, yazdığınız Python programlarını organize etmenizi sağlar. Örneğin binlerce satırdan oluşan tek bir .py dosyası yazmaktansa, programınızı ayrı ayrı modüllere bölebilirsiniz. Mesela programınızda arayüzle ilgili olan kodları bir modül, asıl işi yapan kodları ise başka bir modül olarak düzenleyebilirsiniz. Ya da mesela programınızla ilgili ayarları kontrol eden kodları farklı bir modüle yerleştirebilirsiniz. Daha sonra birbirinden ayırdığınız bu modülleri içe aktarmak suretiyle kullanabilirsiniz. Böylece kalabalık kod yığınlarını yönetmek ve bunların bakımını yapmak çok daha kolay bir iş haline gelir. Bu bilgiler ışığında, şu ana kadar Python programlama dilini öğrenirken alıştırma olsun diye yazdığınız programları modüllere ayırmaya çalışın. Programlarınızı modüllere ayırmanın getirileri üzerinde düşünün.

5. *os* modülünü kullanarak, yalnızca GNU/Linux veya yalnızca Windows üzerinde çalışan bir program yazın.

6. *pyPdf* modülünden yararlanarak bir pdf belgesinin 2, 5 ve 7. sayfalarını ayrı bir pdf belgesi olarak kaydetmeyi deneyin.

7. Bir dizin içinde yer alan bütün dosyalar içinde, yalnızca uzantısı *.txt* olanları listeleyin.

8. Python'da *sys.path* gibi bir niteliğin olmasının, yazacağınız programlar açısından ne gibi faydaları olabileceğini tartışın.

9. *os* modülündeki *system()* fonksiyonu son derece güçlü bir araçtır. Bu güçlü aracın, yanlış kullanıldığında nelere mal olabileceği üzerinde düşünün.

# Dosya İşlemleri

Bu bölümde Python programlama dilini kullanarak dosyaları nasıl yöneteceğimizi, yani nasıl yeni bir dosya oluşturacağımızı, bir dosyaya nasıl bir şeyler yazabileceğimizi ve buna benzer işlemleri öğreneceğiz. Esasında biz bundan önceki bazı derslerimizde bu konuya üstünkörü de olsa değinmiştik. İşte bu bölüm, daha önce görüp de tam olarak anlayamadığımız bazı kod parçalarını çok daha iyi anlayabilmemizi sağlayacak bilgileri edineceğimiz bir bölüm olacak.

Hatırlarsanız `os` modülünü anlatırken Windows için `startfile()` adlı bir fonksiyondan, GNU/Linux için ise `xdg-open` adlı bir sistem komutundan söz etmiştik. Bu araçlar yardımıyla, bilgisayarımda bulunan dosyaları (ve programları), sistemdeki varsayılan uygulama ile açabiliyorduk. Ancak bu bölümde daha farklı bir şeyden söz edeceğiz. Burada yapacağımız şey, sistemimizde bulunan herhangi bir dosyayı varsayılan uygulamayla açmak değil. Biz burada Python'u kullanarak sistemimizde yeni dosyalar oluşturmanın yanısıra, varolan dosyaları da, herhangi bir aracı program kullanmadan doğrudan Python ile açacağız. Bu ikisi arasındaki farkı biraz sonra daha net bir şekilde göreceksiniz.

Programcılık yaşamınız boyunca dosyalarla bol bol haşır neşir olacaksınız. O yüzden bu bölümü dikkatle takip etmenizi öneririm. İsterseniz lafı hiç uzatmadan konumuza geçelim.

## 15.1 Dosya Oluşturmak

Bu bölümde amacımız bilgisayarımızda yeni bir dosya oluşturmak. Anlaması daha kolay olsun diye, Python'la ilk dosyamızı mevcut çalışma dizini altında oluşturacağız. Öncelikle mevcut çalışma dizinimizin ne olduğunu görelim. Hemen Python komut satırını açıyoruz ve şu komutları veriyoruz:

```
>>> import os
>>> os.getcwd()
```

Biraz sonra oluşturacağımız dosya bu komutun çıktısı olarak görünen dizin içinde oluşacaktır. Sayın ki bu dizin Masaüstü (Desktop) olsun. Mevcut çalışma dizinimizi de öğrendiğimize göre artık yeni dosyamızı oluşturabiliriz. Bu iş için `open()` adlı bir fonksiyondan faydalanacağız. Bu fonksiyonu Modüller başlığı altında `pyPdf` modülünü anlatırken de gördüğümüzü hatırlıyorsunuz...

Bu arada bir yanlış anlaşılma olmaması için hemen belirtelim. Bu fonksiyonu kullanmak için `os` modülünün içe aktarılmasına gerek yok. Biraz önce `os` modülünü içe aktarmamızın nedeni

yalnızca `getcwd()` fonksiyonunu kullanmaktı. Bu noktayı da belirttikten sonra komutumuzu veriyoruz:

```
>>> open("deneme_metni.txt", "w")
```

Böylelikle masaüstünde *deneme\_metni.txt* adlı bir metin dosyası oluşturmuş olduk. Şimdi verdiğimiz bu komutu biraz inceleyelim. `open()` fonksiyonunun ne olduğu belli. Bir dosyayı açmaya yarıyor. Tabii ortada henüz bir dosya olmadığı için burada açmak yerine yeni bir dosya oluşturmaya yaradı. Parantez içindeki *deneme\_metni.txt*'nin de ne olduğu açık. Oluşturacağımız dosyanın adını tırnak içine almayı unutmuyoruz. Peki, bu "w" ne oluyor?

Python'da dosyaları yönetirken, dosya izinlerini de belirtmemiz gerekir. Yani mesela bir dosyaya yazabilmek için "w" kipini (*mode*) kullanıyoruz. Bu harf İngilizce'de "yazma" anlamına gelen "*write*" kelimesinin kısaltmasıdır. Bunun dışında bir de "r" kipi ve "a" kipi bulunur. "r", İngilizce'de "okuma" anlamına gelen "*read*" kelimesinin kısaltması. "r" kipi, oluşturulan veya açılan bir dosyaya yalnızca "okuma" izni verildiğini gösterir. Yani bu dosya üzerinde herhangi bir değişiklik yapılamaz. Değişiklik yapabilmek için biraz önce gösterdiğimiz "w" kipini kullanmak gerekir. Bir de "a" kipi vardır, dedik. "a" da İngilizce'de "eklemek" anlamına gelen "*append*" kelimesinden geliyor. "a" kipi önceden oluşturduğumuz bir dosyaya yeni veri eklemek için kullanılır. Bu şu anlama geliyor. Örneğin *deneme\_metni.txt* adlı dosyayı "w" kipinde oluşturup içine bir şeyler yazdıktan sonra tekrar bu kiple açıp içine bir şeyler eklemek istersek dosya içindeki eski verilerin kaybolduğunu görürüz. Çünkü "w" kipi, aynı dosyadan bilgisayarımda zaten var olup olmadığına bakmaksızın, aynı adda yepyeni bir dosya oluşturacak, bunu yaparken de eski dosyayı silecektir. Dolayısıyla dosya içindeki eski verileri koruyup bu dosyaya yeni veriler eklemek istiyorsak "a" kipini kullanmamız gerekecek. Bu kiplerin hepsini sırası geldiğinde göreceğiz. Şimdi tekrar konumuza dönelim.

Biraz önce;

```
>>> open("deneme_metni.txt", "w")
```

komutuyla *deneme\_metni.txt* adında, yazma yetkisi verilmiş bir dosya oluşturduk masaüstünde. Bu komutu bir değişkene atamak, kullanım kolaylığı açısından epey faydalı olacaktır. Biz de şimdi bu işlemi yapalım:

```
>>> ilkdosyam = open("deneme_metni.txt", "w")
```

Bu arada dikkatli olun, dediğimiz gibi, eğer bilgisayarınızda önceden *deneme\_metni.txt* adlı bir dosya varsa, yukarıdaki komut size hiç bir uyarı vermeden eski dosyayı silip üzerine yazacaktır.

Şimdi başka bir örnek verelim:

```
>>> ilkdosyam = open("eski_dosya.txt", "r")
```

Dikkat ederseniz burada "w" kipi yerine "r" kipini kullandık. Biraz önce de açıkladığımız gibi bu kip dosyaya okuma yetkisi verildiğini gösteriyor. Yani eğer biz bir dosyayı bu kipte açarsak dosya içine herhangi bir veri girişi yapma imkânımız olmaz. Ayrıca bu kip yardımıyla yeni bir dosya da oluşturamayız. Bu kip bize varolan bir dosyayı açma imkânı verir. Yani mesela:

```
>>> ikincidosyam = open("deneme.txt", "r")
```

komutunu verdiğimizde eğer bilgisayarda *deneme.txt* adlı bir dosya yoksa bu adla yeni bir dosya oluşturulmayacak, bunun yerine Python bize bir hata mesajı gösterecektir:

```
>>> f = open("deneme.txt")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'deneme.txt'
```

Burada gördüğümüz hata mesajı önemlidir. IOError adlı bu hata mesajı yardımıyla, herhangi bir dosyanın sistemde var olup olmadığını denetleyebiliriz. Mesela şu basit programa bir bakalım:

```
#!/usr/bin/env python
# -*- coding: cp1254 -*-

import sys

def programdan_cik():
    print "Programdan çıkılıyor!"
    print "Hoşçakalın..."
    sys.exit()

while True:
    dosya_adi = raw_input("Açılacak dosyanın adını girin: ")

    if dosya_adi:
        try:
            f = open(dosya_adi, "r")
            print "dosya bulundu"

        except IOError:
            print "Dosya bulunamadı!"

        programdan_cik()

    else:
        print "dosya adı belirtmediniz!"
```

Burada kullanıcıdan, açılacak dosyanın adını alıyoruz. Kullanıcının, sistemde var olmayan bir dosya adı belirtmesi ihtimaline karşı da try... except bloklarından yararlanıyoruz. Eğer kullanıcının verdiği dosya adı geçerliyse, programımız *dosya bulundu* çıktısı vererek programı sonlandırıyor. Ama eğer belirtilen dosya bulunamazsa kullanıcı dosyanın bulunamadığı konusunda uyarılıyor ve kendisinden yeni bir dosya adı girmesi isteniyor. Elbette isterseniz, dosyanın bulunamaması durumunda o adla yeni bir dosya oluşturulmasını da sağlayabilirsiniz:

```
#!/usr/bin/env python
# -*- coding: cp1254 -*-

import sys

def programdan_cik():
    print "Programdan çıkılıyor!"
    print "Hoşçakalın..."
    sys.exit()

while True:
    dosya_adi = raw_input("Açılacak dosyanın adını girin: ")

    if dosya_adi:
        try:
            f = open(dosya_adi, "r")
            print "dosya bulundu"

        except IOError:
```

```
print "Dosya bulunamadı!"
open(dosya_adi, "w")
print "%s adlı yeni bir dosya oluşturuldu." %dosya_adi

programdan_cik()

else:
    print "dosya adı belirtmediniz!"
```

Yukarıdaki örneklerde, yoktan bir dosya oluşturmayı ve halihazırda sistemimizde bulunan bir dosyayı açmayı öğrendik. Python'da bir dosyayı "r" kipinde açtığımız zaman, o dosyayı yalnızca okuma hakkı elde ediyoruz. Bu kiple açtığımız bir dosya üzerinde herhangi bir değişiklik yapamayız. Eğer bir dosyayı "w" kipinde açarsak, Python belirttiğimiz addaki dosyayı sıfırdan oluşturacak, eğer aynı adla başka bir dosya varsa o dosyanın üzerine yazacaktır. Python'da dosya işlemleri yaparken, içeriği dolu bir dosyayı açıp bu dosyaya eklemeler yapmamız da gerekebilir. İşte böyle durumlar için "a" adlı özel bir kipten yararlanacağız. Bu kipi şöyle kullanıyoruz:

```
>>> dosya = open("deneme_metni.txt", "a")
```

Python'da bir dosyayı "a" kipinde açtığımız zaman, o dosyanın içine yeni veriler ekleyebiliriz. Ayrıca "a" kipinin, "r" kipinin aksine bize yeni dosya oluşturma imkânı da verdiği aklımızın bir köşesine not edelim.

Eğer yazdığımız kod içinde yukarıdaki üç kipten hiçbirini kullanmazsak; Python, öntanımlı olarak "r" kipini kullanacaktır. Tabii "r" kipinin yukarıda bahsettiğimiz özelliğinden dolayı, bilgisayarımızda yeni bir dosya oluşturmak istiyorsak, kip belirtmemiz, yani "w" veya "a" kiplerinden birini kullanmamız gerekir. Bu arada, yukarıdaki örneklerde biz dosyamızı mevcut çalışma dizini içinde oluşturduk. Tabii ki siz isterseniz tam yolu belirterek, dosyanızı istediğiniz yerde oluşturabilirsiniz. Mesela:

```
>>> dosya = open("/home/kullanıcı_adı/deneme.txt", "w")
```

komutu /home/kullanıcı\_adı/ dizini altında, yazma yetkisi verilmiş, *deneme.txt* adlı bir dosya oluşturacaktır. Ayrıca sadece .txt uzantılı dosyalar değil, pek çok farklı dosya tipi de oluşturabilirsiniz. Mesela .odt uzantılı bir dosya oluşturarak dosyanın OpenOffice ile açılmasını sağlayabilirsiniz. Ya da .html uzantılı bir dosya oluşturarak internet tarayıcınızla açılacak bir dosya oluşturabilirsiniz.

Yalnız burada küçük bir uyarı yapalım. Yazdığımız kodlarda yol adı belirtirken kullandığımız yatık çizgilerin yönüne dikkat etmemiz gerekir. En emin yol, yukarıda yaptığımız gibi dosya yolunu sağa yatık bölü işaretiyle ayırmaktır:

```
>>> dosya = open("/home/kullanıcı_adı/deneme.txt", "w")
```

Sağa yatık bölü bütün işletim sistemlerinde sorunsuz çalışır. Ama sola yatık bölü problem yaratabilir:

```
>>> f = open("C:\Documents and Settings\fozgul\Desktop\filanca.txt", "a")

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 22] invalid mode ('a') or filename: 'C:\\Documents and Settings\\x0cozgul\\Desktop\\x0cilanca.txt'
```

Burada sorun, Python'un "\" işaretini bir kaçış dizisi olarak algılaması. Halbuki biz burada bu işareti yol ayracı olarak kullanmak istedik... Eğer sağa yatık bölü kullanmak isterseniz "\" işaretini çiftlemeniz gerekir:

```
>>> f = open("C:\\Documents and Settings\\fozgul\\Desktop\\filanca.txt", "a")
```

Veya “r” adlı kaçış dizisinden yararlanabilirsiniz:

```
>>> f = open(r"C:\Documents and Settings\fozgul\Desktop\filanca.txt", "a")
```

Böylece dosya yolunu oluşturan karakter dizisi içindeki kaçış dizilerini işlevsiz hale getirerek Python’ın hata vermesini engelleyebilirsiniz.

## 15.2 Dosyaya Yazmak

Şu ana kadar öğrendiğimiz şey, Python’da dosya açmak ve oluşturmaktan ibarettir. Ancak henüz açtığımız bir dosyaya nasıl müdahale edeceğimizi veya nasıl veri girişi yapabileceğimizi bilmiyoruz. İşte birazdan, bilgisayarımızda halihazırda var olan veya bizim sonradan oluşturduğumuz bir dosyaya nasıl veri girişi yapabileceğimizi göreceğiz. Mesela *deneme.txt* adlı bir dosya oluşturarak içine “Guido Van Rossum” yazalım. Ama bu kez komut satırında değil de metin üzerinde yapalım bu işlemi. Hemen boş bir sayfa açıp içine şunları yazıyoruz:

```
#!/usr/bin/env python
# -*- coding: utf-8

dosya = open("deneme.txt", "w")
dosya.write("Guido Van Rossum")

dosya.close()
```

İlk iki satırın ne olduğunu zaten bildiğimiz için geçiyoruz.

Aynen biraz önce gördüğümüz şekilde dosya adlı bir değişken oluşturup bu değişkenin değeri olarak `open("deneme.txt", "w")` satırını belirledik. Böylelikle *deneme.txt* adında, yazma yetkisi verilmiş bir dosya oluşturduk. Daha sonra `write()` adlı bir fonksiyon yardımıyla *deneme.txt* dosyasının içine “Guido Van Rossum” yazdık. En son da `close()` adlı başka bir fonksiyondan yararlanarak dosyayı kapattık. Aslında GNU/Linux kullanıcıları bu son `dosya.close()` satırını yazmasa da olur. Ama özellikle Windows üzerinde çalışırken, eklemelerin dosyaya işlenebilmesi için dosyanın kapatılması gerekiyor. Ayrıca muhtemelen Python’un ileriki sürümlerinde, bütün platformlarda bu satırı yazmak zorunlu olacak. O yüzden bu satırı da yazmak en iyisi. Şimdi de şöyle bir şey yapalım:

Biraz önce oluşturduğumuz ve içine “Guido Van Rossum” yazdığımız dosyamıza ikinci bir satır ekleyelim:

```
#!/usr/bin/env python
# -*- coding: utf-8

dosya = open("deneme.txt", "a")
dosya.write("\nMonty Python")

dosya.close()
```

Gördüğünüz gibi bu kez dosyamızı “a” kipiyle açtık. Zaten “w” kipiyle açarsak eski dosyayı silmiş oluruz. O yüzden Python’la programlama yaparken bu tür şeylere çok dikkat etmek gerekir.

Dosyamızı “a” kipiyle açtıktan sonra `write()` fonksiyonu yardımıyla “Monty Python” satırını eski dosyaya ekledik. Burada “\n” adlı kaçış dizisinden yararlandığımıza da dikkat edin. Eğer bunu kullanmazsak eklemek istediğimiz satır bir önceki satırın hemen arkasına getirilecektir.

Bütün bunlardan sonra da `close()` fonksiyonu yardımıyla dosyamızı kapattık. Bir de şu örneğe bakalım:

```
#!/usr/bin/env python
#-*- coding: utf-8

dosya = open("şiir.txt", "w")
dosya.write("Bütün güneşler batmadan,\nBi türkü daha \
söyleyeyim bu yerde\n\t\t\t\t\t--Orhan Veli--")

dosya.close()
```

Gördüğünüz gibi, “şiir” adlı bir metin dosyası oluşturup bu dosyaya yazma yetkisi verdik. Burada bir şey dikkatinizi çekmiş olmalı. Dosya adını belirlerken (*şiir.txt*) karakter dizisinin başına bir adet “u” harfi getirdik. Siz bu harfi pyPdf modülünü işlerken de görmüştünüz. Bu “u” harfini kullanmamızın amacı, dosya adında geçen Türkçe karakterin düzgün görüntülenmesini sağlamak. Buradaki “u” harfi, “şiir.txt” adlı karakter dizisini Unicode olarak tanımlamamızı sağlıyor. Bu konuyu birkaç bölüm sonra çok daha ayrıntılı bir şekilde inceleyeceğiz. Şimdilik sadece böyle bir şeyin var olduğunu bilmemiz yeterli olacaktır...

Bu dosyanın içine yazılan verilere ve burada kaçış dizilerini nasıl kullandığımıza çok dikkat edin. İkinci mısrayı bir alt satıra almak için “\n” kaçış dizisini kullandık. Daha sonra “Orhan Veli” satırını sayfanın sağına doğru kaydırmak için “\t” kaçış dizisinden yararlandık. Bu örnekte “\n” ve “\t” kaçış dizilerini yan yana kullandık. Böylece aynı cümleyi hem alt satıra almış, hem de sağa doğru kaydırmış olduk. Ayrıca birkaç tane “\t” kaçış dizisini yan yana kullanarak cümleyi sayfanın istediğimiz noktasına getirdik.

Yukarıdaki `write()` fonksiyonu dışında çok yaygın kullanılmayan bir de `writelines()` fonksiyonu vardır. Bu fonksiyon birden fazla satırı bir kerede dosyaya işlemek için kullanılır. Şöyle ki:

```
#!/usr/bin/env python
#-*- coding: utf-8

dosya = open("şiir2.txt", "w")
dosya.writelines(["Bilmezler yalnız yaşamayanlar\n",
                  "Nasıl korku verir sessizlik insana\n",
                  "İnsan nasıl konuşur kendisiyle\n",
                  "Nasıl koşar aynalara bir cana hasret\n",
                  "Bilmezler...\n"])

dosya.close()
```

Burada parantez içindeki köşeli parantezlere dikkat edin. Aslında oluşturduğumuz şey bir liste. Dolayısıyla bu fonksiyon bir listenin içeriğini doğrudan bir dosyaya yazdırmak için faydalı olabilir. Aynı kodu `write()` fonksiyonuyla yazmaya kalkışırsanız alacağınız şey bir hata mesajı olacaktır. Eğer bir liste içinde yer alan öğeleri `write()` fonksiyonunu kullanarak dosyaya yazdırmak isterseniz `for` döngüsünden yararlanabilirsiniz:

```
>>> liste = ["elma", "armut", "kalem"]
>>> f = open("falanca.txt", "w")
>>> for i in liste:
...     f.write(i+"\n")
...
>>> f.close()
```



## 15.3 Dosyayı Okumak

Şimdiye kadar nasıl yeni bir dosya oluşturacağımızı, bu dosyaya nasıl veri gireceğimizi ve bu dosyayı nasıl kapatacağımızı öğrendik. Şimdi de oluşturduğumuz bir dosyadan nasıl veri okuyacağımızı öğreneceğiz. Bu iş için de `read()`, `readlines()` ve `readline()` fonksiyonlarından faydalanacağız. Şu örneğe bir bakalım:

```
>>> yeni = open(u"şiir.txt","w")
>>> yeni.write("Sular çekilmeye başladı \
... köklerden...\nİsınmaz mı acaba ellerimde kan? \
... \nAh,ne olur! Bütün güneşler batmadan\nBi türkü \
... daha söyleyeyim bu yerde...")
>>> yeni.close()
>>> yeni=open(u"şiir.txt")
>>> yeni.read()

'Sular \xc3\xa7ekilmeye ba\xc5\x9flad\xc4\xbl k\xc3\xb6klerden...
\nİs\xc4\xblnmaz m\xc4\xbl acaba ellerimde kan? \nAh,ne olur!
B\xc3\xbct\xc3\xbcn g\xc3\xbcneşler batmadan \nBi
t\xc3\xbcrkü daha s\xc3\x9cyleyeyim bu yerde...'
```

`yeni.read()` satırına kadar olan kısmı zaten biliyoruz. Burada kullandığımız `read()` fonksiyonu `yeni` adlı değişkenin içeriğini okumamızı sağlıyor. `yeni` adlı değişkenin değeri `şiir.txt` adlı bir dosya olduğu için, bu fonksiyon `şiir.txt` adlı dosyanın içeriğini bize gösterecektir. Gördüğünüz gibi bu komutun çıktısında Türkçe karakterler bozuk görünüyor. Ayrıca kullandığımız “\n” ifadesi de çıktıda yer alıyor. Esasında bu komut bize Python’un yazdığımız kodları nasıl gördüğünü gösteriyor. Eğer biz daha düzgün bir çıktı elde etmek istersek en son satırdaki komutu şu şekilde vermemiz gerekir:

```
>>> print yeni.read()
```

Ayrıca `read()` dışında bir de `readlines()` adlı bir fonksiyon bulunur. Eğer yukarıdaki komutu:

```
>>> yeni.readlines()
```

şeklinde verecek olursak, çıktının bir liste olduğunu görürüz.

Bir de, eğer bu `readlines()` fonksiyonunun sonundaki “s” harfini atıp;

```
>>> yeni.readline()
```

şeklinde bir kod yazarsak, dosya içeriğinin yalnızca ilk satırı okunacaktır. Python’un `readline()` fonksiyonunu değerlendirirken kullandığı ölçüt şudur: “*Dosyanın başından itibaren ilk ‘\n’ ifadesini gördüğün yere kadar oku*”. Bunların dışında, eğer istersek bir `for` döngüsü kullanarak da dosyamızı okuyabiliriz:

```
>>> yeni = open(u"şiir.txt")
>>> for satir in yeni:
...     print satir
```

Dikkat ettiyseniz:

```
>>> print yeni.readlines()
```

veya alternatif komutlarla dosya içeriğini okurken şöyle bir şey oluyor. Mesela içinde;

```
Birinci satır  
İkinci satır  
Üçüncü satır
```

yazan bir dosyamız olsun:

```
>>> dosya.readline()
```

komutuyla bu dosyanın ilk satırını okuyalım. Daha sonra tekrar bu komutu verdiğimizde birinci satırın değil, ikinci satırın okunduğunu görürüz. Çünkü Python ilk okumadan sonra imleci (Evet, biz görmesek de aslında Python'un dosya içinde gezdirdiği bir imleç var.) dosyada ikinci satırın başına kaydırıyor. Eğer bir daha verirsek bu komutu, üçüncü satır okunacaktır. Son bir kez daha bu komutu verirsek, artık dosyanın sonuna ulaşıldığı için, ekrana hiç bir şey yazılmayacaktır. Böyle bir durumda dosyayı başa sarmak için şu fonksiyonu kullanıyoruz. (Dosyamızın adının "dosya" olduğunu varsayıyoruz):

```
>>> dosya.seek(0)
```

Böylece imleci tekrar dosyanın en başına almış olduk. Tabii siz isterseniz, bu imleci farklı noktalara da taşıyabilirsiniz. Mesela:

```
>>> dosya.seek(10)
```

komutu imleci 10. karakterin başına getirecektir (Saymaya her zamanki gibi 0'dan başlıyoruz.) Bu `seek()` fonksiyonu aslında iki adet parametre alabiliyor. Şöyle ki:

```
>>> dosya.seek(5, 0)
```

komutu imleci dosyanın başından itibaren 5. karakterin bulunduğu noktaya getirir. Burada "5" sayısı imlecin kaydırılacağı noktayı, "0" sayısı ise bu işlemin dosyanın başından sonuna doğru olacağını, yani saymaya dosyanın başından başlanacağını gösteriyor:

```
>>> dosya.seek(5, 1)
```

komutu imlecin o anda bulunduğu konumdan itibaren 5. karakterin olduğu yere ilerlemesini sağlar. Burada "5" sayısı yine imlecin kaydırılacağı noktayı, "1" sayısı ise imlecin o anda bulunduğu konumun ölçüt alınacağını gösteriyor.

Son olarak:

```
>>> dosya.seek(-5,2)
```

komutu ise saymaya tersten başlanacağını, yani dosyanın başından sonuna doğru değil de sonundan başına doğru ilerlenerek, imlecin sondan 5. karakterin olduğu yere getirileceğini gösterir.

Bu ifadeler biraz karışık gelmiş olabilir. Bu konuyu anlamanın en iyi yolu bol bol uygulama yapmak ve deneyerek görmektir. İsterseniz, yukarıdaki okuma fonksiyonlarına da belirli parametreler vererek dosya içinde okunacak satırları veya karakterleri belirleyebilirsiniz. Mesela:

```
>>> yeni.readlines(3)
```

komutu dosya içinde, imlecin o anda bulunduğu noktadan itibaren 3. karakterden sonrasını okuyacaktır. Peki, o anda imlecin hangi noktada olduğunu nereden bileceğiz? Python'da bu işlem için de bir fonksiyon bulunur:

```
>>> dosya.tell()
```

komutu yardımıyla imlecin o anda hangi noktada bulunduğunu görebilirsiniz. Hatta dosyayı bir kez:

```
>>> dosya.read()
```

komutuyla tamamen okuttuktan sonra:

```
>>> dosya.tell()
```

komutunu verirsiniz imleç mevcut dosyanın en sonuna geleceği için, ekranda gördüğünüz sayı aynı zamanda mevcut dosyadaki karakter sayısına eşit olacaktır.

Python'da dosya işlemleri yaparken bilmemiz gereken en önemli noktalardan biri de şudur: Python ancak karakter dizilerini (strings) dosyaya yazdırabilir. Sayıları yazdıramaz. Eğer biz sayıları da yazdırmak istiyorsak önce bu sayıları karakter dizisine çevirmemiz gerekir. Bir örnek verelim:

```
>>> x = 50
>>> dosya = open("deneme.txt", "w")
>>> dosya.write(x)
```

Bu kodlar bize şu çıktıyı verir:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: argument 1 must be string
or read-only character buffer, not int
```

Gördüğümüz gibi Python bize bir hata mesajı gösterdi. Çünkü x değişkeninin değeri bir "sayı". Halbuki karakter dizisi olması gerekiyor. Bu meseleyi çözmek için komutumuzu şu şekilde veriyoruz. En baştan alırsak:

```
>>> x = 50
>>> dosya = open("deneme.txt", "w")
>>> dosya.write(str(x))
```

Burada:

```
>>> str(x)
```

komutuyla, bir sayı olan x değişkenini karakter dizisine çevirdik. Tabii ki bu işlemin tersi de mümkün. Eğer x bir karakter dizisi olsaydı, şu komutla onu sayıya çevirebilirdik:

```
>>> int(x)
```

## 15.4 Dosya Silmek

Peki, oluşturduğumuz bu dosyaları nasıl sileceğiz? Python'da herhangi bir şekilde oluşturduğumuz bir dosyayı silmenin en kestirme yolu şudur:

```
>>> os.remove("dosya/yolu")
```

Mesela, masaüstündeki *deneme.txt* dosyasını şöyle siliyoruz:

```
>>> import os
>>> os.remove("/home/kullanıcı_adı/Desktop/deneme.txt")
```

Eğer masaüstü zaten sizin mevcut çalışma dizininiz ise bu işlem çok daha basittir:

```
>>> import os
>>> os.remove("deneme.txt")
```

## 15.5 Dosyaya Rastgele Satır Ekleme

Şimdiye kadar hep dosya sonuna satır ekledik. Peki ya bir dosyanın ortasına bir yere satır eklemek istersek ne yapacağız? Şimdi: Diyelim ki elimizde *deneme.txt* adlı bir dosya var ve içinde şunlar yazılı:

```
Birinci Satır
İkinci Satır
Üçüncü Satır
Dördüncü Satır
Beşinci Satır
```

Biz burada *İkinci Satır* ile *Üçüncü Satır* arasına *Merhaba Python!* yazmak istiyoruz. Önce bu *deneme.txt* adlı dosyayı açalım:

```
>>> kaynak = open("deneme.txt")
```

Bu dosyayı “okuma” kipinde açtık, çünkü bu dosyaya herhangi bir yazma işlemi yapmayacağız. Yapacağımız şey, bu dosyadan veri okuyup başka bir hedef dosyaya yazmak olacak. O yüzden hemen bu hedef dosyamızı oluşturalım:

```
>>> hedef = open("test.txt", "w")
```

Bu dosyayı ise “yazma” modunda açtık. Çünkü kaynak dosyadan okuduğumuz verileri buraya yazdıracağız. Şimdi de, yapacağımız okuma işlemi tanımlayalım:

```
>>> oku = kaynak.readlines()
```

Böylece “kaynak” dosya üzerinde yapacağımız satır okuma işlemi de tanımlamış olduk...

Şimdi kaynak dosyadaki *birinci satır* ve *ikinci satır* verilerini alıp hedef dosyaya yazdırıyoruz. Bu iş için bir *for* döngüsü oluşturacağız:

```
>>> for satirlar in oku[:2]:
...     hedef.write(satirlar)
```

Burada biraz önce oluşturduğumuz “okuma işlemi” değişkeni yardımıyla “0” ve “1” no’lu satırları alıp hedef adlı dosyaya yazdırdık. Şimdi eklemek istediğimiz satır olan *Merhaba Python!* satırını ekleyeceğiz:

```
>>> hedef.write("Merhaba Python!\n")
```

Sıra geldi kaynak dosyada kalan satırları hedef dosyasına eklemeye:

```
>>> for satirlar in oku[2:]:
...     hedef.write(satirlar)
```

Artık işimiz bittiğine göre hedef ve kaynak dosyaları kapatalım:

```
>>> kaynak.close()
>>> hedef.close()
```

Bu noktadan sonra eğer istersek kaynak dosyayı silip adını da hedef dosyanın adıyla değiştirebiliriz:

```
>>> os.remove("deneme.txt")
>>> os.rename("test.txt", "deneme.txt")
```

Tabii bu son işlemleri yapmadan önce `os` modülünü içe aktarmayı unutmuyoruz...

Yukarıdaki işlemleri yapmanın daha pratik bir yolu da var. Diyelim ki elimizde, içeriği şu olan *falanca.xml* adlı bir dosya var:

```
<EnclosingTag>

<Fierce name="Item1" separator="," src="myfile1.csv" />
<Fierce name="Item2" separator="," src="myfile2.csv" />
<Fierce name="Item3" separator="," src="myfile3.csv" />
<Fierce name="Item4" separator="," src="myfile4.csv" />
<Fierce name="Item5" separator="," src="myfile5.csv" />

<NotFierce Name="Item22">
</NotFierce>

</EnclosingTag>
```

---

**Not:** Bu dosya içeriği <http://www.python-forum.org/pythonforum/viewtopic.php?f=1&t=19641> adresinden alınmıştır.

---

Biz bu dosyada, "Item2" ile "Item3" arasına yeni bir satır eklemek istiyoruz. Dilerseniz bu işlemi nasıl yapacağımızı gösteren kodları verelim önce:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

oku = open("falanca.xml")
eklenecek_satir = '<Fierce name="Item2.5" separator="," src="myfile25.csv" />'

icerik = oku.readlines()

icerik.insert(4, eklenecek_satir+"\n")

oku.close()

yaz = open("falanca.xml", "w")
yaz.writelines(icerik)
yaz.close()
```

Şimdi de bu kodları tek tek inceleyelim:

1. `oku = open("falanca.xml")` satırı yardımıyla *falanca.xml* adlı dosyayı okumak üzere açıyoruz.
2. Daha sonra, dosyaya eklemek istediğimiz satırı bir değişkene atıyoruz.
3. Hemen ardından `readlines()` adlı metodu kullanarak *falanca.xml* adlı dosyanın tüm içeriğini bir liste içinde topluyoruz. Böylece dosya içeriğini yönetmek çok daha kolay olacak. Bildiğiniz gibi, `readlines()` metodu bize çıktı olarak bir liste veriyor...
4. Bir sonraki satırda, dosyaya eklemek istediğimiz metni, `readlines()` metodu ile oluşturduğumuz listenin 4. sırasına yerleştiriyoruz. Burada listelerin `insert()` metodunu kul-

landığımıza dikkat edin.

5. Artık dosyayı okuma işlemi sona erdiği için dosyamızı *close()* metodunu kullanarak kapatıyoruz.
6. Şimdi yapmamız gereken şey, gerekli bilgileri dosyaya yazmak olacak. O yüzden bu defa *falanca.xml* adlı dosyayı yazma kipinde açıyoruz. (*yaz = open("falanca.xml", "w")*)
7. Sonra, yukarıda oluşturduğumuz içeriği, yazmak üzere açtığımız dosyaya gönderiyoruz. Bunun için *writelines()* metodunu kullandık. Bildiğiniz gibi bu metot listeleri dosyaya yazdırmak için kullanılıyor.
8. Son olarak, dosyayla işimizi bitirdiğimize göre dosyamızı kapatmayı unutmuyoruz.

## 15.6 Dosyadan Rastgele Satır Silmek

Bazen, üzerinde çalıştığımız bir dosyanın herhangi bir satırını silmemiz de gerekebilir. Bunun için yine bir önceki bölümde anlattığımız yöntemi kullanabiliriz. Dilerseniz gene yukarıda bahsettiğimiz *.xml* dosyasını örnek alalım:

```
<EnclosingTag>

<Fierce name="Item1" separator="," src="myfile1.csv" />
<Fierce name="Item2" separator="," src="myfile2.csv" />
<Fierce name="Item3" separator="," src="myfile3.csv" />
<Fierce name="Item4" separator="," src="myfile4.csv" />
<Fierce name="Item5" separator="," src="myfile5.csv" />

<NotFierce Name="Item22">
</NotFierce>

</EnclosingTag>
```

Şimdi diyelim ki biz bu dosyanın “Item2” satırını silmek istiyoruz. Bu işlemi şu kodlarla halledebiliriz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

oku = open("write_it.xml")

icerik = oku.readlines()

del icerik[3]

oku.close()

yaz = open("write_it.xml", "w")
yaz.writelines(icerik)
yaz.close()
```

Burada da, tıpkı bir önceki bölümde olduğu gibi, öncelikle *readlines()* metodunu kullanarak dosya içeriğini bir listeye gönderdik. Daha sonra bu listede silmek istediğimiz satıra karşılık gelen öğenin sıra numarasını kullanarak *del* deyimi yardımıyla ilgili öğeyi listeden kaldırdık. Son olarak da elimizdeki değiştirilmiş listeyi bir dosyaya yazdırdık.

Gördüğünüz gibi bir dosyaya veri girmek veya dosyadan veri çıkarmak son derece kolay bir işlem. Yapmamız gereken tek şey dosya içeriğini bir listeye alıp, bu liste üzerinde gerekli

değişiklikleri yapmak. Daha sonra da bu değiştirilmiş listeyi dosyaya yazdırarak amacımıza ulaşabiliyoruz.

## 15.7 os Modülünde Dosya-Dizin İşlemleri

Hatırlarsanız geçen bölümde Modüller konusunu işlerken `os` modülünden de ayrıntılı olarak bahsetmiştik. Ancak orada bu modülün bütün özelliklerini anlatmadım. `os` modülünün bazı özelliklerini anlatabilmek için önce “Dosya İşlemleri” konusunu öğrenmemizin daha doğru olacağını düşündüm. Dosya işlemleri konusunu öğrendiğimize göre, geçen derste bahsetmediğimiz bazı metod ve fonksiyonları incelemeye başlayabiliriz.

### 15.7.1 os.path.abspath()

Bu metod bir dosyanın, içinde yer aldığı dizin yolunu verir. Bunu bir örnek üzerinde görelim:

```
>>> os.path.abspath("falanca.pdf")
'C:\\Documents and Settings\\fozgul\\falanca.pdf'
```

Gördüğünüz gibi, `os.path.abspath()` fonksiyonuna argüman olarak bir dosya adı verdik. O da bize bu dosyanın tam yolunu gösterdi. Ancak bu fonksiyonun önemli bir özelliği vardır. `os.path.abspath()`, bir dosyanın sistemde var olup olmadığını kontrol etmez. Yani `os.path.abspath("falanca.pdf")` komutunu verebilmek için sisteminizde *falanca.pdf* adlı bir dosyanın olmasına gerek yok. Bu fonksiyonun yaptığı tek şey, `os.getcwd()` fonksiyonunun çıktısıyla, verdiğiniz dosya adını birleştirmektir... Bu fonksiyonu boş bir karakter dizisi ile çağırdığınızda zaten bu fonksiyonun mevcut çalışma dizinini döndürdüğünü görürsünüz:

```
>>> os.path.abspath("")
'C:\\Documents and Settings\\fozgul'
```

### 15.7.2 os.path.basename()

Bu metod size bir yol içinde yer alan dosyanın adını verir. Örneğin:

```
>>> os.path.basename("C:\\Documents and Settings\\fozgul\\falanca.pdf")
'falanca.pdf'
```

Gördüğünüz gibi, bu metod yol ve dosya adını ayıklayarak, bize dosya adını veriyor.

### 15.7.3 os.path.dirname()

Bu metod ise bir önceki metodun yaptığı işin tam tersini yapar: Dosya adını değil, yol adını döndürür:

```
>>> os.path.dirname("C:\\Documents and Settings\\fozgul\\falanca.pdf")
'C:\\Documents and Settings\\fozgul'
```

### 15.7.4 os.path.exists()

Bu metot bir dosyanın veya dizinin var olup olmadığını gösterir:

```
>>> os.path.exists("C:\\Documents and Settings\\fozgul\\falanca.pdf")
False
>>> os.path.exists("C:\\Documents and Settings\\fozgul")
True
```

### 15.7.5 os.path.isdir()

Bu metot, verilen bir yolun dizin olup olmadığını kontrol eder:

```
>>> os.path.isdir("C:\\Documents and Settings\\fozgul\\falanca.pdf")
False
```

*falanca.pdf* bir dizin olmadığı için False çıktısı aldık:

```
>>> os.path.isdir("C:\\Documents and Settings\\fozgul")
True
```

*C:\\Documents and Settings\\fozgul* ise bir dizin olduğu için True çıktısı aldık.

`os.path.isdir()` metodu, aynı zamanda bir dizinin var olup olmadığını da kontrol eder:

```
>>> os.path.isdir("C:\\falanca\\dizin")
False
```

Bilgisayarımızda *C:\\falanca\\dizin* adlı bir dizin var olmadığı için False çıktısı alıyoruz...

### 15.7.6 os.path.isfile()

Bu metot, verilen bir yolun dosya olup olmadığını kontrol eder:

```
>>> os.path.isfile("C:\\Documents and Settings\\fozgul\\deneme.txt")
True
```

Benim bilgisayarımda *deneme.txt* adlı bir dosya olduğu için True çıktısı aldım.

Bir de şuna bakalım:

```
>>> os.path.isfile("C:\\Documents and Settings\\fozgul\\falanca.txt")
False
```

Bilgisayarımda *falanca.txt* adlı bir dosya yer almadığı için bu kez False çıktısı aldım.

Bu metot dosya ve dizinler arasında da ayırım yapar:

```
>>> os.path.isfile("C:\\Documents and Settings\\fozgul")
False
```



C:\Documents and Settings\fozgul bir dosya değil, ama dizin adı olduğu için *isfile()* metodu False çıktısı veriyor.

### 15.7.7 os.path.split()

Bu metot, dosya ve dizin adını birbirinden ayırır:

```
>>> os.path.split("C:\\Documents and Settings\\fozgul\\falanca.pdf")
('C:\\Documents and Settings\\fozgul', 'falanca.pdf')
```

Dolayısıyla şu komut size bir dosyanın, içinde yer aldığı dizini gösterecektir:

```
>>> yol = os.path.split("C:\\Documents and Settings\\fozgul\\falanca.pdf")
>>> yol[0]
'C:\\Documents and Settings\\fozgul'
```

Şu ise size dosya adını verecektir:

```
>>> yol = os.path.split("C:\\Documents and Settings\\fozgul\\falanca.pdf")
>>> yol[1]
'falanca.pdf'
```

Bu metot da bir dosya ya da dizinin sistemde var olup olmadığına bakmaz...

### 15.7.8 os.path.splitext()

Bu metot, bir dosya adıyla uzantısını birbirinden ayırır:

```
>>> os.path.splitext("falanca.pdf")
('falanca', '.pdf')
```

## 15.8 Kümeler ve Dosyalar

Önceki bölümlerden birinde kümelerden bahsetmiştik. Kümeleri ilk gördüğünüzde bunların tam olarak ne işe yarayacağını anlamamış olabilirsiniz. Ama aslında kümeler Python'un epey kullanışlı araçlarından bir tanesidir. Biz kümeleri dosya işlemleri yaparken dahi kullanabiliriz.

Hatırlarsanız, kümeleri işlerken birtakım metotlardan söz etmiştik. Bildiğiniz gibi kümeler şu metotlara sahiptir:

```
add
clear
copy
difference
difference_update
discard
intersection
intersection_update
isdisjoint
issubset
```

```
issuperset
pop
remove
symmetric_difference
symmetric_difference_update
union
update
```

Bu metotlar içinde özellikle iki tanesi, dosyalarla çalışırken bizim çok işimize yarayacak. Bahsettiğimiz bu metotlar *difference()* ve *intersection()* adlı metotlardır.

Bildiğiniz gibi, *difference()* metodu iki kümenin farkını alıyor. Bu metodu şöyle kullanıyoruz:

```
>>> a_kumesi = set([1, 2, 3])

>>> b_kumesi = set([1, 2, 3, 4])

>>> b_kumesi.difference(a_kumesi)

set([4])
```

Demek ki b kümesinin a kümesinden farkı 4 adlı öge imiş... Bu metodu dosyalara da uygulayabiliriz. Diyelim ki elimizde şöyle iki adet dosya var:

```
birinci satır
ikinci satır
üçüncü satır
```

Yukarıdaki *dosya1.txt* olsun. Bir de *dosya2.txt* adlı bir dosya oluşturalım:

```
birinci satır
ikinci satır
üçüncü satır
dördüncü satır
```

Bu iki dosyanın farkını almak, yani mesela ikinci dosyada bulunup birinci dosyada bulunmayan satırları göstermek için şöyle bir kod yazabiliriz:

```
>>> f1 = open("dosya1.txt")
>>> f2 = open("dosya2.txt")

>>> s1 = set(f1)
>>> s2 = set(f2)

>>> for i in s2.difference(s1):
...     print i

dördüncü satır
```

Aynı şekilde, *intersection()* metodunu kullanarak da iki dosyada ortak olan satırları bulabilirsiniz:

```
>>> for i in s1.intersection(s2):
...     print i

birinci satır
ikinci satır
```

üçüncü satır

## 15.9 with Deyimi

Dosyalarla çalışırken asla unutmamamız gereken şey, işimizi bitirdikten sonra dosyayı kapatmaktır. Çünkü eğer bir dosya üzerinde çalıştıktan sonra o dosyayı kapatmazsak, yaptığımız değişikliklerin hepsi dosyaya işlenmeyebilir. O yüzden bu tür durumları önlemek için açılan dosyayı işlem sonunda kapatmamız gerekir. Ayrıca dosyayı kapatmak, açık bir dosya üzerinde yanlışlıkla değişiklik yapma riskini de ortadan kaldırır.

Normal şartlar altında bir dosya üzerinde çalışma döngüsü şöyle işler:

```
>>> dosya = open("dosya_adı", "w")
>>> dosya.write("bir takım şeyler...")
>>> dosya.close()
```

Burada şöyle bir yol izliyoruz:

1. Dosyayı açıyoruz.
2. Dosya üzerinde gerekli değişiklikleri yapıyoruz.
3. İşimiz bitince dosyayı kapatıyoruz.

Python'da bu döngüyü daha pratik bir şekilde yapmamızı sağlayan bir deyim bulunur. Bu deyim, *with* deyimidir. *with* deyimini yukarıdaki şöyle uygulayabiliriz:

```
>>> with open("dosya_adı", "w") as dosya:
...     dosya.write("bir takım şeyler...")
```

Bu iki satırlık kod sayesinde dosyamızı hem açmış, hem üzerinde gerekli değişiklikleri yapmış, hem de dosyayı kapatmış olduk.

*with* deyiminin en önemli özelliği, açılan dosyayı mutlaka kapatmasıdır. Yani dosya üzerinde işlem yapılırken bir hata da olursa o dosya başarıyla kapatılacaktır. Bu durumu daha net anlamak için şöyle bir örnek verelim:

```
>>> dosya = open("falanca")
>>> dosya.write("bir takım şeyler...")

Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
IOError: File not open for writing

>>> dosya

<open file 'write_it.xml', mode 'r' at 0x00F5B650>
```

Gördüğümüz gibi, dosyayı yazma kipinde açmadığımız için dosyaya veri yazdırmamız mümkün olmuyor. Böyle bir durumda Python bize bir hata mesajı gösteriyor. Ancak bu hata mesajından sonra, dosyayı henüz kapatmadığımız için dosyamız hala açık. Bu durumu dosya değişkenini ekrana yazdırarak teyit ettik. Dosya açık olduğu için üzerinde işlem yapmaya devam edebiliriz:

```
>>> dosya.read()

'varolan içerik...'
```

Bir de şuna bakalım:

```
>>> with open("falanca") as dosya:
...     dosya.write("bir takım şeyler...")

Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
IOError: File not open for writing

>>> dosya

<closed file 'write_it.xml', mode 'r' at 0x00F5B5A0>
```

Gördüğünüz gibi, dosyaya yazma işlemi sırasında hata aldığımız halde, dosya her şeye rağmen kapanıyor. Artık dosya üzerinde herhangi bir işlem yapmamız mümkün değil:

```
>>> dosya.read()

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file
```

Python bizi, kapalı bir dosya üzerinde işlem yapmaya çalıştığımız konusunda nazikçe uyarıyor...

## 15.10 Bölüm Soruları

1. Python yardımıyla, içeriği ve görünüşü aşağıdaki gibi olan, *istihza.html* adlı bir belge oluşturun:

### istihza.com Hakkında

**istihza.com**, Python programlama dili ve **Tkinter** için bir Türkçe kaynak oluşturma çalışmasıdır. Burada aynı zamanda **PyGTK** adlı arayüz takımına ilişkin Türkçe bilgi de verilmektedir. **Günlük** ise konu çeşitliliği açısından daha esnek bir bölüm olarak tasarlandı... Günlük bölümünden, bu siteye ilişkin son gelişmeler ve duyurular hakkında ayrıntılı bilgiye de erişebilirsiniz.

2. Kullanıcıdan aldığı birtakım bilgiler doğrultusunda anlamlı bir paragraf oluşturabilen bir program yazın. Oluşturduğunuz bu paragrafı daha sonra *.html* uzantılı bir dosya halinde kaydedip, sistemdeki öntanımlı internet tarayıcısı yardımıyla kullanıcıya gösterin.

3. Python ile oluşturduğunuz veya açtığınız herhangi bir dosyayla işiniz bittikten sonra eğer bu dosyayı *close()* fonksiyonu ile kapatmazsanız ne gibi sonuçlarla karşılaşabilirsiniz? Açtığınız dosyayı kapatmadan, varolan dosyayı *sağ tık > sil* yolunu takip ederek silmeye çalıştığınızda işletim sisteminiz nasıl bir tepki veriyor?

4. `os.path.abspath()` fonksiyonunun yaptığı işi taklit eden bir fonksiyon yazın.

5. Bir PDF belgesinin herhangi bir sayfasını kesip ayrı bir PDF belgesi oluşturan bir program yazın. Bu program sadece PDF belgeleri üzerinde işlem yapmalı. Eğer kullanıcı PDF dışı bir dosya adı belirtirse, programınız dosyanın PDF biçiminde olmadığı konusunda kullanıcıyı uyarmalı.

6. Bir dosyada belli bir satırın olup olmamasına göre işlem yapan bir program yazın. Mesela yazdığınız program bir dosyaya bakıp içinde o satırın geçip geçmediğini tespit edebilmeli. Eğer ilgili satır dosyada yoksa o satırı dosyaya eklemeli, aksi halde herhangi bir işlem yapmadan programı sonlandırmalı.

---

# Karakter Dizileri

---

Dikkat ettiyseniz ilk dersten bu yana sürekli olarak karakter dizileri ile haşır neşir oluyoruz. Karakter dizileriyle bu kadar içli dışlı olmamız kesinlikle boşuna değil. Zira karakter dizileri Python'ın en önemli konularından biridir. Gerçi biz şu ana kadar karakter dizilerine ilişkin epey şey öğrendik. Ancak karakter dizileri konusunda henüz görmediğimiz, ama görmemiz gereken pek çok şey de var. İşte bizim bu bölümde amacımız, karakter dizileri hakkında daha önce öğrendiklerimizi bir yandan pekiştirirken, bir yandan da bu konu hakkında öğrenmemiz gerekenleri öğrenmek. Bu bölümde, karakter dizilerini ve bunların metotlarını/fonksiyonlarını derinlemesine inceleme fırsatı bulacağız.

## 16.1 Karakter Dizisi Nedir?

İsterseniz işe en başından başlayalım ve şu soruyu soralım kendimize: “Karakter dizisi nedir?”

Kabaca tarif etmek gerekirse, tırnak içinde gösterebileceğimiz her şey bir karakter dizisidir. Mesela şu bir karakter dizisidir:

```
>>> "elma"
```

Gördüğünüz gibi, Python'da bir karakter dizisi tanımlamak için çok özel bir işlem yapmanıza gerek yok. Karakter dizisi olmasını istediğiniz bir şeyi tırnak içine aldığınız anda o artık bir karakter dizisidir. İsterseniz bu durumu teyit edelim:

```
>>> type(elma)
```

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
NameError: name 'elma' is not defined
```

Gördüğünüz gibi, “elma” kelimesini tırnak işaretleri içine almadığımızda (ve eğer daha önce elma adlı bir değişken de tanımlamamışsak) Python bize yukarıdaki gibi bir hata mesajı gösterecektir. Bir de şuna bakın:

```
>>> type("elma")
```

```
<type 'str'>
```

Ama eğer “elma” kelimesini tırnak içine alırsak her şey bir anda değişir. Bu işlemi yaptığımız anda artık “elma” kelimesi Python açısından bir karakter dizisidir. “Eee, ne olmuş yani?” dediğinizi duyar gibiyim... Belki bu bilgi şu anda size pek fazla bir şey ifade etmemiş olabilir. Ancak ileride nesne tabanlı programlama konusunu incelerken bunun ne kadar önemli bir özellik olduğunu anlayacaksınız. Şimdilik biz yolumuza devam edelim.

Yukarıdaki komutun çıktısından da anladığımız gibi, Python’da karakter dizileri *str* kısaltmasıyla gösteriliyor. Bu ifade İngilizce *string* (karakter dizisi) kelimesinin kısaltmasıdır.

Peki bir veri tipinin karakter dizisi olup olmamasının ne önemi var?

Python’da program yazarken elimizdeki herhangi bir verinin tipini bilmek son derece önemlidir. Yani o anda elimizde olan şeyin mesela bir sayı mı yoksa karakter dizisi mi olduğunu bilmek, yazdığımız kodların düzgün çalışabilmesi açısından büyük önem taşır. Mesela şu örneğe bakalım:

```
# -*- coding: utf-8 -*-  
  
with open("sayilar.txt", "w") as f:  
    for i in range(5):  
        f.write(i)
```

Bu kodlar çalışma sırasında hata verecektir. Çünkü *range()* fonksiyonu sayı üretir. Yani *range()* fonksiyonundan gelen veri tipi sayıdır. Dolayısıyla bu veri tipini doğrudan bu şekilde dosyaya yazdıramazsınız. Bu fonksiyondan gelen veri tipinin dosyaya yazdırılabilmesi için öncelikle karakter dizisine çevrilmesi gerekir:

```
# -*- coding: utf-8 -*-  
  
with open("sayilar.txt", "w") as f:  
    for i in range(5):  
        f.write(str(i))
```

Burada *str()* fonksiyonundan yararlanarak, *range()* fonksiyonunun ürettiği sayıları birer karakter dizisine çevirdik ve ondan sonra bunları doğrudan dosyaya yazdırdık.

Aynı şekilde, mesela *raw\_input()* fonksiyonu ile kullanıcıdan aldığınız verilerin tipinin de birer karakter dizisi olduğunu bilmek çok önemlidir. Eğer bunu bilmezseniz şöyle bir kod yazma hatasına düşebilirsiniz:

```
# -*- coding: utf-8 -*-  
  
sayi = raw_input("Bir sayı girin: ")  
  
print "Girdiğiniz sayının karesi: ", sayi ** 2
```

Bu kodlar çalışma sırasında hata verecektir. Çünkü dediğimiz gibi, *raw\_input()* fonksiyonu bize bir karakter dizisi verir. Eğer biz bu fonksiyonla sayı almak istiyorsak, buradan gelen karakter dizisini sayıya çevirmemiz gerekir:

```
# -*- coding: utf-8 -*-  
  
sayi = raw_input("Bir sayı girin: ")  
  
print "Girdiğiniz sayının karesi: ", int(sayi) ** 2
```

Burada *int()* fonksiyonunu kullanarak, *raw\_input()*’tan gelen karakter dizisini sayıya çevirdik. Burada elbette *raw\_input()* fonksiyonundan gelen karakter dizisini dümdüz bir şekilde sayıya çevirmeye çalışmak iyi bir yöntem değildir. Python’da herhangi bir dönüştürme işlemi yaparken, bu işlem sırasında bazı hataların da ortaya çıkabileceğini hesaba katmamız gerekir.

Mesela yukarıdaki örnekte kullanıcı sayı yerine bir harf girerse kodlarımız hata verecektir. Bu yüzden yukarıdaki gibi bir kod yazmamız gerektiğinde bu tür hatalara karşı önlem almamız gerekir:

```
# -*- coding: utf-8 -*-

def deger_hatasi():
    return "Harf yerine sayı girmeyi deneyin!"

def sifir_uyarisi():
    return "0 dışında bir sayı girin!"

sayi = raw_input("Bir sayı girin: ")

if sayi == "0":
    print sifir_uyarisi()
    quit()

try:
    n = int(sayi)
    print "Girdiğiniz sayının karesi: ", n ** 2
except ValueError:
    print deger_hatasi()
```

Bu arada, bir sayının karesini almak için Python'daki `pow()` adlı bir fonksiyondan da yararlanabiliriz:

```
>>> print pow(12, 2)
```

```
144
```

`pow()` fonksiyonu bir sayının istediğimiz bir kuvvetini hesaplamamızı sağlar. Mesela yukarıdaki örnekte biz 12 sayısının 2. kuvvetini hesapladık. Eğer 12'nin 3. kuvvetini hesaplamak isteseydik şöyle bir şey yazardık:

```
>>> print pow(12, 3)
```

```
1728
```

Yukarıdaki iki örnek sırasıyla şununla eşdeğerdir:  $12^2$  ve  $12^3$

## 16.2 Tırnak Tipleri

Dediğimiz gibi, Python'da karakter dizilerini başka veri tiplerinden ayıran en önemli özellik, tırnak içinde gösterilmeleridir. Python, karakter dizisi tanımlarken kullanabileceğimiz tırnak tipleri konusunda bize birkaç farklı seçenek sunar. Karakter dizilerini tanımlarken tek tırnak, çift tırnak veya üç tırnak işaretlerinden yararlanabiliriz. Örneğin:

```
>>> kardiz = "elma"
```

Biz yukarıda tanımladığımız `elma` karakter dizisini çift tırnak ile gösterdik. Ama siz isterseniz bu karakter dizisini tek veya üç tırnak işaretlerini kullanarak da tanımlayabileceğinizi biliyorsunuz. Hemen bir kaç örnek verelim:

```
>>> "C Programlama Dili"
>>> 'Guido Van Rossum'
>>> """Monty Python"""
```

Bunların hepsi birer karakter dizisidir. Ancak daha önceki derslerimizden edindiğimiz tecrübeye göre, bir karakter dizisini tanımlarken kullanacağımız tırnak tipi konusunda bazı noktalara da dikkat etmemiz gerekiyor. Örneğin karakter dizisi içinde bir kesme işareti varsa, karakter dizisini tek tırnak işaretleri içinde göstermemiz bazı sorunlara yol açabilir. Mesela “Python’ın 3.x sürümleri” ifadesini bir karakter dizisi olarak tanımlamak istediğimizde, kullanacağımız tırnak tipini belirlerken ifade içinde geçen kesme işaretini de dikkate almamız gerekir. Eğer bu ifadeyi karakter dizisi olarak tanımlarken tek tırnak işaretlerini kullanırsak, “Python’ın” kelimesi içinde geçen kesme işareti karışıklığa sebep olacaktır:

```
>>> print 'Python'ın 3.x sürümleri'
```

Yukarıdaki gibi bir kullanımın hataya yol açacağını biliyorsunuz. Bu tür hataları önlemek için izleyebileceğimiz birkaç farklı yol var:

```
>>> print "Python'ın 3.x sürümleri"
```

Burada kesme işaretinin karakter dizisini başlatan ve bitiren tırnak işaretleriyle karışmasını önlemek için karakter dizisini tek tırnak yerine çift tırnak içinde gösterdik. Çift tırnak yerine üç tırnak da kullanabiliriz:

```
>>> print """Python'ın 3.x sürümleri"""
```

Bu kullanım da geçerli bir yoldur. Ama eğer mutlaka tek tırnak kullanmak istersek kaçış dizilerinden faydalanmamız gerektiğini önceki derslerimizde de söylemiştik:

```
>>> print 'Python\'ın 3.x sürümleri'
```

Peki bu tırnak tiplerinden hangisini kullanmak daha iyidir? Cevap: Hangisi işinize geliyorsa! Teknik olarak bu tırnak tiplerinden hangisini kullandığınızın hiç bir önemi yok. Yazdığınız karakter dizisini tanımlamada hata vermediği sürece herhangi bir tırnak tipini kullanabilirsiniz. Python camiasında özellikle tek ve çift tırnaklar oldukça yaygın bir şekilde kullanılır. Ancak tek tırnak işareti karakter dizisi içinde geçen kesme işaretleriyle karışabileceği için, eğer kaçış dizileriyle de uğraşmak istemiyorsanız, tek tırnak yerine çift tırnak işaretlerini kullanmak daha pratik olabilir. Üç tırnak işareti ise genellikle karakter dizisi tanımlamada pek kullanılmaz. Python programcıları, birden fazla satıra bölünecek karakter dizilerini daha rahat bir şekilde oluşturmamızı sağladığı için üç tırnak işaretlerini çoğunlukla belgelendirme dizilerini tanımlarken kullanır. Mesela:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def bir_fonksiyon(parametre):
    """Bu fonksiyonun herhangi bir işlevi
    yoktur. Örnek olsun diye verilmiştir..."""
    pass
```

Gördüğünüz gibi, burada üç tırnak işaretlerini belgelendirme dizisi oluşturmak maksadıyla kullandık. Çünkü üç tırnak, öteki tırnak tiplerinden farklı olarak bize birden fazla satıra bölünmüş karakter dizilerini kolayca gösterme imkanı verir. Dilerseniz bununla ilgili şöyle bir örnek daha verelim:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

print """\
    Falanca v0.1'e hoşgeldiniz. Programı şu
    parametreleri kullanarak çalıştırabilirsiniz:
```



```
-v      sürüm numarasını görüntüler
-h      yardım belgesini görüntüler
-c      program yazarları hakkında bilgi verir
"""

def bir_fonksiyon(parametre):
    pass
```

Gördüğünüz gibi, üç tırnak sayesinde, karakter dizisini nasıl yazdıysak çıktı da aynı şekilde görünüyor... Aynı çıktıyı tek veya çift tırnakla elde etmek oldukça güçtür.

## 16.3 Karakter Dizilerini Birleştirmek

Python’la programlama maceranızda karakter dizileri ile en sık yapacağınız işlemlerden biri de karakter dizilerini birbiriyle birleştirmek olacaktır. Python’da karakter dizilerini birleştirmenin birden fazla yöntemi var:

```
>>> ad = "Fırat"
>>> soyad = "Özgül"
>>> ad_soyad = ad + soyad
>>> print ad_soyad
```

FıratÖzgül

Burada basitçe “+” işleminden yararlanarak iki karakter dizisini birleştirdik. Ancak gördüğünüz gibi, burada iki karakter dizisi arasında boşluk yok. Ama tabii siz bu boşluğu kendiniz de ekleyebilirsiniz:

```
>>> bosluk = " "
>>> print ad + bosluk + soyad
```

Fırat Özgül

Bu yöntemi kullanarak şöyle bir şey yazabilirsiniz mesela:

```
>>> a = ""
>>> for i in range(10):
...     a += str(i)
...
>>> print a
```

0123456789

Python’da karakter dizileri “değiştirilemeyen” (*immutable*) veri tipleridir. Yani karakter dizileri üzerinde değişiklik yapılamaz. Ama mesela listeler böyle değildir. Listeler değiştirilebilir (*mutable*) veri tipleri oldukları için listeler üzerinde değişiklik yapabiliyoruz. Bir örnek verelim:

```
>>> lst = ["elma", "armut", "muz"]
>>> lst.append("kiraz")
>>> print lst
```

["elma", "armut", "muz", "kiraz"]

Gördüğünüz gibi, oluşturduğumuz bir listeye *append()* metodunu kullanarak yeni bir öğe ekleyebildik. Ama karakter dizileri böyle değildir. Örneğin:

```
>>> kardiz = "elma"
>>> kardiz + "armut"

'elmaarmut'
```

Burada sanki karakter dizisi üzerinde değişiklik yapabildiğimiz gibi görünüyor, ama aslında durum öyle değil:

```
>>> print kardiz

elma
```

Gördüğümüz gibi, özgün karakter dizisinde herhangi bir değişiklik olmadı. Özgün karakter dizisi hala “elma”... Eğer bir karakter dizisi üzerinde değişiklik yapmak istersek, özgün karakter dizisinin üzerine yazmamız veya değişiklikleri farklı bir karakter dizisi içinde depolamamız gerekir. Örneğin değişiklikleri farklı bir karakter dizisinde depolamak için şöyle bir şey yazabiliriz:

```
>>> kardiz2 = kardiz + "armut"
>>> print kardiz2

elmaarmut
```

Bu şekilde “elma” değerini taşıyan kardiz adlı karakter dizisi ile “armut” karakter dizisini kardiz2 adlı başka bir değişkene atamış olduk. Ancak tabii ki özgün karakter dizisi yerli yerinde duruyor:

```
>>> print kardiz

elma
```

Dediğimiz gibi, değişiklikleri özgün karakter dizisinin üzerine de yazabiliriz:

```
>>> kardiz = kardiz + "armut"
>>> print kardiz

elmaarmut
```

Burada kardiz adlı karakter dizisini silip, yine “kardiz” adını taşıyan başka bir karakter dizisi oluşturmuş olduk. Yukarıdaki örneği şu şekilde kısaltabileceğimizi biliyorsunuz:

```
>>> kardiz += "armut"
>>> print kardiz

elmaarmut
```

Gösterdiğimiz bu özelliği daha karmaşık bir örnek içinde kullanalım. Diyelim ki elinizde şöyle bir metin var:

```
saüipö 1990 ajnjöfcö çv acöc çyaym çkş
üpsnvnvm üçşçğjöfcö hgnkuükşkngö mjzşcm zg fköcokm çkş fknfkş.
trb fkbkokökö tcfg pnoctj, mpnca rışgöknogtk zg sşphşco
hgnkuükşog tyşgdkök ibncöfjşoctj kng üçöjöö çv fkn
wköfpwt, höv/nkövx zg ocdpt x hkçk sgm epm ğcşmnj kungüko
tktügok ybgşköfg ecnjucçknogmügfkş. fpncajtjanc ügm çkş
sncüğpşofc hgnkuükşfkıkökb çkş saüipö vahvnoctj, ybgşköfg
ike çkş fgıkukmnkm acsacac hgşgm pnocfcö zgac myeym
fgıkukmnkmngşng çcumc sncüğpşoncşfc fc ecnjucçkngdgmükş.
```

Anlamsız harflerden oluşmuş gibi görünen bu metin, Türkçe bir metnin şifrelenmiş halidir. Burada hangi harfin hangi harfe karşılık geldiğini gösteren şöyle bir tablo da olsun elimizde:

ğ = f	ı = ğ	v = u	g = e	ş = r
a = y	c = a	b = z	e = ç	d = c
ç = b	f = d	i = h	h = g	k = i
j = ı	m = k	l = j	o = m	n = l
p = o	s = p	r = ö	u = ş	t = s
ö = n	y = ü	z = v	ü = t	

Bu tablo, şifrelenmiş metin içinde gördüğümüz her harfin alfabedeki karşılığını gösteriyor. Yani mesela şifrelenmiş metin içindeki “ğ” harfini “f” harfiyle değiştireceğiz.

Bu bilgileri kullanarak, yukarıdaki şifreli metni rahatlıkla çözebiliriz. Bunun için şöyle bir kod yazmamız yeterli olacaktır:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

sifreli_metin = """saüipö 1990 ajnjöfcö çv acöc çyaym çkş
üpsnvnvm üçşçjjöfcö hgnkuükşkngö mjzşcm zg fköcokm çkş fknfkş.
trb fkbkokökö tcfg pnoctj, mpnca rıışgöknogtk zg şşphşco
hgnkuükşog tyşgdkök ijbncöfjşoctj kng üçjöcö çv fkn
wköfpwt, höv/nkövx zg ocdpt x hkçk sgm epm üçşmnj kungüko
tktügok ybgşköfg ecnjucçknogmügfkş. fpncajtjanc ügm çkş
sncüğpşofc hgnkuükşfkıkökb çkş saüipö vahvncocctj, ybgşköfg
ike çkş fgıkukmnkm acsocac hgşgm pnocfcö zgac myeym
fgıkukmnkmngşng çcumc sncüğpşoncşfc fc ecnjucçkngdgmükş."""

sozluk = {"a": "y", "b": "z", "c": "a",
          "ç": "b", "d": "c", "e": "ç",
          "f": "d", "g": "e", "ğ": "f",
          "h": "g", "ı": "ğ", "i": "h",
          "j": "ı", "k": "i", "l": "j",
          "m": "k", "n": "l", "o": "m",
          "ö": "n", "p": "o", "r": "ö",
          "s": "p", "ş": "r", "t": "s",
          "u": "ş", "ü": "t", "v": "u",
          "y": "ü", "z": "v"}

sifresiz_metin = ""

for harf in sifreli_metin:
    sifresiz_metin += sozluk.get(harf, harf)

print sifresiz_metin
```

Burada karakter dizilerini nasıl birleştirdiğimize dikkat edin. Aslında bu kodlarda şöyle bir şey yapmış olduk:

1. Önce şifreli metnimizi bir karakter dizisi olarak tanımladık. Uzun bir metinle karşı karşıya olduğumuzdan, satır sonlarında bir sonraki satıra geçerken problem yaşamamak için karakter dizisini tanımlarken üç tırnak işaretlerinden yararlandık. Bildiğiniz gibi, birden fazla satıra yayılmış uzun karakter dizilerini tanımlamanın en kolay yolu üç tırnak kullanmaktır.
2. Daha sonra hangi harfin hangi harfe karşılık geldiğini gösteren tablomuzu bir sözlük haline getiriyoruz.
3. Ardından, şifreli metnin çözülmüş halini tutacak olan sifresiz\_metin adlı karakter diz-

imizi tanımlıyoruz. Bir sonraki adımda her bir karakteri birbiriyle birleştirip bu boş karakter dizisi içine göndereceğiz.

4. Şimdi de `sifreli_metin` adlı karakter dizisi içindeki her bir karakteri, sözlükteki karşılıklarına göre tek tek `sifresiz_metin` adlı karakter dizisine gönderiyoruz. Burada sözlüklerin `get()` adlı metodunu kullandığımıza dikkat edin. Eğer bu metodun nasıl kullanıldığına ilişkin şüpheleriniz varsa Sözlükler bölümündeki [get\(\) metoduna](#) bakabilirsiniz.
5. Son olarak da `sifresiz_metin` adlı karakter dizisini ekrana yazdırıyoruz.

#### Ayrıca bkz.:

Benzer bir örnek: <http://www.istihza.com/blog/this-modulu-icindeki-sifreli-metin.html/>

Dediğimiz gibi, Python'da karakter dizilerini birleştirmek için elimizde çeşitli alternatifler var. Biz yukarıda "+" işlecini kullanarak karakter dizilerini nasıl birleştirebileceğimizi gördük. Karakter dizilerini "," işaretleriyle de birleştirebiliriz:

```
>>> a = "birinci karakter dizisi"
>>> b = "ikinci karakter dizisi"
>>> print a, b

birinci karakter dizisi ikinci karakter dizisi
```

Gördüğünüz gibi, karakter dizilerini "," işaretiyle birleştirdiğimizde Python karakter dizileri arasına otomatik olarak bir boşluk yerleştiriyor. Gelin isterseniz bununla ilgili olarak az çok işe yarar bir örnek verelim:

```
# -*- coding: cp1254 -*-

import os

print "Kullanıcı adı: ", os.environ["USERNAME"]
print "Bilgisayar adı: ", os.environ["COMPUTERNAME"]
print "Ev dizini:      ", os.environ["HOMEPATH"]
print "İşlemci:        ", os.environ["PROCESSOR_IDENTIFIER"]
print "İşlemci sayısı: ", os.environ["NUMBER_OF_PROCESSORS"]
print "İşletim sistemi:", os.environ["OS"]
```

Bu betik yardımıyla kullanıcının işletim sistemine ilişkin bazı bilgileri ekrana döktük. Burada `os` modülünün `environ` adlı niteliğinden yararlandığımıza dikkat edin. `os.environ` aslında bir sözlüktür. Dolayısıyla bu sözlük birtakım anahtar-değer çiftlerinden oluşur. Bu sözlükteki bütün anahtar-değer çiftlerini ekrana dökmek için şöyle bir şey yazabilirsiniz:

```
>>> for k, v in os.environ.items():
...     print k, v
```

Bu komutların çıktısı, kullandığınız işletim sistemine göre bazı farklılıklar gösterebilir. Örneğin yukarıdaki betik Windows kurulu bir bilgisayar üzerinde yazılmıştır. Dolayısıyla bu betiğin çıktısı GNU/Linux işletim sistemlerinde farklı olacaktır.

Yukarıdaki iki örnekte de karakter dizilerini birleştirmek için "," işaretlerinden yararlandığımıza dikkat edin.

## 16.4 Karakter Dizilerini Dilimlemek

Bazı durumlarda, elimizdeki bir karakter dizisinin tamamına ihtiyacımız olmayabilir. Yazdığımız program açısından, elimizdeki karakter dizisinin sadece bir bölümünü kullanmak isteyebiliriz.

Elimizde şöyle bir karakter dizisi olduğunu varsayalım:

```
>>> url = "http://www.istihza.com"
```

Diyelim ki biz bu karakter dizisinin başındaki `http://` kısmını atmak istiyoruz... İşte bunun için karakter dizilerinin “dilimleme” (*slicing*) özelliğinden faydalanabiliriz. Yukarıdaki örneği dilimlemeden önce derseniz bir iki temel örnek üzerinden dilimlemenin ne demek olduğunu anlamaya çalışalım:

```
>>> kardiz = "elma"
>>> print kardiz[0]

e

>>> print kardiz[1]

l

>>> print kardiz[-1]

a
```

Burada `kardiz` adlı karakter dizisinin öğelerine tek tek nasıl ulaşabildiğimizi görüyorsunuz. Böyle bir kullanımı listelerde ve demetlerde de görmüştük:

```
>>> lst = ["Python", "Perl", "Ruby", "PHP"]
>>> print lst[0]

Python

>>> print lst[-1]

PHP
```

Bu yöntemi kullanarak, karakter dizilerini dilim dilim bölebiliriz. Mesela:

```
>>> kardiz = "Python"
>>> print kardiz[0:4]

Pyth

>>> print kardiz[2:5]

tho
```

Gördüğünüz gibi, bu yöntemle karakter dizilerinin istediğimiz bir parçasını alabiliyoruz. Gelin bununla ilgili bazı denemeler yapalım:

```
>>> kardiz = "Kahramanmaraşlı Abdullah"
>>> print kardiz[0:5]

Kahra

>>> print kardiz[:5]

Kahra
```

Eğer `:` işaretinden önceki değerimiz 0 ise, bu değeri yazmasak da olur. `:` işaretinden önce herhangi bir değer belirtmezsek Python saymaya en baştan başlayacak, sanki oraya 0

yazmışsınız gibi davranacaktır. Yani ":" işaretinden önceki ögenin varsayılan değeri 0'dır... Bir de şuna bakalım:

```
>>> print kardiz[3:]  
ramanmaraşlı Abdullah
```

":" işaretinden sonraki değeri yazmadığımızda ise karakter dizisinin sonuna kadar gidiliyor. Yukarıdaki komut şununla eşdeğerdir:

```
>>> print kardiz[3:len(kardiz)]
```

Şimdi isterseniz bu bölümün en başında verdiğimiz örneğe dönelim:

```
>>> url = "http://www.istihza.com"
```

Biraz önce gösterdiğimiz dilimleme yöntemini kullanarak bu karakter dizisinin en başındaki http:// kısmını atacağız:

```
>>> print url[7:]  
www.istihza.com
```

Gördüğünüz gibi, baştaki http:// kısmını gayet şık bir şekilde attık... Bunu şöyle bir örnek içinde kullanabiliriz:

```
# -*- coding: utf-8 -*-  
  
url_list = ["http://www.python.org",  
            "http://www.istihza.com",  
            "http://www.google.com",  
            "http://www.yahoo.com"]  
  
for i in url_list:  
    print i[7:]
```

Karakter dizilerinin dilimlenme özelliğinden yararlanarak yapabilecekleriniz bunlarla sınırlı değildir elbette. Mesela:

```
>>> kardiz = "Python Programlama Dili"  
>>> print kardiz[2:15:2]  
  
to rgal
```

Burada, karakter dizisinin 2. karakterinden 15. karakterine kadar olan kısmı ikişer ikişer atlayarak ekrana yazdırdık. Aynı karakter dizisinin başından sonuna kadar üçer üçer atlayarak ilerlemek isteseydik şöyle bir şey yazabilirdik:

```
>>> print kardiz[::3]  
  
Ph oaa l
```

Burada dilimleme özelliğinin varsayılan değerlerinden yararlandığımıza dikkat ediyoruz. Yani yukarıdaki kodu aslında şöyle de yazabilirdik:

```
>>> print kardiz[0:len(kardiz):3]  
  
Ph oaa l
```

Eğer bir karakter dizisini ters çevirmek isterseniz yine bu dilimleme özelliğinden faydalanabilirsiniz:

```
>>> kardiz = "Python programlama dili öğrenmesi kolay bir dildir!"
>>> print kardiz[::-1]

!ridlid rib yalok isemnerğö ilid amalmargorp nohtyP
```

Karakter dizilerini dilimleyerek yapabilecekleriniz hakkında bir örnek daha vererek bu konuyu kapatalım:

```
# -*- coding: utf-8 -*-

url_list = ["http://www.python.org",
            "http://www.istihza.com",
            "http://www.google.com",
            "http://www.yahoo.com"]

for i in url_list:
    print "ftp" + i[10:]
```

Bu örnekte, url\_list içinde bulunan adreslerin başındaki http://www kısmını atarak yerlerine ftp ifadesini getirdik...

## 16.5 Karakter Dizilerinin Metotları

Bu bölümde Python'daki karakter dizilerinin metotlarından söz edeceğiz. Şu halde isterseniz gelin Python'un bize hangi metotları sunduğunu topluca görelim:

```
>>> for i in dir(str):
...     if "_" not in i:
...         print i

['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
'__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
'__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__',
'__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
'__str__', 'capitalize', 'center', 'count', 'decode', 'encode', 'endswith',
'expandtabs', 'find', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',
'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill']
```

Gördüğünüz gibi, Python'da karakter dizilerinin bir hayli metodu varmış... Eğer bu listeleme biçimi gözünüze biraz karışık göründüyse, elbette çıktıyı istediğiniz gibi biçimlendirmek sizin elinizde.

Mesela önceki bilgilerinizi de kullanıp şöyle bir şey yaparak çıktıyı biraz daha okunaklı hale getirebilirsiniz:

```
>>> for i in dir(""):
...     print i
```

Hatta kaç adet metot olduğunu da merak ediyorsanız şöyle bir yol izleyebilirsiniz. ("Elle tek tek sayarım," diyenlere teessüflerimi iletiyorum...)

```
>>> print len(dir(""))
```

Şimdi sıra geldi bu metotları incelemeye...

### 16.5.1 Büyük-Küçük Harf Değiştirme

Python'da karakter dizilerinin büyük-küçük harf durumlarını değiştirmek için kullanabileceğimiz beş farklı metot var. Bu metotlar şöyle sıralanabilir:

**capitalize() metodu** Karakter dizilerinin ilk harfini büyütür.

**title() metodu** Karakter dizisi içinde geçen her bir kelimenin ilk harfini büyütür.

**upper() metodu** Küçük harflerden oluşan bir karakter dizisi içindeki bütün harfleri büyütür.

**lower() metodu** Büyük harflerden oluşan bir karakter dizisi içindeki bütün harfleri küçültür.

**swapcase() metodu** Karakter dizisi içindeki büyük harfleri küçük, küçük harfleri büyük hale getirir.

Bu metotların tanımlarını verdiğimiz göre, artık bunlarla ilgili örnekler yaparak bu metotların tam olarak ne işe yaradıklarını anlamaya çalışabiliriz.

İlk olarak *capitalize()* metoduyla başlayalım...

#### capitalize()

Dediğimiz gibi, *capitalize()* metodu bir karakter dizisinin ilk harfini büyütür. Örneğin:

```
>>> "adana".capitalize()
'Adana'

>>> kent = "adana"
>>> kent.capitalize()

'Adana'
```

Yalnız burada dikkat etmemiz gereken bir nokta var: Bu metot yardımıyla birden fazla kelime içeren karakter dizilerinin sadece ilk kelimesinin ilk harfini büyütebiliyoruz:

```
>>> a = "lütfen parolanızı giriniz"
>>> print a.capitalize()

Lütfen parolanızı giriniz
```

Bu metodu kullanarak bir liste içindeki bütün karakter dizilerinin ilk harfini büyütebilirsiniz:

```
>>> a = ["elma", "armut", "kebab", "salata"]
>>> for i in a:
...     print i.capitalize()
...
Elma
Armut
Kebab
Salata
```



## title()

*capitalize()* metoduna çok benzeyen bir başka metot ise *title()* adlı metottur. *capitalize()* metodu bir karakter dizisi içinde geçen yalnızca ilk kelimenin ilk harfini büyütüyordu. *title()* metodu ise bir karakter dizisi içindeki bütün kelimelerin ilk harflerini büyütüyor:

```
>>> a = u"cumhurbaşkanlığı senfoni orkestrası"  
>>> print a.title()
```

```
Cumhurbaşkanlığı Senfoni Orkestrası
```

Burada dikkat ederseniz karakter dizisini tanımlarken karakter dizisinin en başına bir “u” harfi getirdik. Esasında biz bu harfi önceki derslerimizde de görmüştük. Bir-iki ders sonra bu harfin tam olarak ne işe yaradığını gayet net bir şekilde öğreneceğiz. Şimdilik biz sadece bunun karakter dizisi içindeki Türkçe harflerin düzgün çıktı vermesini sağladığını bilelim yeter. Dilerseniz karakter dizisini “u” harfi olmadan tanımlamayı deneyip sonucun ne olduğunu kendi gözlerinizle görebilirsiniz.

Sıra geldi *upper()* adlı metodu incelemeye...

## upper()

Yukarıda yaptığımız tanımdan da anlaşılacağı gibi, bu metot yardımıyla tamamı küçük harflerden oluşan bir karakter dizisinin bütün harflerini büyütebiliriz. Hemen bir örnek verelim:

```
>>> "enflasyon".upper()
```

```
ENFLASYON
```

Gördüğünüz gibi, “enflasyon” kelimesinin bütün harflerini büyük harfe çevirdik.

*upper()* metodunun bir de tersi vardır: *lower()* metodu. *lower()* metodu büyük harflerden oluşan karakter dizilerini küçük harfli karakter dizilerine dönüştürüyor:

```
>>> a = "ARMUT"  
>>> a.lower()
```

```
armut
```

## swapcase()

Bu başlıkta inceleyeceğimiz son metodumuzun adı *swapcase()*. Bu metot bize, bir karakter dizisinin o anda sahip olduğu harflerin büyüklük ve küçüklük özellikleri arasında geçiş yapma imkanı sağlıyor. Yani, eğer o anda bir harf büyükse, bu metodu kullandığımızda o harf küçülüyor; eğer bu harf o anda küçükse, bu metot o harfi büyük harfe çeviriyor. Gördüğünüz gibi, bu metodun ne yaptığını, anlatarak açıklamak zor oluyor. O yüzden hemen birkaç örnek yapalım:

```
>>> a = "kebab"  
>>> a.swapcase()
```

```
KEBAP
```

```
>>> b = "KEBAP"  
>>> b.swapcase()
```

```
kebab
```

```
>>> c = "KeBaP"  
>>> c.swapcase()
```

```
kEbAp
```

## 16.5.2 Büyük-Küçük Harf ve Boşluk Sorgulama

Yukarıda gösterdiğimiz şekilde karakter dizilerinin büyük-küçük harf durumlarını değiştirebiliyoruz. Python bize aynı zamanda karakter dizilerinin büyük-küçük harf durumlarını sorgulama imkanı da veriyor. Bu sorgulama işlemleri için yine birtakım metotlardan yararlanıyoruz. Bu metotları şöyle sıralayabiliriz:

**islower()** Karakter dizisinin tamamen küçük harflerden oluşup oluşmadığını sorgular

**isupper()** Karakter dizisinin tamamen büyük harflerden oluşup oluşmadığını sorgular

**istitle()** Karakter dizisi içindeki kelimelerin ilk harflerinin büyük olup olmadığını sorgular

**isspace()** Karakter dizisinin tamamen boşluk karakterlerinden oluşup oluşmadığını sorgular

İlk metodumuz olan *islower()* ile başlayalım...

### islower()

Bir karakter dizisinin tamamen küçük harflerden oluşup oluşmadığını sorgulamak için *islower()* metodundan yararlanacağız. Mesela:

```
>>> kent = "istanbul"  
>>> kent.islower()
```

```
True
```

Demek ki kent değişkeninin değeri olan karakter dizisi tamamen küçük harflerden oluşuyor-muş.

Aşağıdaki örnekler ise False (yanlış) çıktısı verecektir:

```
>>> a = "İstanbul"  
>>> a.islower()
```

```
False
```

```
>>> b = "ADANA"  
>>> b.islower()
```

```
False
```

### isupper()

Bir karakter dizisinin tamamen büyük harflerden oluşup oluşmadığını denetlemek için ise *isupper()* adlı başka bir metottan faydalanıyoruz:

```
>>> a = "ADANA"
>>> a.isupper()

True

>>> "istanbul".isupper()

False
```

## istitle()

Hatırlarsanız daha önce öğrendiğimiz metotlar arasında `title()` adlı bir metot vardı. Bu metot yardımıyla tamamı küçük harflerden oluşan bir karakter dizisinin ilk harflerini büyütebiliyorduk. İşte şimdi öğreneceğimiz *istitle()* metodu da bir karakter dizisinin ilk harflerinin büyük olup olmadığını sorgulamamızı sağlayacak:

```
>>> a = "Karakter Dizisi"
>>> a.istitle()

True

>>> b = "karakter dizisi"
>>> b.istitle()

False
```

Gördüğümüz gibi, eğer karakter dizisinin ilk harfleri büyükse bu metot True çıktısı; aksi halde False çıktısı veriyor.

## isspace()

Son olarak *isspace()* adlı bir metottan söz edeceğiz. Bu metot ile, bir karakter dizisinin tamamen boşluk karakterlerinden oluşup oluşmadığını kontrol ediyoruz. Eğer bir karakter dizisi tamamen boşluk karakterlerinden oluşuyorsa, bu metot True çıktısı verecektir. Aksi halde, alacağımız çıktı False olacaktır:

```
>>> a = " "
>>> a.isspace()

True

>>> a = "selam!"
>>> a.isspace()

False

>>> a = ""
>>> a.isspace()

False
```

Son örnekten de gördüğümüz gibi, bu metodun True çıktısı verebilmesi için karakter dizisi içinde en az bir adet boşluk karakteri olması gerekiyor.

### 16.5.3 Metotlarda Türkçe Karakter Sorunu

Yukarıda anlattığımız beş metodu kullanırken bir şey dikkatinizi çekmiş olmalı. Karakter dizilerinin *upper()*, *lower()*, *title()*, *capitalize()* ve *swapcase()* adlı metotları, içinde Türkçe karakterler geçen karakter dizilerini dönüştürmede sorun çıkarabiliyor. Mesela şu örneklerle bir bakalım:

```
>>> a = "şekerli çay"
>>> print a.capitalize()
şekerli çay
```

Gördüğünüz gibi, “şekerli çay” karakter dizisinin ilk harfi olan “ş”de herhangi bir değişiklik olmadı. Halbuki *capitalize()* metodunun bu harfi büyütmesi gerekiyordu. Bu problemi şu şekilde aşabiliriz:

```
>>> a = u"şekerli çay"
>>> print a.capitalize()
Şekerli çay
```

Burada “şekerli çay” karakter dizisini “unicode” olarak tanımladık. Böylece *capitalize()* metodu bu karakter dizisinin ilk harfini doğru bir şekilde büyüttebildi.

Aynı sorun öteki metotlar için de geçerlidir:

```
>>> a = "şekerli çay"
>>> print a.upper() # "ŞEKERLİ ÇAY" vermeli.
ŞEKERLİ çAY
>>> print a.title() # "Şekerli Çay" vermeli.
ŞEkerli çAy
>>> a = "şEkErLi çAy"
>>> print a.swapcase() # "ŞeKeRlİ ÇaY" vermeli.
ŞeKeRlİ çAY
>>> a = "ŞEKERLİ ÇAY"
>>> print a.lower() # "şekerli çay" vermeli.
Şekerlİ çay
```

Yukarıdaki sorunların çoğunu, ilgili karakter dizisini unicode olarak tanımlayarak giderebiliriz:

```
>>> a = u"şekerli çay"
>>> print a.title()
Şekerli Çay
```

Ancak karakter dizisini unicode olarak tanımlamanın dahi işe yaramayacağı bir durum da vardır. Türkçe'deki "ı" harfi hiçbir dönüşümde "I" sonucunu vermez... Örneğin:

```
>>> a = u"şekerli çay"
>>> print a.upper()
ŞEKERLI ÇAY
>>> a = "şEkErLi çAy"
>>> print a.swapcase()
ŞeKeRlI ÇaY
```

Gördüğünüz gibi, "ı" harfinin büyük hali yanlış bir şekilde "I" oluyor. Aynı biçimde "I" harfi de küçültüldüğünde "ı" harfini değil, "i" harfini verecektir:

```
>>> a = u"ISLIK"
>>> print a.lower()
islik
```

Bu sorunları çözebilmek için, kendi metodlarınızı icat etmeyi deneyebilirsiniz. Örneğin Türkçe karakterleri düzgün bir şekilde büyütebilen bir *tr\_upper()* fonksiyonu tanımlayabilirsiniz:

```
# -*- coding: utf-8 -*-
lst = {u"i": u"İ",
       u"ı": u"I"}

def tr_upper(kardiz):
    for i in lst:
        if i in kardiz:
            return kardiz.replace(i, lst[i]).upper()

    return kardiz.upper()
```

Buradaki *replace()* metodunu biraz sonra işleyeceğiz...

Gelin isterseniz bu Türkçe sorunu ile ilgili bir örnek daha gösterelim:

```
>>> c = "Gün Bugündür"
>>> c.istitle()

False
```

Halbuki sonucun True olması gerekiyordu, değil mi? Peki sizce neden böyle oldu? Evet, tahmin ettiğiniz gibi, karakter dizisinin içinde Türkçe'ye özgü harfler olduğu için metodumuz yanlış sonuç verdi. Şimdi bunu düzeltmeyi öğreneceğiz. Ama önce isterseniz Türkçe karakterlerin bazen nasıl sonuçlar doğurabileceğine bir örnek verelim:

```
>>> d = "gün bugündür"
>>> print d.title()

GÜN BugÜNdÜR
```

Gördüğünüz gibi Türkçe karakterler, *title* metodunun tamamen kontrolden çıkmasına, kısa devre yapmasına yol açtı!... Normalde *title()* metodunun ne yapması gerektiğini biliyoruz:

```
>>> e = "armut bir meyvedir"
>>> e.title()

'Armut Bir Meyvedir'
```

Yukarıda gördüğümüz, Türkçe karakterler barındıran örnekte ise *title()* metodu karakter dizisi içindeki kimi harfleri büyüttü, kimi harfleri ise küçük bıraktı!

Bu durumu nasıl düzelteceğimizi biliyorsunuz:

```
>>> c = u"Gün Bugündür"
>>> c.istitle()

True
```

Burada karakter dizisini unicode olarak tanımladığımıza dikkat edin. Bu defa metotların sorunsuz işlediğini göreceksiniz.

### 16.5.4 Sağa-Sola Yaslama İşlemleri

Python'da karakter dizilerini sağa-sola yaslama işlemleri için kullanabileceğimiz üç farklı metot bulunur. Bunları şöyle sıralayabiliriz:

**center() metodu** Karakter dizilerinin sağında ve solunda, programcının belirlediği sayıda boşluk bırakarak karakter dizisini iki yana yaslar.

**ljust() metodu** Karakter dizilerinin sağında boşluk bırakarak, karakter dizisinin sola yaslanmasını sağlar.

**rjust() metodu** Karakter dizilerinin solunda boşluk bırakarak, karakter dizisinin sağa yaslanmasını sağlar.

Gelelim bu metotların tam olarak ne işe yaradığını öğrenmeye... Öncelikle *center()* metodundan başlayalım.

#### center()

Tanımında da söylediğimiz gibi, *center()* metodu bir karakter dizisini iki yana yaslamak için kullanılır. Gelin isterseniz buna küçük bir örnek verelim:

```
>>> "a".center(15)

'      a      '
```

Gördüğünüz gibi, bu metodun parantezi içine yazdığımız sayı karakter dizisinin sağında ve solunda toplam ne kadar boşluk bırakılacağını gösteriyor. Yalnız burada önemli bir ayrıntıya dikkatinizi çekmek isterim. Parantez içinde gösterilen 15 karakterlik boşluğa karakter dizisinin kendisi de dahildir. Peki bu ne anlama geliyor? Durumu hemen bir örnekle açıklamaya çalışalım:

```
>>> kardiz = "istanbul"
>>> kardiz.center(5)

'istanbul'
```

Gördüğünüz gibi, karakter dizisinin sağında ve solunda herhangi bir boşluk oluşmadı. Bunun nedeni, parantez içinde belirttiğimiz 5 sayısının "istanbul" karakter dizisi içindeki karakter sayısından az olmasıdır. Yani "istanbul" karakter dizisi bu 5 karakterlik boşluğun tamamını

zaten kendisi dolduruyor. Dolayısıyla karakter dizisinin sağında ve solunda boşluk olabilmesi için en az, karakter dizisinin uzunluğunun bir fazlasını vermemiz gerekiyor. Yani:

```
>>> kardiz.center(len("istanbul") + 1)

' istanbul '
```

Buna göre, "istanbul" karakter dizisi içinde 8 karakter bulunduğu için bizim en az 9 sayısını kullanmamız gerekiyor:

```
>>> kardiz.center(9)

' istanbul '
```

`center()` metodu bize, boşluk yerine kendi belirlediğimiz bir karakteri yerleştirme imkanı da verir:

```
>>> "istanbul".center(15, "#")

'####istanbul###'
```

Bu çıktı aslında `center()` metodunun nasıl çalıştığını çok daha net bir şekilde ortaya koyuyor. Dikkat ederseniz çıktıda toplam 15 karakter var. Bunlardan dört tanesi bizim belirlediğimiz "#" karakteri, 8 tanesi "istanbul" karakter dizisini oluşturan karakterler, 3 tanesi ise yine bizim belirlediğimiz "#" karakteri...

Gördüğünüz gibi, parantez içinde belirttiğimiz sayı bırakılacak boşluktan ziyade, bir karakter dizisinin ne kadar yer kaplayacağını gösteriyor. Yani mesela yukarıdaki örneği göz önüne alırsak, asıl karakter dizisi ("istanbul") + 7 adet "#" işareti = 15 adet karakter dizisinin yerleştirildiğini görüyoruz. Eğer karakter dizimiz, 15 harften oluşsaydı, parantez içinde verdiğimiz 15 sayısı hiç bir işe yaramayacaktı. Böyle bir durumda, "#" işaretini çıktıda gösterebilmek için parantez içinde en az 16 sayısını kullanmamız gerekirdi.

Gelelim `ljust()` metoduna...

## **`ljust()`**

Tanımında da belirttiğimiz gibi bu metot, karakter dizilerinin sağında boşluk bırakarak, karakter dizisinin sola yaslanmasını sağlar:

```
>>> "istanbul".ljust(15)

'istanbul      '
```

Tıpkı `center()` metodunda olduğu gibi, bunun parametrelerini de istediğimiz gibi düzenleyebiliriz:

```
>>> "istanbul".ljust(15, "#")

'istanbul#####'
```

Son olarak `rjust()` metodunu inceleyelim.

## **`rjust()`**

Bu metot `ljust()`'ın tersidir. Yani karakter dizilerini sağa yaslar:

```
>>> "istanbul".rjust(15, "#")  
  
'#####istanbul'
```

Sağa-sola yaslama işlemleri için kullanılan bu metotlar içinde özellikle *ljust()* ve *rjust()* epey işinize yarayacak. Mesela bu metotları kullanarak “tablovari” görünümde elde edebilirsiniz:

```
#!/usr/bin/env python  
# -*- coding: utf-8 -*-  
  
distro = ["Ubuntu", "Debian", "SuSe", "Pardus",  
          "Fedora", "PCLinuxOS", "Arch", "Gentoo",  
          "Hadron", "Truva", "Gelecek", "Mandriva",  
          "Sabayon", "Mint", "Slackware", "Mepis",  
          "CentOS", "Puppy", "GnewSense", "Ututo",  
          "Red Hat", "Knoppix", "BackTrack", "Karoshi",  
          "Kongoni", "DreamLinux", "Yoper", "Slax"]  
  
for k, v in enumerate(distro):  
    if k % 4 == 0:  
        print  
    print "%s"%(v.ljust(15)),
```

Bu kodları çalıştırdığımızda şöyle bir çıktı elde ederiz:

Ubuntu	Debian	SuSe	Pardus
Fedora	PCLinuxOS	Arch	Gentoo
Hadron	Truva	Gelecek	Mandriva
Sabayon	Mint	Slackware	Mepis
CentOS	Puppy	GnewSense	Ututo
Red Hat	Knoppix	BackTrack	Karoshi
Kongoni	DreamLinux	Yoper	Slax

Son satırdaki *ljust()* metodu yerine *rjust()* metodunu kullanırsak aynı kodlar şöyle bir çıktı verir:

Ubuntu	Debian	SuSe	Pardus
Fedora	PCLinuxOS	Arch	Gentoo
Hadron	Truva	Gelecek	Mandriva
Sabayon	Mint	Slackware	Mepis
CentOS	Puppy	GnewSense	Ututo
Red Hat	Knoppix	BackTrack	Karoshi
Kongoni	DreamLinux	Yoper	Slax

Gördüğümüz gibi, bu defa tablo öğeleri sağa yaslandı. Bu öğeleri ekrana ortalamak için ise *center()* metodunu kullanabileceğimizi biliyorsunuz. Mesela son satırı şöyle yazalım:

```
print "%s"%(v.center(15, "~")),
```

Bu şekilde şöyle bir çıktı elde ederiz:

```
~~~~~Ubuntu~~~~~ ~~~~~Debian~~~~~ ~~~~~SuSe~~~~~ ~~~~~Pardus~~~~~  
~~~~~Fedora~~~~~ ~~~~~PCLinuxOS~~~~~ ~~~~~Arch~~~~~ ~~~~~Gentoo~~~~~  
~~~~~Hadron~~~~~ ~~~~~Truva~~~~~ ~~~~~Gelecek~~~~~ ~~~~~Mandriva~~~~~  
~~~~~Sabayon~~~~~ ~~~~~Mint~~~~~ ~~~~~Slackware~~~~~ ~~~~~Mepis~~~~~  
~~~~~CentOS~~~~~ ~~~~~Puppy~~~~~ ~~~~~GnewSense~~~~~ ~~~~~Ututo~~~~~  
~~~~~Red Hat~~~~~ ~~~~~Knoppix~~~~~ ~~~~~BackTrack~~~~~ ~~~~~Karoshi~~~~~  
~~~~~Kongoni~~~~~ ~~~~~DreamLinux~~~~~ ~~~~~Yoper~~~~~ ~~~~~Slax~~~~~
```



## 16.5.5 Sıfırla Doldurma

### zfill()

Yukarıda bahsettiğimiz *ljust()*, *rjust()* gibi metotlar yardımıyla karakter dizilerinin sağını-solunu istediğimiz karakterlerle doldurabiliyorduk. Karakter dizilerinin metotları arasında yer alan *zfill()* adlı bir metot yardımıyla da bir karakter dizisinin soluna istediğimiz sayıda “0” yerleştirebiliriz:

```
>>> a = "8"
>>> a.zfill(4)
```

```
0008
```

*zfill()* metodunun kullanımıyla ilgili şöyle bir örnek verebiliriz:

```
import time

while True:
    for i in range(21):
        time.sleep(1)
        print str(i).zfill(2)
        while i > 20:
            continue
```

## 16.5.6 Karakter Değiştirme

### replace()

Python’daki karakter dizisi metotları içinde belki de en çok işimize yarayacak metotlardan birisi de bu *replace()* metodudur. “*replace*” kelimesi İngilizce’de “değiştirmek, yerine koymak” gibi anlamlara gelir. Dolayısıyla anlamından da anlaşılacağı gibi bu metot yardımıyla bir karakter dizisi içindeki karakterleri başka karakterlerle değiştiriyoruz. Metot şu formül üzerine işler:

```
karakter_dizisi.replace("eski_karakter", "yeni_karakter")
```

Hemen bir örnek vererek durumu somutlaştıralım:

```
>>> karakter = "Kahramanmaraşlı Abdullah"
>>> print karakter.replace("a", "o")
```

```
Kohromonmoroşlı Abdulloh
```

Gördüğümüz gibi, *replace()* metodu yardımıyla karakter dizisi içindeki bütün “a” harflerini kaldırıp yerlerine “o” harfini koyduk. Burada normalde *print* deyimini kullanmasak da olur, ama karakter dizisi içinde Türkçe’ye özgü harfler olduğu için, eğer :keyword‘print’ deyimini kullanmazsak çıktıda bu harfler bozuk görünecektir.

Bu metodu, isterseniz bir karakteri silmek için de kullanabilirsiniz. O zaman şöyle bir şey yapmamız gerekir:

```
>>> karakter = "Adanalı istihza"
>>> karakter_dgs = karakter.replace("a", "")
>>> print karakter_dgs
```

```
Adnlı istihz
```

Burada bir karakteri silmek için içi boş bir karakter dizisi oluşturduğumuza dikkat edin.

`replace()` metodunun, yukarıdaki formülde belirtmediğimiz üçüncü bir parametresi daha vardır. Dikkat ettiyseniz, yukarıdaki kod örneklerinde `replace` metodu karakter dizisi içindeki bir karakteri, dizi içinde geçtiği her yerde değiştiriyordu. Yani örneğin `a.replace("b", "c")` dediğimizde, `a` değişkeninin sakladığı karakter dizisi içinde ne kadar “b” harfi varsa bunların hepsi “c”ye dönüşüyor. Bahsettiğimiz üçüncü parametre yardımıyla, karakter dizisi içinde geçen harflerin kaç tanesinin değiştirileceğini belirleyebiliyoruz:

```
>>> karakter = "Adanalı istihza"
>>> karakter_dgs = karakter.replace("a", "", 2)
>>> print karakter_dgs
```

```
Adnlı istihza
```

Burada, “Adanalı istihza” karakter dizisi içinde geçen “a” harflerinden 2 tanesini siliyoruz.

“a” harfi ile “A” harfinin Python’un gözünde birbirlerinden farklı iki karakterler olduğunu unutmayın...

### 16.5.7 Karakter Dizilerinin Başlangıç ve Bitiş Değerlerini Sorgulama

Python’da karakter dizilerinin en başındaki ve en sonundaki karakterler üzerinde işlem yapabilmemizi sağlayan iki farklı metot bulunur. Bunlardan birincisi `startswith()` metodudur.

#### `startswith()`

Bu metot yardımıyla bir karakter dizisinin belirli bir harf veya karakterle başlayıp başlamadığını denetleyebiliyoruz. Örneğin:

```
>>> a = "elma"
>>> a.startswith("e")
```

```
True
```

```
>>> b = "armut"
>>> a.startswith("c")
```

```
False
```

Görüldüğü gibi eğer bir karakter dizisi parantez içinde belirtilen harf veya karakterle başlıyorsa, yani bir karakter dizisinin ilk harfi veya karakteri parantez içinde belirtilen harf veya karakterse “doğru” anlamına gelen `True` çıktısını; aksi halde ise “yanlış” anlamına gelen `False` çıktısını elde ediyoruz.

Bu metot sayesinde karakter dizilerini ilk harflerine göre sorgulayıp sonuca göre istediğimiz işlemleri yaptırabiliyoruz:

```
>>> liste = ["elma", "erik", "ev", "elbise", "karpuz", "armut", "kebab"]
>>> for i in liste:
...     if i.startswith("e"):
...         i.replace("e", "i")
...
'ılma'
'irik'
'iv'
'ilbisi'
```

Sizin bu metodu kullanarak daha faydalı kodlar yazacağınıza inanıyorum...

### endswith()

*endswith()*, yukarıda anlattığımız *startswith()* metodunun yaptığı işin tam tersini yapıyor. Hatırlarsanız *startswith()* metodu ile, bir karakter dizisinin hangi harfle başladığını denetliyorduk. İşte bu *endswith()* metodu ile ise karakter dizisinin hangi harfle bittiğini denetleyeceğiz. Kullanımı *startswith()* metoduna çok benzer:

```
>>> a = "elma"
>>> a.endswith("a")

True

>>> b = "armut"
>>> a.endswith("a")

False
```

Bu metot yardımıyla, cümle sonlarında bulunan istemediğiniz karakterleri ayıklayabilirsiniz:

```
>>> kd1 = "ekmek elden su gölden!"
>>> kd2 = "sakla samanı gelir zamanı!"
>>> kd3 = "karga karga gak dedi..."
>>> kd4 = "Vay vicdansızlar..."

>>> for i in kd1, kd2, kd3, kd4:
...     if i.endswith("!"):
...         print i.replace("!", "")
...
ekmek elden su gölden
sakla samanı gelir zamanı
```

## 16.5.8 Karakter Dizilerini Sayma

### count()

*count()* metodu bize bir karakter dizisi içinde bir karakterden kaç adet bulunduğunu denetleme imkanı verecek. Lafı uzatmadan bir örnek verelim:

```
>>> besiktas = "Sinan Paşa Pasajı"
>>> besiktas.count("a")

5
```

Demek ki “Sinan Paşa Pasajı” karakter dizisi içinde 5 adet “a” harfi varmış...

Şimdi bu metodun nerelerde kullanılabileceğine ilişkin bir örnek verelim:

```
# -*-coding: utf-8 -*-

while True:
    #replace metodu ile karakter dizisi içindeki
    #boşlukları siliyoruz.
    soru = raw_input("Bir karakter dizisi giriniz: ").replace(" ", "")
```

```
#Kümeleri kullanarak, karakter dizisi içindeki
#çift öğeleri teke indiriyoruz
ortak = set(soru)

#ortak kümesi içindeki öğeleri karakter dizisi
#içinde kaç adet geçtiğini buluyoruz.
for i in ortak:
    print "%s harfinden = => %s tane" %(i, soru.count(i))
```

Yukarıdaki çalışmada kullanıcıdan herhangi bir karakter dizisi girmesini istiyoruz. Kodlarımız bize, kullanıcının girdiği karakter dizisi içinde her bir harften kaç tane olduğunu söylüyor.

### 16.5.9 Karakter Dizilerinin Niteliğini Sorgulama

Karakterleri üçe ayırarak inceleyebiliriz:

1. alfabetik karakterler
2. nümerik karakterler
3. alfanümerik karakterler

Alfabetik karakterler, adından da az çok anlaşılacağı gibi, alfabedeki harflerin ta kendisidir. Mesela “a”, “b”, “e”, “j”, vb...

Sayılar ve “\_?=” gibi simgeler alfabetik değildir.

#### isalpha()

Bir karakterin alfabetik olup olmadığını anlamak için Python’da *isalpha()* adlı bir metottan yararlanabiliriz:

```
>>> "a".isalpha()
```

```
True
```

Demek ki “a”, alfabetik bir karakter dizisi imiş. Bir de şuna bakalım:

```
>>> "1".isalpha()
```

```
False
```

Demek ki 1 alfabetik bir karakter değilmiş.

Şöyle bir örnek vererek durumu biraz daha netleştirelim:

```
>>> for i in "abc234_?*=jkl":
...     if i.isalpha():
...         print "%s alfabetik bir karakter dizisidir"%i
...     else:
...         print "%s alfabetik bir karakter dizisi değildir"%i
...
a alfabetik bir karakter dizisidir.
b alfabetik bir karakter dizisidir.
c alfabetik bir karakter dizisidir.
2 alfabetik bir karakter dizisi değildir.
3 alfabetik bir karakter dizisi değildir.
4 alfabetik bir karakter dizisi değildir.
```

```
_ alfabetik bir karakter dizisi değildir.  
? alfabetik bir karakter dizisi değildir.  
= alfabetik bir karakter dizisi değildir.  
* alfabetik bir karakter dizisi değildir.  
j alfabetik bir karakter dizisidir.  
k alfabetik bir karakter dizisidir.  
l alfabetik bir karakter dizisidir.
```

Ancak burada dikkat edilmesi gereken bir nokta var. Mesela şu örneğe bir bakalım:

```
>>> "ğ".isalpha()  
  
False
```

Gördüğünüz gibi, *isalpha()* metodu Türkçe karakterleri doğru değerlendiremiyor. Bu sorunun üstesinden gelmek için Türkçe karakter dizilerini unicode olarak tanımlamanız gerekir:

```
>>> u"ğ".isalpha()  
  
True
```

Sadece tek bir karakterin değil, birden fazla karakterden oluşan dizilerin alfabetik olup olmadığını da denetleyebiliriz. Örneğin:

```
>>> "istihza".isalpha()  
  
True
```

Burada “istihza” karakter dizisi tamamen alfabetik karakterlerden oluştuğu için, “istihza kelimesi alfabetiktir,” diyoruz...

Bir de şuna bakalım:

```
>>> "istihza.com".isalpha()  
  
False
```

“istihza.com” karakter dizisi içinde alfabetik olmayan bir karakter bulunduğu için (“.” karakteri), bu karakter dizisi alfabetik değildir. Dolayısıyla bir karakter dizisinin alfabetik olabilmesi için, o karakter dizisini oluşturan bütün karakterlerin alfabetik olması gerekir. Tek bir alfabetik olmayan karakter dizisi dahi işi bozar.

Bu arada, içinde Türkçe karakterler geçen bir karakter dizisinin alfabetik olup olmadığını denetlerken bu karakter dizisini unicode olarak tanımlamayı unutmuyoruz:

```
>>> u"ışık".isalpha()  
  
False
```

Doğrusu şöyle olacak:

```
>>> u"ıışık".isalpha()  
  
True
```

Gelelim “nümerik” karakterlere...

## isdigit()

Nümerik karakterler alfabetik karakterlerin tersidir. Buna göre, sayıları temsil eden karakterlere nümerik adı verilir. Örneğin 1, 2, 5, 10, 20 sayıları nümerik karakterlerdir.

Bir karakterin nümerik olup olmadığını *isdigit()* metodunu kullanarak denetleyebiliriz:

```
>>> "1".isdigit()
True
```

Ama:

```
>>> "a".isdigit()
False
```

Alfabetik karakter dizilerindeki “ya hep ya hiç” kuralı nümerik karakter dizileri için de geçerlidir. Yani bir karakter dizisinin nümerik olabilmesi için, o karakter dizisini oluşturan bütün karakterlerin nümerik olması gerekir. Tek bir nümerik olmayan karakter dahi, o karakter dizisinin nümerik olmamasına yol açacaktır. . .

Sıra geldi “alfanümerik” karakterlere...

## isalnum()

Alfanümerik karakterler, yukarıda anlattığımız alfabetik ve nümerik karakterlerinin birleşimidir. Yani içinde hem alfabetik hem de nümerik karakterler barındıran dizilere alfanümerik karakter dizileri adı verilir. Tabii sadece alfabetik ve sadece nümerik karakterler barındıran diziler de aynı zamanda birer alfanümerik karakter dizisidir. . .

Tek bir karakter için konuşmak gerekirse, alfabetik veya nümerik olan karakterlere alfanümerik adı verilir.

Bir karakterin alfanümerik olup olmadığını Python’daki *isalnum()* adlı metot yardımıyla denetleyebiliriz.

Dilerseniz bu açıklamaları birkaç örnekle taçlandıralım:

```
>>> "a".isalnum()
True
```

Demek ki “a”, alfanümerik bir karaktermiş.

```
>>> "1".isalnum()
True
```

Demek ki 1 de alfanümerik bir karaktermiş. . .

Buradan şu sonuçları çıkarabiliriz:

- Bütün alfabetik karakterler aynı zamanda birer alfanümerik karakterdir.
- Bütün nümerik karakterler de aynı zamanda birer alfanümerik karakterdir.
- Bütün alfanümerik karakterler aynı zamanda birer alfabetik karakter OLMAYABİLİR.
- Bütün alfanümerik karakterler aynı zamanda birer nümerik karakter de OLMAYABİLİR.

Birden fazla karakterden oluşan dizilerin durumuna da bakalım isterseniz:

```
>>> "abc124".isalnum()
True
>>> "istihza.com".isalnum()
False
```

Gördüğünüz gibi, “istihza.com” karakter dizisi içindeki “.” karakterinden ötürü karakter dizimiz alfanümerik olma özelliği taşıyamıyor. Bir karakter dizisinin alfanümerik olabilmesi için, barındırdığı bütün karakterlerin alfanümerik olması gerekir. . .

### 16.5.10 Sekme Boşluklarını Genişletme

#### **expandtabs()**

*expandtabs()* adlı bir metot yardımıyla karakter dizileri içindeki sekme boşluklarını genişletebiliyoruz: Örneğin:

```
>>> a = "elma\tbir\tmeyvedir"
>>> print a.expandtabs(10)
elma          bir      meyvedir
```

### 16.5.11 Karakter Konumu Bulma

Python’da bir karakterin karakter dizisi içinde hangi konumda yer aldığını bulabilmek için dört farklı metottan yararlanıyoruz. Bunlar *find()*, *rfind()*, *index()* ve *rindex()* metotlarıdır.

Öncelikle *find()* metodundan başlayalım.

#### **find()**

Bu metot, bir karakterin, karakter dizisi içinde hangi konumda yer aldığını söylüyor bize:

```
>>> a = "armut"
>>> a.find("a")
0
```

*find()* metodu karakter dizilerini soldan sağa doğru okur. Dolayısıyla eğer aradığımız karakter birden fazla sayıda bulunuyorsa, çıktıda yalnızca en soldaki karakter görünecektir:

```
>>> b = "adana"
>>> a.find("a")
0
```

Gördüğünüz gibi, *find()* metodu yalnızca ilk “a” harfini gösterdi.

Eğer aradığımız karakter, o karakter dizisi içinde bulunmuyorsa, çıktıda “-1” sonucu görünecektir:

```
>>> c = "mersin"
>>> c.find("t")

-1
```

*find()* metodu bize aynı zamanda bir karakter dizisinin belli noktalarında arama yapma imkanı da sunar. Bunun için şöyle bir sözdizimini kullanabiliriz:

```
>>> "karakter_dizisi".find("aranacak_karakter", başlangıç_noktası, bitiş_noktası)
```

Bir örnek verelim:

```
>>> a = "adana"
```

Burada normal bir şekilde "a" harfini arayalım:

```
>>> a.find("a")

0
```

Doğal olarak *find()* metodu karakter dizisi içinde ilk bulunduğu "a" harfinin konumunu söyleyecektir. Bizim örneğimizde "a" harfi kelimenin başında geçtiği için çıktıda "0" ifadesini görüyoruz. Demek ki bu karakter dizisi içindeki ilk "a" harfi "0'ıncı" konumdaymış.

İstersek şöyle bir arama yöntemi de kullanabiliriz:

```
>>> a.find("a", 1, 3)
```

Bu arama yöntemi şu sonucu verecektir:

```
2
```

Bu yöntemle, "a" harfini, karakter dizisinin 1 ve 3. konumlarında arıyoruz. Bu biçimin işleyişi, daha önceki derslerimizde gördüğümüz dilimleme işlemine benzer:

```
>>> a[1:3]

"da"
```

Peki bir karakter dizisi içinde geçen herhangi bir karakterin, o karakter dizisi içinde geçtiği bütün konumları bulmanın bir yolu var mı? Elbette var. Bunun için şöyle bir şey yazabilirsiniz:

```
# -*- coding: utf-8 -*-

def nerede(kardiz, harf):
    lst = []

    for i in range(len(kardiz)):
        lst.append(kardiz.find(harf, i))

    final = set(lst)

    for o in final:
        if int(o) > -1:
            print o,
```

Bu fonksiyonu şöyle kullanıyoruz:

```
nerede(u"kahramanmaraş", "a")
```



Böylece “kahramanmaraş” karakter dizisi içindeki “a” harflerinin hangi konumlarda yer aldığını bulmuş olduk... Aynı şekilde “izmir” karakter dizisi içindeki “i” harflerinin konumunu sorgulayalım:

```
nerede(u"izmir", "i")
```

Bununla ilgili kendi kendinize bazı denemeler yaparak, işleyişi tam anlamıyla kavrayabilirsiniz.

## **rfind()**

Python’da karakter dizisi içindeki karakterlerin konumunu bulmak için kullanabileceğimiz ikinci fonksiyonumuz *rfind()* fonksiyonudur. Bu metot yukarıda anlattığımız *find()* metodu ile aynı işi yapar. Tek farklı karakter dizilerini sağdan sola doğru okumasıdır.

*find()* metodu karakter dizilerini soldan sağa doğru okur. Mesela:

```
>>> a = "adana"
>>> a.find("a")
0
```

*rfind()* metodu ise karakter dizilerini sağdan sola doğru okur. Mesela:

```
>>> a.rfind("a")
4
```

Gördüğünüz gibi, *rfind()* metodu karakter dizisini sağdan sola doğru okuduğu için öncelikle en sondaki “a” harfini döndürdü.

## **index()**

Konum bulma işlemleri için kullanabileceğimiz üçüncü metodumuz olan *index()* metodu da yukarıda anlattığımız *find()* metoduna çok benzer. İki metot da aynı işi yapar:

```
>>> a = "istanbul"
>>> a.index("t")
2
```

Bu metot da bize, tıpkı *find()* metodunda olduğu gibi, konuma göre arama olanağı sunar:

```
>>> b = "kahramanmaraş"
>>> b.index("a", 8, 10)
9
```

Demek ki, b değişkeninin tuttuğu karakter dizisinin 8 ve 10 numaralı konumları arasında “a” harfi 9. sırada yer alıyormuş....

Peki bu *index()* metodunun *find()* metodundan farkı nedir?

Hatırlarsanız *find()* metodu aradığımız karakteri bulamadığı zaman “-1” sonucunu veriyordu. *index()* metodu ise aranan karakteri bulamadığı zaman bir hata mesajı gösterir bize. Örneğin:

```
>>> c = "istanbul"
>>> c.index("m")
```

Bu kodlar şu çıktıyı verir:

```
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: substring not found
```

## rindex()

Konum bulma işlemlerinde kullanacağımız son metodumuz ise *rindex()* metodudur. Bu metod, *index()* metodu ile aynıdır. Farkları, *rindex()* metodunun karakter dizisini sağdan sola doğru; *index()* metodunun ise soldan sağa doğru okumasıdır:

```
>>> c = "adana"
>>> c.index("a")

0

>>> c.rindex("a")

4
```

## 16.5.12 join() metodu

*join()* metodu, karakter dizisi metotları arasında bulunan en önemli ve en yaygın kullanılan metotlardan biridir. O yüzden bu metodu hakkında öğrenmeniz büyük yarar var...

Bu metodu açıklamak biraz zor ve kafa karıştırıcı olabilir. O yüzden açıklama yerine doğrudan bir örnekle, bu metodun ne işe yaradığını göstermeye çalışalım.

Şöyle bir karakter dizimiz olsun:

```
>>> a = "Linux"
```

Şimdi şöyle bir işlem yapalım:

```
>>> ".".join(a)
```

Elde edeceğimiz çıktı şöyle olur:

```
L.i.n.u.x
```

Sanırım burada *join()* metodunun ne iş yaptığı anlaşılıyor. "Linux" karakter dizisi içindeki bütün karakterlerin arasına birer tane "." (nokta) koydu. Tabii ki, nokta yerine başka karakterler de kullanabiliriz:

```
>>> "*".join(a)
```

```
L*i*n*u*x
```

Dikkat ederseniz *join()* metodunun sözdizimi öteki metotlarından biraz farklı. Bu metotta parantez içine doğrudan değişkenin kendisi yazdık. Yani *a.join("\*")* gibi bir şey yazmıyoruz.

Bu metod yardımıyla ayrıca listeleri de karakter dizisine çevirebiliriz. Mesela elimizde şöyle bir liste olsun:

```
>>> a = ["python", "php", "perl", "C++", "Java"]
```

Bu listenin öğelerini tek bir karakter dizisine dönüştürmek için şu kodu kullanıyoruz:

```
>>> "; ".join(a)
'python; php; perl; C++; Java'
```

İstersek bu kodu bir değişken içinde depolayıp kalıcı hale de getirebiliriz:

```
>>> b = "; ".join(a)
>>> print b
python; php; perl; C++; Java
```

En baştaki a adlı liste de böylece bozulmadan kalmış olur:

```
>>> print a
['python', 'php', 'perl', 'C++', 'Java']
```

Hatırlarsanız biz bu metodu aslında daha önce de farklı yerlerde kullanmıştık. Mesela “Modüller” konusunu anlatırken de görmüştük bu *join()* adlı metodu... Orada bunu `os.sep.join()` şeklinde kullanmıştık.

```
>>> import os
>>> yol = ["home", "istihza", "Desktop"]

>>> os.sep.join(yol)
'home/istihza/Desktop'
```

Veya:

```
>>> yol = ["C:", "Program Files", "Python26"]

>>> os.sep.join(yol)
'C:\\Program Files\\Python26'
```

Bu kodları sırasıyla şöyle de yazabilirdik:

```
>>> yol = ["home", "istihza", "Desktop"]
>>> "/".join(yol)
'/home/istihza/Desktop/'

>>> yol = ["C:", "Program Files", "Python26"]
>>> "\\".join(yol)
'C:\\Program Files\\Python26'
```

Hatırlarsanız bu bölümün başında şöyle bir örnek vermiştik:

```
>>> a = ""
>>> for i in range(10):
...     a += str(i)
...
>>> print a
```

Bu örnek, karakter dizilerinin nasıl birleştirileceğini gösteriyordu. Aynı şeyi *join()* metodu yardımıyla çok daha verimli bir biçimde de yapabiliriz:

```
>>> "".join([str(i) for i in range(10)])  
'0123456789'
```

Gördüğünüz gibi, işimizi tek satırda hallettik. Bu örnekteki söz dizimi size biraz tuhaf görünmüş olabilir. Ama hiç endişe etmeyin. Liste Üreteçleri konusunu işlerken bu söz dizimini çok daha ayrıntılı bir şekilde inceleyeceğiz.

### 16.5.13 translate metodu

Bu metot, karakter dizisi metotları içindeki en karmaşık metotlardan birisi olmakla birlikte, zor işleri halletmekte kullanılabilecek olması açısından da bir hayli faydalı bir metottur.

*translate()* metodu yardımıyla mesela şifreli mesajları çözebiliriz. Yalnız bu metodu *string* modülündeki *maketrans()* adlı fonksiyonla birlikte kullanacağız. Bir örnek verelim.

Elimizde şöyle bir karakter dizisi olsun:

```
"tbyksr çsn jücho elu gloglu"
```

Bu şifreli mesajı çözmek için de şöyle bir ipucumuz var diyelim:

t ==> p s ==> o m ==> j

Elimizdeki ipucuna göre şifreli mesajdaki “t” harfinin karşılığı “p” olacak. Alfabetik olarak düşünürsek:

“t” harfi, “p” harfine göre, 5 harf geride kalıyor (p, r, s, ş, t)

“s” harfi “o” harfine göre 5 harf geride kalıyor (o, ö, p, r, s)

“j” harfi “m” harfine göre 4 harf geride kalıyor (j, k, l, m)

Bu çıkarımın bizi bir yere götürmeyeceği açık. Çünkü harfler arasında ortak bir ilişki bulamadık. Peki ya alfabedeki Türkçe karakterleri yok sayarsak? Bir de öyle deneyelim:

“t” harfi, “p” harfine göre, 4 harf geride kalıyor (p, r, s, t)

“s” harfi “o” harfine göre 4 harf geride kalıyor (o, p, r, s)

“j” harfi “m” harfine göre 4 harf geride kalıyor (j, k, l, m)

Böylece karakterler arasındaki ilişkiyi tespit etmiş olduk. Şimdi hemen bir metin düzenleyici açıp kodlarımızı yazmaya başlayabiliriz:

```
# -*- coding: utf8 -*-
```

Bu satırı açıklamaya gerek yok. Ne olduğunu biliyoruz:

```
import string
```

Python modülleri arasından *string* modülünü içe aktarıyoruz (import):

```
metin = "tbyksr çsn jücho elu gloglu"
```

Şifreli metnimizi bir değişkene atayarak sabitliyoruz:

```
kaynak= "defghijklmnoprstuvyzabc"  
hedef = "abcdefghijklmnoprstuvyz"
```

Burada “kaynak”, şifreli metnin yapısını; “hedef” ise alfabenin normal yapısını temsil ediyor. “kaynak” adlı değişkende “d” harfinden başlamamızın nedeni yukarıda keşfettiğimiz harfler arası ilişkidir. Dikkat ederseniz, “hedef”teki harfleri, “kaynak”taki harflere göre, her bir harf dört sıra geride kalacak şekilde yazdık. (d -> a, e ->b, gibi...) Dikkat edeceğimiz bir başka nokta ise bunları yazarken Türkçe karakter kullanmamamız gerektiğidir:

```
cevir = string.maketrans(kaynak,hedef)
```

Burada ise, yukarıda tanımladığımız harf kümeleri arasında bir çevrim işlemi başlatabilmek için *string* modülünün *maketrans()* adlı fonksiyonundan yararlanıyoruz. Bu komut, parantez içinde gösterdiğimiz kaynak değişkenini hedef değişkenine çeviriyor; aslında bu iki harf kümesi arasında bir ilişki kuruyor. Bu işlem sonucunda kaynak ve hedef değişkenleri arasındaki ilişkiyi gösteren bir formül elde etmiş olacağız:

```
soncevir = metin.translate(cevir)
```

Bu komut yardımıyla, yukarıda “cevir” olarak belirlediğimiz formülü, “metin” adlı karakter dizisine uyguluyoruz:

```
print soncevir
```

Bu komutla da son darbeyi vuruyoruz.

Şimdi bu komutlara topluca bir bakalım:

```
# -*- coding: utf8 -*-

import string

metin = "tbyksr çsn jücho elu gloglu"
kaynak= "defghijklmnoprstuvyzabc"
hedef = "abcdefghijklmnopqrstuvwxyz"
cevir = string.maketrans(kaynak,hedef)
soncevir = metin.translate(cevir)
print soncevir
```

Bu programı komut satırından çalıştırdığınızda ne elde ettiniz?

### 16.5.14 Karakter Dizilerini Parçalarına Ayırma

Python’da karakter dizilerini parçalarına ayırmak için iki metottan yararlanacağız. Bunlar *partition()* ve *rpartition()* adlı metotlar.

Önce *partition()* metoduna bakalım.

#### **partition()**

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', '', '' )
```

### **rpartition()**

*rpartition()* metodu 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.

## **16.5.15 Karakter Dizilerini Kırpma**

Python'da karakter dizilerini kırmak için üç farklı metottan yararlanacağız. Bunlar *strip()*, *rstrip()* ve *lstrip()*.

İnceleyeceğimiz ilk metot *strip()*.

### **strip()**

Bu metot bir karakter dizisinin başında (solunda) ve sonunda (sağında) yer alan boşluk ve yeni satır (\n) gibi karakterleri siler:

```
>>> a = " boşluk "  
>>> a.strip()
```

```
'boşluk'

>>> b = "boşluk\n"
>>> b.strip()

'boşluk'
```

### **rstrip()**

*rstrip()* metodu bir karakter dizisinin sadece sonunda (sağında) yer alan boşluk ve yeni satır (\n) gibi karakterleri siler:

```
>>> a = "boşluk "
>>> a.rstrip()

'boşluk'

>>> b = "boşluk\n"
>>> b.rstrip()

'boşluk'
```

### **lstrip()**

*lstrip()* metodu ise bir karakter dizisinin sadece başında (solunda) yer alan boşluk ve yeni satır (\n) gibi karakterleri siler:

```
>>> a = " boşluk"
>>> a.lstrip()

'boşluk'

>>> b = "\nboşluk"
>>> b.lstrip()

'boşluk'
```

## **16.5.16 Karakter Dizilerini Bölme**

Bu bölümde, karakter dizilerini bölmeye yarayan üç farklı metottan söz edeceğiz. Bunlar *split()*, *rsplit()* ve *splitlines()*.

**split()** Karakter dizisini soldan sağa doğru okuyarak liste haline getirir.

**rsplit()** Karakter dizisini sağdan sola doğru okuyarak liste haline getirir.

**splitlines()** Karakter dizisini satır sonlarından bölerek liste haline getirir.

Anlatmaya önce *split()* metodundan başlayalım.

### **split()**

Bu metot biraz *join()* metodunun yaptığı işi tersine çevirmeye benzer. Hatırlarsanız *join()* metodu yardımıyla bir listenin öğelerini karakter dizisi halinde sıralayabiliyorduk:

```
>>> a = ["Debian", "Pardus", "Ubuntu", "SuSe"]
>>> b = ", ".join(a)
>>> print b
```

```
Debian, Pardus, Ubuntu, SuSe
```

İşte *split()* metoduyla bu işlemi tersine çevirebiliriz:

```
>>> yeni = b.split(",")
>>> print yeni
```

```
['Debian', ' Pardus', ' Ubuntu', ' SuSe']
```

Burada karakter dizisini, virgüllerin olduğu kısımlardan böldük. Böylece her karakter dizisi farklı bir liste öğesi haline geldi:

```
>>> yeni[0]
```

```
'Debian'
```

```
>>> yeni[1]
```

```
'Pardus'
```

```
>>> yeni[2]
```

```
'Ubuntu'
```

```
>>> yeni[3]
```

```
'SuSe'
```

Eğer karakter dizisi içinde virgül yoksa, elbette *split()* metodunu yukarıdaki şekilde kullanmak istediğiniz şeyi yerine getirmeyecektir:

```
>>> kardiz = "Python Programlama Dili"
>>> kardiz.split(",")
```

```
['Python Programlama Dili']
```

Yukarıdaki gibi bir karakter dizisi ile karşı karşıya olduğunuzda, bu karakter dizisini virgüllerin olduğu kısımlardan değil, boşlukların olduğu kısımlardan bölmeniz gerekir:

```
>>> kardiz = "Python Programlama Dili"
>>> kardiz.split(" ")
```

```
['Python', 'Programlama', 'Dili']
```

Hatta eğer isterseniz karakter dizilerini belli harflerin olduğu kısımlardan da bölebilirsiniz:

```
>>> kardiz = "Python Programlama Dili"
>>> kardiz.split("P")
```

```
['', 'ython ', 'rogramlama Dili']
```

Bu metotta ayrıca isterseniz ölçütün yanısıra ikinci bir parametre daha kullanabilirsiniz:

```
>>> b = "Debian, Pardus, Ubuntu, SuSe"
>>> c = b.split(",", 1)
```



```
>>> print c
```

```
['Debian', 'Pardus', 'Ubuntu', 'SuSe']
```

Gördüğünüz gibi, parantez içinde "," ölçütünün yanına bir adet "1" sayısı koyduk. Çıktıyı dikkatle incelediğimizde *split()* metodunun bu parametre yardımıyla karakter dizisi içinde sadece bir adet bölme işlemi yaptığını görüyoruz. Yani oluşan listenin bir ögesi *Debian*, öteki ögesi de *Pardus*, *Ubuntu*, *SuSe* oldu. Bunu şu şekilde daha açık görebiliriz:

```
>>> c[0]
```

```
'Debian'
```

```
>>> c[1]
```

```
'Pardus', 'Ubuntu', 'SuSe'
```

Gördüğünüz gibi listenin 0. ögesi *Debian* iken; listenin 1. ögesi *Pardus*, *Ubuntu*, *Suse* üçlüsü. Yani bu üçlü tek bir karakter dizisi şeklinde tanımlanmış.

Yukarıda tanımladığımız yeni adlı listeyle *c* adlı listenin uzunluklarını karşılaştırarak durumu daha net görebiliriz:

```
>>> len(yeni)
```

```
4
```

```
>>> len(c)
```

```
2
```

Parantez içindeki "1" parametresini değiştirerek kendi kendinize denemeler yapmanız metodu daha iyi anlamanıza yardımcı olacaktır.

## rsplit()

Bu bölümde inceleyeceğimiz ikinci metodumuzun adı *rsplit()*. 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:

```
>>> a = "www.python.quotaless.com"
```

```
>>> a.split(".")
```

```
['www', 'python', 'quotaless', 'com']
```

```
>>> a.rsplit(".")
```

```
['www', 'python', 'quotaless', 'com']
```

Burada "www.python.quotaless.com" adlı karakter dizisini noktaların olduğu kısımlardan böldük.

Bu örnekte ikisi arasındaki fark pek belli olmasa da, *split()* metodu soldan sağa doğru okurken, *rsplit()* metodu sağdan sola doğru okuyor. Daha açık bir örnek verelim:

```
>>> orneksplit = a.split(".", 1)
```

```
>>> print orneksplit
```

```
['www', 'python.quotaless.com']

>>> ornekrsplit = a.rsplit(".", 1)
>>> print ornekrsplit

['www.python.quotaless', 'com']
```

Sanırım bu şekilde ikisi arasındaki fark daha belirgin oldu. Öyle değil diyorsanız bir de şuna bakın:

```
>>> orneksplit[0]

'www'

>>> ornekrsplit[0]

'www.python.quotaless'
```

## splitlines()

Bu bölümün son metodunun adı *splitlines()*. Bu metot yardımıyla, bir karakter dizisini satır kesme noktalarından bölerek, bölünen öğeleri liste haline getirebiliyoruz:

```
>>> satir = "Tahir olmak da ayıp değil\nZühre olmak da"
>>> print satir.splitlines()

["Tahir olmak da ayıp değil", 'Zühre olmak da']
```

Gördüğünüz gibi, *splitlines()* metodu karakter dizisini “\n” kaçış dizisinin olduğu yerden ikiye böldü...

Böylece karakter dizisi metotlarını bitirmiş olduk. Dikkat ederseniz metot listesi içindeki iki metodu anlatmadık. Bunlar *encode* ve *decode* metotları... Bunları unicode konusunu işlerken anlatmak üzere şimdilik bir kenara bırakıyoruz.

Bu konuyu iyice sindirebilmek için kendi kendinize bolca örnek ve denemeler yapmanızı, bu konuyu arada sırada tekrar etmenizi öneririm.

## 16.6 Bölüm Soruları

1. Elinizde şöyle bir liste var:

```
url_list = ['http://www.python.org',
            'http://www.istihza.com',
            'http://www.google.com',
            'http://www.yahoo.com']
```

Bu listedeki öğelerin başında bulunan `http://www` kısmını `https://` ile değiştirerek şöyle bir liste elde edin:

```
url_list = ['https://python.org',
            'https://istihza.com',
            'https://google.com',
            'https://yahoo.com']
```

Ancak betiğin yazımı sırasında hiç bir aşamada ikinci bir liste oluşturmayın. İşlemlerinizi doğrudan `url_list` adlı liste üzerinde gerçekleştirin.

**Hatırlatma:** Bir listenin öğeleri üzerinde değişiklik yapmak için şu yolu izliyorduk:

```
>>> lst = ["Python", "Ruby", "Perl", "C++"]
>>> lst[3] = "C"
>>> print lst

["Python", "Ruby", "Perl", "C"]
```

2. “Metotlarda Türkçe Karakter Sorunu” başlığı altında, Türkçe karakterleri düzgün bir şekilde büyüten `tr_upper()` adlı bir fonksiyon tanımlamıştık. Siz de aynı şekilde Türkçe karakterleri düzgün bir şekilde küçülten `tr_lower()` adlı bir fonksiyon tanımlayın.

3. Konu anlatımı sırasında şöyle bir örnek vermiştik:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

distro = ["Ubuntu", "Debian", "SuSe", "Pardus",
          "Fedora", "PCLinuxOS", "Arch", "Gentoo",
          "Hadron", "Truva", "Gelecek", "Mandriva",
          "Sabayon", "Mint", "Slackware", "Mepis",
          "CentOS", "Puppy", "GnewSense", "Ututo",
          "Red Hat", "Knoppix", "BackTrack", "Karoshi",
          "Kongoni", "DreamLinux", "Yoper", "Slax"]

for k, v in enumerate(distro):
    if k % 4 == 0:
        print
    print "%s"%(v.ljust(15)),
```

Siz bu örneği `print` yerine, `sys` modülünde anlattığımız `sys.stdout.write()` fonksiyonunu kullanarak yazın. Bu ikisi arasındaki farkların ne olduğunu açıklayın.

4. Kullanıcıdan parola belirlemesini isteyen bir betik yazın. Kullanıcı, belirlediği parolada karakterler arasında boşluk bırakmamalı.

5. Bu bölümde şöyle bir örnek verdiğimizizi hatırlıyorsunuz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

sifreli_metin = """saüipö 1990 ajnjöfcö çv acöc çyaym çkş
üpsnvnvm üçşçğjöfcö hgnkuükşkngö mjzşcm zg fköcokm çkş fknfkş.
trb fkbkokökö tcfg pnoctj, mpnca rışgöknogtk zg sşphşco
hgnkuükşog tyşgdkök ijbncöfjsoctj kng ücöjöcö çv fkn
wköfpwt, höv/nkövx zg ocdpt x hkçk sgm epm ççşmnj kungüko
tktügok ybgşköfg ecnjucçknogmügfkş. fpncajtjanc ügm çkş
sncüğpşofc hgnkuükşfkıkökb çkş saüipö vahvncöctj, ybgşköfg
ike çkş fgıkukmnkm acsocac hgşgm pnoçfcö zgac myeym
fgıkukmnkmngşng çcumc sncüğpşoncşfc fc ecnjucçkngdgmükş."""

sozluk = {"a": "y", "b": "z", "c": "a",
          "ç": "b", "d": "c", "e": "ç",
          "f": "d", "g": "e", "ğ": "f",
          "h": "g", "ı": "ğ", "i": "h",
          "j": "ı", "k": "i", "l": "j",
          "m": "k", "n": "l", "o": "m",
```

```
"ö": "n", "p": "o", "r": "ö",  
"s": "p", "ş": "r", "t": "s",  
"u": "ş", "ü": "t", "v": "u",  
"y": "ü", "z": "v"}  
  
sifresiz_metin = ""  
  
for harf in sifreli_metin:  
    sifresiz_metin += sozluk.get(harf, harf)  
  
print sifresiz_metin
```

Siz şimdi bu örneği bir de *join()* metodunu kullanarak yazmayı deneyin. Eğer yazamazsanız endişe etmeyin, çünkü, daha önce de söylediğimiz gibi, bu örneği *join()* metodunu kullanarak rahatlıkla yazmanızı sağlayacak bilgiyi birkaç bölüm sonra daha ayrıntılı bir şekilde edineceğiz.

# Biçim Düzenleyiciler

Bu bölümde, daha önce sık sık kullandığımız, ancak ayrıntılı bir şekilde hiç incelemediğimiz bir konudan söz edeceğiz. Konumuz “karakter dizilerinde biçim düzenleyiciler”. Yabancılar buna “*format modifiers*” adı veriyor...

Dediğimiz gibi, biz daha önceki konularımızda biçim düzenleyicilerden yararlanmıştık. Dolayısıyla yabancıyı olduğumuz bir konu değil bu.

Python’da her türlü biçim düzenleme işlemi için tek bir simge bulunur. Bu simge “%”dir. Biz bunu daha önceki derslerimizde şu şekilde kullanabileceğimizi görmüştük:

```
>>> print "Benim adım %s" %"istihza"
```

Burada “%” adlı biçim düzenleyiciyi “s” karakteriyle birlikte kullandık. Bu kodlardaki “s” karakteri İngilizce “*string*”, yani “karakter dizisi” ifadesinin kısaltmasıdır.

Python’da biçim düzenleyicileri kullanırken dikkat etmemiz gereken en önemli nokta, karakter dizisi içinde kullandığımız biçimlendirici sayısı, karakter dizisinin dışında bu biçimlendiricilere karşılık gelen değerlerin sayısının aynı olmasıdır. Bu ne demek oluyor? Hemen şu örneğe bakalım:

```
>>> print "Benim adım %s, soyadım %s" %"istihza"
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: not enough arguments for format string
```

Gördüğünüz gibi bu kodlar hata verdi. Çünkü karakter dizisi içindeki iki adet “%s” ifadesine karşılık, karakter dizisinin dışında tek bir değer var (“istihza”). Halbuki bizim şöyle bir kod yazmamız gerekiyordu:

```
>>> isim = "istihza"
>>> print "%s adlı kişinin mekanı www.%s.com adresidir." %(isim, isim)
```

Bu defa herhangi bir hata mesajı almadık. Çünkü bu kodlarda, olması gerektiği gibi, karakter dizisi içindeki iki adet “%s” ifadesine karşılık, dışarıda iki adet değer var...

Bu arada, yukarıdaki örnek biçim düzenleyiciler hakkında bize önemli bir bilgi daha veriyor. Dikkat ettiyseniz, karakter dizisi dışında tek bir değer varsa bunu parantez içinde belirtmemize gerek yok. Ama eğer değer sayısı birden fazlaysa bu değerleri bir “demet” (tuple)

olarak tanımlamamız gerekiyor. Yani bu değerleri parantez içinde göstermeliyiz. Aksi halde yazdığımız kodlar hata verecektir.

Yukarıdaki örneklerde “%” adlı biçim düzenleyiciyi “s” karakteriyle birlikte kullandık. Esasında en yaygın çift de budur. Yani etraftaki Python programlarında yaygın olarak “%s” yapısını görürüz... Ancak Python’da “%” biçim düzenleyicisiyle birlikte kullanılacak tek karakter “s” değildir. Daha önce de dediğimiz gibi, “s” karakteri “string”, yani “karakter dizisi” kelimesinin kısaltmasıdır. Yani aslında “%s” yapısı Python’da özel olarak karakter dizilerini temsil eder. Peki bu ne demek oluyor? Bir karakter dizisi içinde “%s” yapısını kullandığımızda, dışarıda buna karşılık gelen değerlerin bir karakter dizisi veya karakter dizisine çevrilebilecek bir değer olması gerek. Bunun tam olarak ne demek olduğunu biraz sonra daha net bir şekilde anlayacağız. “Biraz sabır,” diyerek yolumuza devam edelim...

Gördüğünüz gibi, Python’da biçim düzenleyici olarak kullanılan simge aynı zamanda “yüzde” (%) anlamına da geliyor... O halde size şöyle bir soru sorayım: Acaba 0’dan 100’e kadar olan sayıların başına birer “yüzde” işareti koyarak bu sayıları nasıl gösterirsiniz? %0, %1, %10, %15, gibi... Önce şöyle bir şey deneyelim:

```
>>> for i in range(100):
...     print "%s" %i
...
```

Bu kodlar tabii ki sadece 0’dan 100’e kadar olan sayıları ekrana dökmekle yetinecektir... Sayıların başında “%” işaretini göremeyeceğiz...

Bir de şöyle bir şey deneyelim:

```
>>> for i in range(100):
...     print "%s" %i
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: not all arguments converted during string formatting
```

Bu defa da hata mesajı aldık... Doğru cevap şu olmalıydı:

```
>>> for i in range(100):
...     print "%s" %i
...
```

Burada “%” işaretini arka arkaya iki kez kullanarak bir adet “%” işareti elde ettik. Daha sonra da normal bir şekilde “%s” biçimini kullandık. Yani üç adet “%” işaretini yan yana getirmiş olduk.

## 17.1 Biçim Düzenlemede Kullanılan Karakterler

Daha önce de dediğimiz gibi, biçim düzenleyici karakterler içinde en yaygın kullanılanı “s” harfidir ve bu harf, karakter dizilerini ya da karakter dizisine dönüştürülebilen değerleri temsil eder. Ancak Python’da “%” adlı biçim düzenleyici ile birlikte kullanılacak tek harf “s” değildir. Python’da farklı amaçlara hizmet eden, bunun gibi başka harfler de bulunur. İşte bu kısımda bu harflerin neler olduğunu ve bunların ne işe yaradığını inceleyeceğiz.

### 17.1.1 “d” Harfi

Yukarıda gördüğümüz “s” harfi nasıl karakter dizilerini temsil ediyorsa, “d” harfi de sayıları temsil eder. İsterseniz küçük bir örnekle açıklamaya çalışalım durumu:

```
>>> print "Şubat ayı bu yıl %d gün çekiyor" %28
```

```
Şubat ayı bu yıl 28 gün çekiyor.
```

Gördüğünüz gibi, karakter dizisi içinde “%s” yerine bu defa “%d” gibi bir şey kullandık. Buna uygun olarak da dış tarafta “28” sayısını kullandık. Peki yukarıdaki ifadeyi şöyle de yazamaz mıyız?

```
>>> print "Şubat ayı bu yıl %s gün çekiyor" %28
```

Elbette yazabiliriz. Bu kod da bize doğru çıktı verecektir. Çünkü daha önce de dediğimiz gibi, “s” harfi karakter dizilerini ve **karakter dizisine çevrilebilen değerleri** temsil eder. Python’da sayılar karakter dizisine çevrilebildiği için “%s” gibi bir yapıyı hata almadan kullanabiliyoruz. Ama mesela şöyle bir şey yapamayız:

```
>>> print "Şubat ayı bu yıl %d gün çekiyor" %"yirmi sekiz"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: %d format: a number is required, not str
```

Gördüğünüz gibi bu defa hata aldık. Çünkü “d” harfi yalnızca sayı değerleri temsil edebilir. Bu harfle birlikte karakter dizilerini kullanamayız.

Doğrusunu söylemek gerekirse, “d” harfi aslında tamsayı (integer) değerleri temsil eder. Eğer bu harfin kullanıldığı bir karakter dizisinde değer olarak mesela bir kayan noktalı sayı (float) verirsek, bu değer tamsayıya çevrilecektir. Bunun ne demek olduğunu hemen bir örnekle görelim:

```
>>> print "%d" %13.5
```

```
13
```

Gördüğünüz gibi, “%d” ifadesi, “13.5” sayısının ondalık kısmını çıktıda göstermiyor. Çünkü “d” harfi sadece tamsayıları temsil etme işlevi görüyor...

### 17.1.2 “i” Harfi

Bu harf de “integer”, yani “tamsayı” kelimesinin kısaltmasıdır. Kullanım ve işlev olarak, “d” harfinden hiç bir farkı yoktur.

### 17.1.3 “o” Harfi

Bu harf “octal” (sekizlik) kelimesinin kısaltmasıdır. Adından da anlaşılacağı gibi, sekizlik düzendeki sayıları temsil eder. Örneğin:

```
>>> print "%i sayısının sekizlik düzendeki karşılığı %o sayıdır." %(10, 10)
```

```
10 sayısının sekizlik düzendeki karşılığı 12 sayıdır.
```

### 17.1.4 “x” Harfi

Bu harf “hexadecimal”, yani onaltılık düzendeki sayıları temsil eder:

```
>>> print "%i sayısının onaltılık düzendeki karşılığı %x sayıdır." %(20, 20)
```

20 sayısının onaltılık düzendeki karşılığı 14 sayıdır.

Buradaki “x” küçük harf olarak kullanıldığında, onaltılık düzende harfle gösterilen sayılar da küçük harfle temsil edilecektir:

```
>>> print "%i sayısının onaltılık düzendeki karşılığı %x sayıdır." %(10, 10)
```

10 sayısının onaltılık düzendeki karşılığı a sayıdır.

### 17.1.5 “X” Harfi

Bu da tıpkı “x” harfinde olduğu gibi, onaltılık düzendeki sayıları temsil eder. Ancak bunun farkı, harfle gösterilen onaltılık sayıları büyük harfle temsil etmesidir:

```
>>> print "%i sayısının onaltılık düzendeki karşılığı %X sayıdır." %(10, 10)
```

10 sayısının onaltılık düzendeki karşılığı A sayıdır.

### 17.1.6 “f” Harfi

Python’da karakter dizilerini biçimlendirirken “s” harfinden sonra en çok kullanılan harf “f” harfidir. Bu harf İngilizce’deki “float”, yani “kayan noktalı sayı” kelimesinin kısaltmasıdır. Adından da anlaşılacağı gibi, karakter dizileri içindeki kayan noktalı sayıları temsil etmek için kullanılır...

```
>>> print "Dolar %f TL olmuş..." %1.4710
```

Dolar 1.471000 TL olmuş...

Eğer yukarıdaki komutun çıktısı sizi şaşırttıysa okumaya devam edin. Biraz sonra bu çıktıyı istediğimiz kıvama nasıl getirebileceğimizi inceleyeceğiz...

### 17.1.7 “c” Harfi

Bu harf de Python’daki önemli karakter dizisi biçimlendiricilerinden biridir. Bu harf tek bir karakteri temsil eder:

```
>>> print "%c" %"a"
```

a

Ama:

```
>>> print "%c" %"istihza"
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: %c requires int or char
```



Gördüğünüz gibi, “c” harfi sadece tek bir karakteri kabul ediyor. Karakter sayısı birden fazla olduğunda bu komut hata veriyor.

“c” harfinin bir başka özelliği de ASCII tablosunda sayılara karşılık gelen karakterleri de gösterebilmesidir:

```
>>> print "%c" %65
```

A

ASCII tablosunda 65 sayısı “A” harfine karşılık geldiği için yukarıdaki komutun çıktısı “A” harfini gösteriyor. Eğer isterseniz “c” harfini kullanarak bütün ASCII tablosunu ekrana dökebilirsiniz:

```
>>> for i in range(128):  
...     "%s ==> %c" %(i, i)
```

Eğer bu ASCII tablosuna tam anlamıyla “tablovari” bir görünüm vermek isterseniz şöyle bir şey yazabilirsiniz:

```
a = ["%s = %c" %(i, i) for i in range(32, 128)]  
  
for i in range(0, 120, 7):  
    a.insert(i, "\n")  
  
for v in a:  
    print v.rjust(8),
```

Burada yaptığımız şey şu:

Öncelikle bir liste üretici (list comprehension) kullanarak 32 ile 128 arasındaki sayılara karşılık gelen harfleri bir liste haline getiriyoruz. Eğer oluşan bu “a” adlı listeyi ekrana yazdırırsanız, liste öğelerinin şöyle bir biçime sahip olduğunu görürsünüz:

```
['32 = ', '33 = !', '34 = "', '35 = #', '36 = $', '37 = %', '38 = &', ...]
```

Bu 127’ye kadar devam eder...

Ardından bu listenin her 6. öğesinden sonra (yani 7. sıraya) bir adet “\n” karakteri yerleştiriyoruz. Bu sayede listeyi her 7. öğede bir alt satıra geçecek şekilde ayarlamış oluyoruz. `for i in range(0, 120, 7)` satırı içindeki 120 sayısını deneme yanılma yoluyla bulabilirsiniz veya bu sayıyı tespit edebilmek için kendinizce bir formül de üretebilirsiniz. Ancak formül üretmeye üşenenler için deneme yanılma yöntemi daha cazip gelecektir! Burada amacımız listedeki her 7. öğeyi bulurken liste sonuna kadar ulaşabilmek... Ben burada bu sayıyı biraz yüksek tuttum, ki tablo ekrana basıldıktan sonra bir satır fazladan boşluk olsun... Eğer isterseniz yukarıdaki kodları şu şekilde yazarak `a.insert(i, "\n")` satırının listenin hangi noktalarına “\n” karakteri yerleştirdiğini inceleyebilirsiniz:

```
a = ["%s = %c" %(i, i) for i in range(32, 128)]  
  
for i in range(0, 120, 7):  
    a.insert(i, "\n")  
  
print a
```

Ayrıca 120 yerine farklı sayılar koyarak hangi sayının yetersiz kaldığını, hangi sayının fazla geldiğini de inceleyebilirsiniz...

Son olarak da, karakter dizisi metotlarını işlerken gördüğümüz “`rjust()`” adlı metot yardımıyla, tabloya girecek karakterleri 8 karakterlik bir alan içinde sağa yaslayarak ekrana döktük.

## 17.2 İleri Düzeyde Karakter Dizisi Biçimlendirme

Python'da karakter biçimlendirmenin amacı birtakım kodlar yardımıyla elde edilen verilerin istediğimiz bir biçimde kullanıcıya sunulabilmesini sağlamaktır. Karakter biçimlendirme genel olarak karmaşık verilerin son kullanıcı tarafından en kolay şekilde anlaşılabilmesini sağlamak için kullanılır. Örneğin şöyle bir sözlük olsun elimizde:

```
>>> stok = {"elma": 10, "armut": 20, "patates": 40}
```

Stokta kaç kilo elma kaldığını öğrenmek isteyen kullanıcıya bu bilgiyi şu şekilde verebiliriz:

```
>>> mal = "elma"
>>> print stok[mal]
```

```
10
```

Ancak bundan daha iyisi, çıktıyı biraz biçimlendirerek kullanıcıya daha “temiz” bir bilgi sunmaktır:

```
>>> mal = "elma"
>>> print "Stokta %d KG %s kaldı!" %(stok["elma"], mal)
```

```
Stokta 10 KG elma kaldı!
```

Ya da mevcut stokların genel durumu hakkında bilgi vermek için şöyle bir yol tercih edebilirsiniz:

```
stok = {"elma": 10, "armut": 20, "patates": 40}

print "stok durumu:"
for k, v in stok.items():
    print "%s\t=\t%s KG"%(k, v)
```

Burada öncelikle stok sözlüğümüzü tanımladık. Daha sonra `for k, v in stok.items()` satırı ile, stok sözlüğünün bütün öğelerini teker teker “k” ve “v” adlı iki değişkene atadık. Böylece sözlük içindeki anahtarlar “k” değişkenine; değerler ise “v” değişkenine atanmış oldu... Son olarak da `print "%s\t=\t%s KG"%(k, v)` satırıyla bu “k” ve “v” değişkenlerini örneğin “elma = 10 KG” çıktısını verecek şekilde biçimlendirip ekrana yazdırdık. Burada meyve adlarıyla meyve miktarları arasındaki mesafeyi biraz açmak için “\t” adlı kaçış dizisinden yararlandığımıza dikkat edin.

Yukarıdaki örnekler, karakter dizisi biçimlendirme kavramının en temel kullanımını göstermektedir. Ancak isterseniz Python'daki karakter dizisi biçimlendiricilerini kullanarak daha karmaşık işlemler de yapabilirsiniz...

### 17.2.1 Karakter Dizisi Biçimlendirmede Sözlükleri Kullanmak

Aslında Python'daki karakter dizisi biçimlendiricilerinin, yukarıda verdiğimiz örneklerde görüldüğünden daha karmaşık bir sözdizimi vardır. Mesela şu örneğe bir bakalım:

```
>>> print "Ben %(isim)s %(soyisim)s" %{"isim": "Fırat", "soyisim": "Özgül"}
```

Buradaki yapı ilk anda gözünüze karmaşık gelmiş olabilir. Ancak aslında oldukça basittir. Üstelik bu yapı epey kullanışlıdır ve bazı durumlarda işlerinizi bir hayli kolaylaştırabilir. Burada yaptığımız şey, “%s” adlı karakter dizisi biçimlendiricisindeki “%” ve “%s” karakterleri arasına

değişken adları yerleştirmekten ibarettir. Burada belirttiğimiz değişken adlarını daha sonra karakter dizisi dışında bir sözlük olarak tanımlıyoruz.

Bir de şu örneğe bakalım:

```
#-*- coding: utf-8 -*-

randevu = {"gun_sayi": 13,
           "ay": "Ocak",
           "gun": u"Çarşamba",
           "saat": "17:00"}

print u"%(gun_sayi)s %(ay)s %(gun)s %(saat)s'da buluşalım!" %randevu
```

Tabii eğer isterseniz sözlüğünüzü doğrudan karakter dizisini yazarken de tanımlayabilirsiniz:

```
#-*- coding: utf-8 -*-

print u"%(gun_sayi)s %(ay)s %(gun)s %(saat)s'da buluşalım!" %{"gun_sayi": 13,
                                                             "ay": "Ocak",
                                                             "gun": u"Çarşamba",
                                                             "saat": "17:00"}
```

Kodları bu şekilde yazdığımızda karakter dizisi dışında “%” işaretinden sonra demet yerine sözlük kullandığımıza dikkat edin.

Python’un bize sunduğu bu yapı karakter dizisi içindeki değişkenlerin neler olduğunu takip edebilmek açısından epey kullanışlı bir araçtır.

## 17.2.2 Sayılarda Hassas Biçimlendirme

Yukarıda “f” adlı biçimlendiriciden bahsederken hatırlarsanız şöyle bir örnek vermiştik:

```
>>> print "Dolar %f TL olmuş..." %1.4710

Dolar 1.471000 TL olmuş...
```

Burada karakter dizisinin dışında belirttiğimiz sayı 1.4710 olduğu halde çıktıda elde ettiğimiz sayı 1.471000... Gördüğünüz gibi, elde ettiğimiz sayı tam istediğimiz gibi değil. Ama eğer arzu edersek bu çıktıyı ihtiyaçlarımıza göre biçimlendirme imkanına sahibiz. Bu bölümde, Python’daki sayıları nasıl hassas biçimlendireceğimizi inceleyeceğiz. Küçük bir örnekle başlayalım:

```
>>> "%f" %1.4

'1.400000'
```

Gördüğünüz gibi, noktadan sonra istediğimizden daha fazla sayıda basamak var. Diyelim ki biz noktadan sonra sadece 2 basamak olsun istiyoruz:

```
>>> "%.2f" %1.4

'1.40'
```

Burada yaptığımız şey, “%” işareti ile “f” karakteri arasına bir adet nokta (.) ve istediğimiz basamak sayısını yerleştirmekten ibaret... Bir de şu örneğe bakalım:

```
>>> "%.2f" %5
'5.00'
```

Gördüğünüz gibi, bu özelliği kayan noktalı sayıların yanısıra tamsayılara (integer) da uygulayabiliyoruz.

### 17.2.3 Sayıların Soluna Sıfır Eklemek

Bir önceki bölümde “%s” işareti ile “f” harfi arasına özel öğeler yerleştirerek yaptığımız şey “f” harfi dışındaki karakterler üzerinde farklı bir etki doğurur. Lafı dolandırıp kafa karıştırmak yerine isterseniz basit bir örnek verelim. Hatırlarsanız “f” harfi ile şöyle bir şey yazabiliyorduk:

```
>>> print "Bugünkü dolar kuru: %.3f" %1.49876
Bugünkü dolar kuru: 1.499
```

Burada yazdığımız “.3” ifadesi, noktadan sonra sadece 3 basamaklık bir hassasiyet istediğimizi gösteriyor. Ama bir de şu örneğe bakın:

```
>>> print "Bugünkü dolar kuru: %.3d" %1.49876
Bugünkü dolar kuru: 001
```

Gördüğünüz gibi, “f” yerine “d” karakterini kullandığımızda “.3” gibi bir ifade, sayıların sol tarafını sıfırlarla dolduruyor. Burada dikkat ederseniz, çıktıda üç adet 0 yok. “.3” ifadesinin yaptığı şey, toplam üç basamaklı bir tamsayı oluşturmaktır. Eğer tamsayı normalde tek basamaklı ise, bu ifade o tek basamaklı sayının soluna iki sıfır koyarak basamak sayısını 3’e tamamlar. Eğer sayı 3 veya daha fazla sayıda basamaktan oluşuyorsa, “.3” ifadesinin herhangi bir etkisi olmaz... Bu yapının tam olarak ne işe yaradığını anlamak için farklı sayılarla denemeler yapmanızı tavsiye ederim.

Bu özelliği şöyle bir iş için kullanabilirsiniz:

```
>>> for i in range(11):
...     print "%.3d" %i
...
000
001
002
003
004
005
006
007
008
009
010
```

Yukarıdaki özelliğin, daha önce karakter dizisi metotlarını işlerken öğrendiğimiz `zfill()` metoduna benzediğini farketmişsinizdir. Aynı şeyi `zfill()` metodunu kullanarak şöyle yapıyorduk:

```
>>> for i in range(11):
...     print str(i).zfill(3)
...
000
001
```

```
002
003
004
005
006
007
008
009
010
```

## 17.2.4 Karakter Dizilerini Hizalamak

Hatırlarsanız karakter dizilerinin metotlarını incelerken `rjust()` adlı bir metottan bahsetmiştik. Bu metot karakter dizilerini sağa yaslamamızı sağlıyordu... Daha önce bu metodu kullandığımız şöyle bir örnek vermiştik:

```
a = ["%s = %c" %(i, i) for i in range(32, 128)]

for i in range(0, 120, 7):
    a.insert(i, "\n")

for v in a:
    print v.rjust(8),
```

Burada `print v.rjust(8)`, satırı, oluşturduğumuz tablodaki öğelerin sağa yaslanmasını sağlamıştı. Aslında aynı etkiyi, biçim düzenleyiciler yardımıyla da sağlayabiliriz:

```
a = ["%s = %c" %(i, i) for i in range(32, 128)]

for i in range(0, 120, 7):
    a.insert(i, "\n")

for v in a:
    print "%8s" %v,
```

İsterseniz konuyu daha iyi anlayabilmek için daha basit bir örnek verelim:

```
>>> for i in range(20):
...     print "%2d" %i
...
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
```

```
17
18
19
```

Dikkat ederseniz burada 10'dan önceki sayılar sağa yaslanmış durumda. Yukarıdaki kodları herhangi bir biçimlendirici içermeyecek şekilde yazarsanız farkı daha net görebilirsiniz:

```
>>> for i in range(11):
...     print i
...
0
1
2
3
4
5
6
7
8
9
10
```

Farkı görüyorsunuz.. Bu defa sayılar normal bir şekilde sola yaslanmış durumda.

Yine bu biçimlendiricileri kullanarak karakter dizileri arasında belli sayıda boşluklar da bırakabilirsiniz. Mesela:

```
>>> print "merhaba %10s dünya" %""
merhaba           dünya
```

Buna benzer bir şeyi kaçış dizilerinden biri olan “\t” ile de yapabiliyoruz. Ancak “\t”yi kullandığımızda boşluk miktarının tam olarak ne kadar olacağını kestiremeyiz. Karakter dizisi biçimlendiricileri ise bize bu konuda çok daha fazla kontrol imkanı verir.

### 17.2.5 Karakter Dizilerini Hem Hizalamak Hem de Sola Sıfır Eklemek

Yukarıda karakter dizilerini nasıl hizalayacağımızı ve sol taraflarına nasıl sıfır ekleyebileceğimizi ayrı ayrı gördük. Esasında Python’daki biçim düzenleyiciler bize bu işlemlerin her ikisini birlikte yapma imkanı da tanır. Hatırlarsanız bir karakter dizisinin sol tarafına sıfır eklemek için şöyle bir şey yapıyorduk:

```
>>> for i in range(11):
...     print "%.3d" %i
```

Karakter dizilerini hizalamak için ise şöyle bir şey..

```
>>> for i in range(20):
...     print "%2d" %i
```

Karakter dizilerini hem hizalamak hem de sol taraflarına sıfır eklemek için de şöyle bir şey yapıyoruz:

```
>>> for i in range(11):
...     print "%3.2d"%i
```

Gördüğünüz gibi nokta işaretinden önce getirdiğimiz sayı karakter dizisinin hizalanma miktarını gösterirken, noktadan sonra getirdiğimiz sayı karakter dizisinin sol tarafına eklenecek sıfırların sayısını gösteriyor.

Eğer yukarıdaki örneklerde “f” adlı biçimlendirme karakterini kullanırsanız, sonuç biraz farklı olacaktır:

```
>>> print "kur: %8.3f" %1.2
kur:      1.200
```

Bu işaret ve karakterlerin tam olarak nasıl bir etki oluşturduğunu daha iyi anlayabilmek için kendi kendinize örnekler yapmanızı tavsiye ederim...

Böylece Python’daki önemli bir konuyu daha geride bırakmış olduk. Aslında burada söylenecek birkaç şey daha olsa da yukarıdakilerin Python programları yazarken en çok ihtiyacınız olacak bilgiler olduğunu gönül rahatlığıyla söyleyebiliriz...

## 17.3 Bölüm Soruları

1. Elimizde şöyle bir liste var:

```
stok = ["elma", "armut", "kiraz"]
```

Bu listede bulunan öğeleri kullanarak şu cümleyi kurun:

```
Stokta bulunan ürünler: elma, armut, kiraz
```

2. Kullanıcıya, banka hesabında kaç TL’si olduğunu soran bir program yazın. Bu program, kullanıcıdan aldığı TL bilgisini dolara çevirerek 6 haneli bir sayı biçiminde ekrana basabilmeli. Örneğin kullanıcının banka hesabında 1000 TL varsa, programınız ekrana şu bilgiyi basmalı:

```
bankada 001000 TL’niz var! (000657 $)
```

---

# ascii, unicode ve Python

---

---

**Not:** Burada verilen bilgiler, *len() Fonksiyonu ve ascii'nin Laneti* adlı makalede anlatılanlarla birlikte değerlendirilmelidir. Bu belge ve oradaki makale birbirini tamamlayıcı bilgiler içermektedir. Söz konusu makalede temel düzeyde bilgiler verilmekte, burada ise konunun ayrıntılarına inilmektedir.

---

## 18.1 Giriş

Orada burada mutlaka Python'un varsayılan kod çözücüsünün ascii olduğuna dair bir cümle duymuşsunuzdur.

Yine sağda solda Python'un unicode'yi desteklediğine dair bir cümle de duymuş olmalısınız.

O halde bu "ascii" ve "unicode" denen şeylerin ne olduğunu da merak etmişsinizdir.

Ben şimdi size burada işin teknik kısmına bulaşmadan hem "ascii", hem de "unicode" ile ilgili çok kaba bir tarif vereceğim:

"ascii", Türkçe'ye özgü "ş", "ç", "ö", "ğ", "ü" ve "ı" gibi harfleri tanımayan bir karakter kodlama biçimidir.

"unicode" ise Türkçe'ye özgü harflerle birlikte dünya üzerindeki pek çok dile ait harfleri de tanıyabilen karakter kodlama biçimlerini içeren, gelişmiş bir sistemdir.

Esasında "unicode", karakter kodlarını gösteren büyük bir listeden ibarettir. Bu liste içinde dünyadaki (hemen hemen) bütün dillere ait karakterler, simgeler, harfler, vb. yer alır. Aynı şekilde ascii de karakter kodlarını içeren bir listedir. Ancak ascii listesi sadece 128 karakterden oluşurken, unicode yüz binlerce karakteri temsil edebilir.

İşte biz bu makalede, her yerde karşımıza çıkan bu "ascii" ve "unicode" kavramlarını anlamaya, bu kavramların Python'daki yerini ve Python'da ne şekilde kullanıldığını öğrenmeye çalışacağız.

Önce ascii'den başlayalım...



## 18.2 ascii

Ne kadar gelişmiş olurlarsa olsunlar, bugün kullandığımız bilgisayarların anladığı tek şey sayılardır. Yani mesela “karakter”, “harf” ve “kelime” gibi kavramlar bir bilgisayar için hiçbir şey ifade etmez. Örneğin ekranda gördüğümüz “a” harfi özünde sadece bir sayıdan ibarettir. Bilgisayar bunu “a” harfi şeklinde değil, bir sayı olarak görür. Dolayısıyla, ekranda gördüğümüz bütün harfler ve simgeler makine dilinde bir sayı ile temsil edilir. İşte bu her karakteri bir sayıyla temsil etme işlemine “karakter kodlama” (character encoding) adı verilir. Her bir karakter bir sayıya karşılık gelecek şekilde, bu karakterleri ve bunların karşılığı olan sayıları bir araya toplayan sistematik yapıya ise “karakter kümesi” (charset – character set) denir. Örneğin ascii, unicode, ISO-8859, vb. birer karakter kümesidir. Çünkü bunlar her biri bir sayıya karşılık gelen karakterleri bir arada sunan sistematik yapılardır. Biraz sonra bu sistemleri daha detaylı bir şekilde incelediğimizde tam olarak ne demek istediğimizi gayet net anlayacaksınız.

1960’lı yıllara gelinceye kadar, “yazı yazmaya yarayan makineler” üreten her firma, farklı karakterlerin farklı sayılarla temsil edildiği karakter kümeleri kullanıyordu. Bu karmaşıklığın bir sonucu olarak, herhangi bir makinede oluşturulan bir metin başka bir makinede aynı şekilde görüntülenemiyordu. Çünkü örneğin “a” harfi bir makinede falanca sayı ile temsil edilirken, başka bir makinede filanca sayı ile temsil ediliyordu. Hatta aynı üreticinin farklı model makinelerinde dahi bir bütünlük sağlanmış değildi... Dolayısıyla her üretici kendi ihtiyaçlarına uygun olarak farklı bir karakter kümesi kullanıyordu...

Bu dağınıklığı önlemek ve “yazı makinelerinde” bir birlik sağlamak için *American Standards Association* (Amerikan Standartları Birliği) tarafından *American Standard Code for Information Interchange* (Bilgi Alışverişi için Amerikan Standart Kodu) adı altında bir standartlaşma çalışması yürütülmesine karar verildi. İşte “ascii” bu “American Standard Code for Information Interchange” ifadesinin kısaltmasıdır. Bu kelime “askii” şeklinde telaffuz edilir...

İlk sürümü 1963 yılında yayımlanan ascii 7-bitlik bir karakter kümesidir. Dolayısıyla bu küme içinde toplam  $2^7 = 128$  adet karakter yer alabilir, yani bu sistemle sadece 128 adet karakter kodlanabilir. (Eğer “7-bit” kavramı size yabancı ise okumaya devam edin. Biraz sonra bu kavramı açıklayacağız.)

Bu bahsettiğimiz 128 karakteri içeren ascii tablosuna (yani “ascii karakter kümesine”) <http://www.asciitable.com/> adresinden erişebilirsiniz. Buradaki tablodan da göreceğiniz gibi, toplam 128 sayının (0’dan başlayarak) ilk 33 tanesi ekranda görünmeyen karakterlere ayrılmıştır. Bu ilk 33 sayı, bir metnin “akışını” belirleyen “sekme”, “yeni satır” ve benzeri karakterleri temsil eder. Ayrıca gördüğünüz gibi, “karakter kümesi” denen şey, temelde belli karakterlerle belli sayıları eşleştiren bir tablodan ibarettir...

Mesela ascii tablosunda “10” sayısının “yeni satır” karakterini temsil ettiğini görüyoruz (O tabloda “dec” sütununa bakın). ascii tablosu bugün için de geçerlidir. Bu yüzden bu tablodaki karakterleri modern sistemler de kabul edecektir. Mesela Python’da, `chr()` fonksiyonundan da yararlanarak şöyle bir şey yazabiliriz:

```
>>> print "Merhaba" + chr(10) + "dünya"
```

```
Merhaba
dünya
```

Gördüğünüz gibi, gerçekten de “10” sayısı yeni bir satıra geçmemizi sağladı. Burada `chr()` fonksiyonu ascii tablosundaki sayıların karakter karşılıklarını görmemizi ve kullanabilmemizi sağlıyor...

Aynı tabloda “9” sayısının ise “sekme” karakterini temsil ettiğini görüyoruz. Bu da bize Python’da şöyle bir şey yazma olanağı sağlıyor:

```
>>> print "Merhaba" + chr(9) + "dünya"
```

```
Merhaba dünya
```

Yukarıda yazdığımız kodları işleten Python, “Merhaba” ve “dünya” kelimeleri arasında bir sekme boşluk bıraktı... Eğer buradaki sekme boşluğunun az olduğunu düşünüyorsanız kodunuzu şöyle de yazabilirsiniz:

```
>>> print "Merhaba" + chr(9) * 2 + "dünya"
```

```
Merhaba   dünya
```

chr(9)’a karşılık gelen sekme karakterini istediğiniz kadar tekrar ederek, karakter dizileri arasında istediğiniz boyutta sekmeler oluşturabilirsiniz... Elbette bunu Python’da “\t” kaçış dizisini kullanarak da yapabileceğinizi biliyorsunuz.

Eğer ascii sisteminden yararlanıp bilgisayarınızın bir “bip” sesi çıkarmasını isterseniz sadece şöyle bir şey yazmanız yeterli olacaktır:

```
>>> print chr(7)
```

```
!bip!
```

Ancak bu biplmeyi (eğer kernel modüllerinden biri olan “pcspkr” yüklü değilse) GNU/Linux sistemlerinde duyamayabilirsiniz...

Elbette ascii tablosu sadece yukarıdaki görünmez karakterlerden ibaret değildir. Tablodan gördüğümüz gibi, 33’ten 128’e kadar olan kısımda ekrana basılabilen karakterler yer alıyor. Mesela “a” harfi “97” sayısı ile gösteriliyor. Dolayısıyla Python’da şöyle bir şey yazabiliyoruz:

```
>>> print "Merhab" + chr(97) + dünya"
```

```
Merhabadünya
```

Burada iki kelime arasında tabii ki boşluk yer almıyor. Boşluk karakterini kendiniz yerleştirebilirsiniz... Ama eğer ascii kodlarını doğrudan karakter dizilerinin içinde kullanmak isterseniz, ascii tablosunda “dec” sütunu yerine “oct” sütunundaki sayıları da kullanabilirsiniz. Tablodaki “oct” sütununu kontrol ettiğimizde “a” harfinin “141” sayısına karşılık geldiğini görüyoruz. O halde şöyle bir kod yazmamız mümkün:

```
>>> print "Merhab\141 dünya"
```

```
Merhaba dünya
```

Burada “141” sayısını doğrudan kullanmadığımıza dikkat edin. Python’un ekrana “141” yazdırmak istediğimizi zannetmemesi için “\” kaçış dizisini kullanarak, Python’a “o 141 bildiğin 141 değil,” gibi bir mesaj vermiş oluyoruz...

“Yok, ben illa dec sütunundaki ondalık sayıları kullanmak istiyorum,” diyorsanız sizi kıracak değiliz. Bunu elbette şu şekilde yapabileceğinizi biliyorsunuz:

```
>>> print "Merhab%s dünya" %chr(97)
```

Dikkat ederseniz, ascii tablosunda “Hx” (Hexadecimal) adlı bir sütun daha var. Eğer bu sütundaki sayılar size daha sevimli geliyorsa şöyle bir yazım tarzı benimseyebilirsiniz:

```
>>> print "Merhab\x61 dünya"
```

ascii tablosunu incelediğinizde, bu tabloda bazı şeylerin eksik olduğu gözünüze çarpmış olmalı. Mesela bu tabloda “çşöğü” gibi harfleri göremiyoruz...

Şimdi Python’da şöyle bir kod yazdığımızı düşünelim:

```
print "Türkçe karakterler: çşöğüİ"
```

Eğer bu kodları doğrudan etkileşimli kabuk üzerinde çalıştırmışsanız Python size düzgün bir çıktı vermiş olabilir. Ancak bu sizi yanıltmasın. Etkileşimli kabuk pratik bir araçtır, ama her zaman güvenilir bir araç olduğu söylenemez. Burada aldığınız çıktıların her ortamda aynı çıktığına vereceğinden emin olamazsınız. O yüzden asıl programlarınızda doğrudan yukarıdaki gibi bir kod yazamazsınız.

Bu kodu bir metin dosyasına yazıp kaydettikten sonra programı çalıştırmak istediğimizde şöyle bir çıktı alırız:

```
SyntaxError: Non-ascii character '\xfc' in file deneme.py on line 1, but no encoding declared; see http://www.python.org/peps/pep-0263.html for details
```

Burada gördüğünüz gibi, Python “ascii olmayan” karakterler kullandığımızdan yakınıyor... Aslında bu çok doğal. Çünkü dediğimiz ve gördüğümüz gibi, “çşöğüİ” harfleri ascii tablosunda yer almıyor...

Bu yazının “Giriş” bölümündeki kaba tarifte de ifade ettiğimiz gibi, “ascii” Türkçe karakterleri tanımayan bir karakter kodlama biçimidir. Aslında ascii sadece Türkçe’yi değil, İngilizce dışındaki hiç bir dilin özel karakterlerini tanımaz.

Python varsayılan olarak ascii kodlamasını kullanır... Hemen bunu teyit edelim:

```
>>> import sys
>>> sys.getdefaultencoding()

'ascii'
```

Demek ki Python’daki varsayılan kodlama biçimi hakikaten “ascii” imiş... Yukarıdaki durum nedeniyle, Python “ascii” tablosunda yer almayan bir karakterle karşılaştığı zaman doğal olarak hata mesajında gördüğümüz çıktıyı veriyor.

Bu arada, ascii’nin 7-bitlik bir sistem olduğunu söylemiştik. Peki “7-bitlik” ne demek? Hemen kısaca onu da açıklayalım:

Bildiğiniz gibi bilgisayarların temelinde 1’ler ve 0’lar yatar. Bilgisayarlar yalnızca bu sayıları anlayıp bu sayılarla işlem yapabilir. Bu 1 ve 0’ların her birine “bit” adı verilir. Bu 1 ve 0’ların 7 tanesi yan yana gelince 7-bitlik bir sayı oluşturur! Gayet mantıklı, değil mi?

İşte ascii sisteminde tanımlanan bütün karakterler 7-bitlik bir sayı içine sığdırılabilir. Yani 7 adet 1 ve 0 bütün ascii karakterleri temsil etmeye yetiyor. Mesela “F” harfinin ascii karşılığı olan “70” sayısı ondalık bir sayıdır. Bu ondalık sayı ikili düzende 1000110 sayısına karşılık gelir. Python’da herhangi bir ondalık sayının ikili düzende hangi sayıya karşılık geldiğini bulmak için bin() adlı fonksiyondan yararlanabilirsiniz:

```
>>> bin(70)
'0b1000110'
```

Bu sayının başındaki “0b” karakterleri Python’un ikili sayıları göstermek için kullandığı bir araçtır. Bu araç Python’a bir şeyler ifade ediyor olabilir, ama bize bir şey ifade etmez. O yüzden bu sayıda bizi ilgilendiren kısım “1000110”.

Gördüğünüz gibi burada toplam 7 adet 1 ve 0 var... Başka bir örnek verelim: ascii tablosunda “r” harfi “114” sayısına karşılık geliyor. Bu ondalık sayının ikili düzendeki karşılığı 1110010 sayıdır:

```
>>> bin(114)
'0b1110010'
```

Yine gördüğünüz gibi, bu ikili sayı da 7 adet 1 ve 0'dan oluşuyor. Dolayısıyla bu da 7-bitlik bir sayıdır. 7-bitle gösterilebilecek son sayı 127'dir. Bunun da ikili düzendeki karşılığı 1111111 sayıdır. 128 sayısı ikili düzende 10000000 olarak gösterilir. Gördüğünüz gibi 128'in ikili düzendeki karşılığı olan sayı 7-bit değil. Çünkü burada 7 adet 1 ve 0 yerine 8 adet 1 ve 0 var... Dolayısıyla 7 bit kullanarak 0 sayısı ile birlikte 128 adet ondalık sayıyı gösterebiliyoruz. 129. sayı ancak 8 bit kullanılarak gösterilebiliyor.

1990'lı yılların ortalarına kadar bilgisayar sistemleri sadece bu 7-bitlik veriler ile işlem yapıyordu. Ancak 90'lu yılların ortalarından itibaren bütün bilgisayar sistemleri 8 bitlik verileri de depolayabilmeye başladı. 8. bitin gelişyle birlikte, 7-bitlik sistemin sunduğu 128 karaktere ek olarak bir 128 karakteri daha temsil edebilme olanağı doğdu. Çünkü 8-bitlik sistemler  $2^8 = 256$  adet karakterin kodlanmasına imkan tanır.

128 karakteri destekleyen standart ascii'nin İngilizce dışındaki karakterleri gösterememesi ve 8. bitin sağladığı olanaklar göz önünde bulundurularak ortaya "Genişletilmiş ascii" diye bir şey çıktı. Ancak ilk 128 karakterden sonra gelen 128 karakterin hangi karakterleri temsil edeceği konusu, yani "Genişletilmiş ascii" denen sistem standart bir yapıya hiç bir zaman kavuşamadı. Bilgisayar üreticileri bu 8. biti kendi ihtiyaçlarına göre doldurma yolunu seçtiler. Böylece ortaya "kod sayfası" (codepage) denen bir kavram çıktı.

Özellikle IBM ve Microsoft gibi büyük şirketlerin, kendi ihtiyaçlarına ve farklı ülkelere göre düzenlenmiş farklı kod sayfaları bulunur. Örneğin Microsoft firmasının Türkiye'ye gönderdiği bilgisayarlarda kullandığı, 857 numaralı bir kod sayfası vardır. Bu kod sayfasına <http://msdn.microsoft.com/en-us/library/cc195068.aspx> adresinden ulaşabilirsiniz. Bu adresteki kod sayfasında göreceğiniz gibi, sayılar 32'den başlıyor. ascii tablosunun ilk 32 sayısı ekranda görünmeyen karakterleri temsil ettiği için Microsoft bunları gösterme gereği duymamış. Bu tabloda 128'e kadar olan kısım zaten standart ascii'dir ve bütün kod sayfalarında ilk 128'lik kısım aynıdır. Sonraki 128'lik kısım ise farklı kod sayfalarında farklı karakterlere karşılık gelir... Örneğin buradaki 857 numaralı kod sayfasında ikinci 128'lik kısım özel karakterlere ayrılmıştır. Bu kod sayfası Türkçe'deki karakterleri düzgün gösterebilmek için üretildiğinden bu listede "ş", "ç", "ö", "ğ", "ı" ve "İ" gibi harfleri görebiliyoruz. Ama eğer <http://msdn.microsoft.com/en-us/library/cc195065.aspx> adresindeki 851 numaralı kod sayfasına bakacak olursanız, bu listede ikinci 128'lik kısmın 857 numaralı kod sayfasındakilerden farklı karakterlere ayrıldığını görürsünüz. Çünkü 851 numaralı kod sayfası Yunan alfabesindeki harfleri gösterebilmek için tasarlanmıştır...

Kod sayfaları dışında, az çok standartlaşmış karakter kümeleri de bu dönemde İngilizce dışındaki dillerin karakter ihtiyaçlarını karşılamak için kullanılıyordu. Örneğin "ISO" karakter kümeleri de kod sayfalarına benzer bir biçimde 8. biti olabildiğince standartlaştırmaya çalışan girişimlerdi. ISO-8859 karakter kümeleri bunların en bilinenlerindendir. Bunlar içinde Türkçe'ye ayrılan küme ISO-8859-9 adlı karakter kümesidir. ISO-8859-9 karakter kümesine <http://czyborra.com/charsets/iso8859.html#ISO-8859-9> adresinden ulaşabilirsiniz...

Hatırlarsanız yukarıda chr() adlı bir fonksiyondan bahsetmiştik. Bu fonksiyon kendisine argüman olarak verilen ondalık bir sayının ascii tablosundaki karşılığını gösteriyordu. Python'da bir de bu chr() fonksiyonunun yaptığı işin tam tersini yapan başka bir fonksiyon daha bulunur. Bu fonksiyonun adı ord()'dur. ord() fonksiyonunu şöyle kullanıyoruz:

```
>>> ord("g")
103
```

Demek ki "g" karakterinin ascii tablosundaki karşılığı 103'müş...

Bu chr() ve ord() fonksiyonları farklı platformlarda farklı davranabilir. Örneğin:

```
>>> print ord("ç")
135
>>> print chr(135)
ç
```

Gördüğünüz gibi, `ord("ç")` çıktısı 135 sonucunu veriyor. 135 sayısına karşılık gelen karakter standart ascii'nin dışında kalır. Çünkü dediğimiz gibi standart ascii sadece 128 karakterden oluşur... Dolayısıyla "ç" karakteri ikinci 128'lik kısımda yer alır. Ancak dediğimiz gibi, yukarıdaki sonuç epey yanıltıcıdır. Çünkü bu çıktıyı yalnızca Windows'ta elde edebilirsiniz. Aynı komut GNU/Linux üzerinde hata verecektir. Microsoft firması ikinci 128'lik kısım için kendine özgü bir kod sayfası kullanıyor ve 135 sayısı bu özel kod sayfasında "ç" karakterine karşılık geliyor. İsterseniz bir de öteki Türkçe karakterlerin durumunu kontrol edelim:

```
>>> for i in "şçöğüİ":
...     print i, ord(i)
...
ş 159
ç 135
ö 148
ğ 167
ü 129
ı 141
İ 152
```

Gördüğünüz gibi bütün Türkçe karakterler 128'den büyük. O yüzden bunlar standart ascii tablosuna girmiyor. Ama Microsoft'un kullandığı 857 numaralı kod sayfası sayesinde bu karakterler ikinci 128'lik kısma girebilmiş... Ancak dediğim gibi, yukarıdaki çıktı güvenilir değildir. Bu komutu ben Windows kurulu bir bilgisayarda verdim. Aynı komutu GNU/Linux işletim sisteminde verdiğinizde hiç beklemediğiniz sonuçlar elde edersiniz. Çünkü elbette GNU/Linux işletim sistemleri Microsoft'un 857 numaralı kod sayfasını kullanmıyor... GNU/Linux farklı bir kodlama sistemini kullanır. Bunun ne olduğunu biraz sonra göreceğiz. Şimdilik biz konumuza devam edelim.

Buraya kadar anlattıklarımızdan anlaşılacağı gibi, standart ascii tablosu İngilizce dışındaki dillere özgü karakter ve simgeleri göstermek konusunda yetersizdir. Standart ascii 128 karakterden ibarettir. Genişletilmiş ascii ise toplam 256 karakterden oluşur. Ancak ilk 128 karakterden sonra gelen ikinci 128'lik kısım standart bir yapıya sahip değildir. Her üretici bu ikinci 128'lik kısmı kendi "kod sayfası"ndaki karakterlerle doldurur. Bu yüzden genişletilmiş ascii'ye güvenip herhangi bir iş yapamazsınız.

İşte tam da bu noktada "unicode" denen şey devreye girer. O halde gelelim bu "unicode" mevzuuna...

## 18.3 unicode

Her ne kadar ascii kodlama sistemi İngilizce dışındaki karakterleri temsil etmek konusunda yetersiz kalsa da insanlar uzun süre ascii'yi temel alarak çalıştılar. Yerel ihtiyaçları gidermek için ise 8. bitin sağladığı 128 karakterlik alandan ve üreticilerin kendi kod sayfalarından yararlandı. Ancak bu yaklaşımın yetersizliği gün gibi ortadadır. Bir defa 7 bit bir kenara, 8 bit dahi dünya üzerindeki bütün dillerin karakter ve simgelerini göstermeye yetmez. Örneğin Asya dillerinin alfabelerindeki harf sayısını temsil etmek için 256 karakterlik alan kesinlikle yetmeyecektir ve yetmemiştir de... Ayrıca mesela tek bir metin içinde hem Türkçe hem de Yunanca

karakterler kullanmak isterseniz ascii sizi yarı yolda bırakacaktır. Türkçe için 857 numaralı kod sayfasını (veya iso-8859-9 karakter kümesini) kullanabilirsiniz. Ama bu kod sayfasında Yunanca harfler yer almaz... Onun için 851 numaralı kod sayfasını veya iso-8859-7 karakter kümesini kullanmanız gerekir. Ancak bu iki kod sayfasını aynı metin içinde kullanmanız mümkün değil...

İşte bütün bu yetersizlikler “evrensel bir karakter kümesi” oluşturmanın gerekliliğini ortaya çıkardı. 1987 yılında Xerox firmasından Joseph D. Becker ve Lee Collins ile Apple firmasından Mark Davis böyle bir “evrensel karakter kümesi” oluşturabilmenin altyapısı üzerinde çalışmaya başladı. “unicode” kelimesini ortaya atan kişi Joe Becker’dır. Becker bu kelimeyi “benzersiz (unique), birleşik (unified) ve evrensel (universal) bir kodlama biçimi” anlamında kullanmıştır.

Joseph Becker’in 29 Ağustos 1988 tarihinde yazdığı *Unicode 88* başlıklı makale, unicode hakkındaki ilk taslak metin olması bakımından önem taşır. Becker bu makalede neden unicode gibi bir sisteme ihtiyaç duyulduğunu, ascii’nin İngilizce dışındaki dillere ait karakterleri göstermekteki yetersizliğini ve herkesin ihtiyacına cevap verebilecek bir sistemin nasıl ortaya çıkarılabileceğini anlatır. Bu makaleye <http://unicode.org/history/unicode88.pdf> adresinden ulaşabilirsiniz.

1987 yılından itibaren unicode üzerine yapılan yoğun çalışmaların ardından 1991 yılında hem “Unicode Konsorsiyum” kurulmuş hem de ilk unicode standardı yayımlanmıştır. Unicode ve bunun tarihi konusunda en ayrıntılı bilgiyi <http://www.unicode.org/> adresinden edinebilirsiniz...

İlk unicode standardı 16-bit temel alınarak hazırlanmıştır. Unicode kavramı ilk ortaya çıktığında Joe Becker 16 bit’in dünyadaki bütün dillere ait karakterleri temsil etmeye yeteceğini düşünüyordu. Ne de olsa;

$$2^7 = 128$$

$$2^8 = 256$$

$$2^{16} = 65536$$

Ancak zamanla 16 bit’in dahi tüm dilleri kapsayamayacağı anlaşıldı. Bunun üzerine unicode 2.0’dan itibaren 16-bit sınırlaması kaldırılarak, dünya üzerinde konuşulan dillere ilaveten arkaik dillerdeki karakterler de temsil edilebilme olanağına kavuştu. Dolayısıyla “unicode 16-bitlik bir sistemdir,” yargısı doğru değildir. Unicode ilk çıktığında 16-bitlikti, ama artık böyle bir sınırlama yok...

Unicode, ascii’ye göre farklı bir bakış açısına sahiptir. ascii’de sistem oldukça basittir. Buna göre her sayı bir karaktere karşılık gelir. Ancak unicode için aynı şeyi söylemek mümkün değil. Unicode sisteminde karakterler yerine “kod konumları” (code points) bulunur. O yüzden unicode sisteminde “baytlar” üzerinden düşünmek yanıltıcı olacaktır...

Örneğin <http://www.unicode.org/charts/PDF/U0100.pdf> adresindeki “Latin Extended A” adlı unicode tablosuna bakalım. Bu tablo Türkçe ve Azerice’nin de dahil olduğu bazı dillere özgü karakterleri barındırır. Bu sayfada yer alan tabloda her karakterin bir kod konumu bulunduğunu görüyoruz. Mesela “ğ” harfinin kod konumu 011F’dır. Unicode dilinde kod konumları geleneksel olarak “U+xxxx” şeklinde gösterilir. Mesela “ğ” harfinin kod konumu “U+011F”dir. Esasında bu “011F” bir onaltılık sayıdır. Bunun ondalık sayı olarak karşılığını bulmak için `int()` fonksiyonundan yararlanabilirsiniz:

```
>>> int("011F", 16)
```

287

Python’da `chr()` fonksiyonuna çok benzeyen `unichr()` adlı başka bir fonksiyon daha bulunur. `chr()` fonksiyonu bir sayının ascii tablosunda hangi karaktere karşılık geldiğini gösteriyordu. `unichr()` fonksiyonu ise bir sayının unicode tablosunda hangi kod konumuna karşılık geldiğini

gösterir. Mesela yukarıda gördüğümüz “287” sayısının hangi kod konumuna karşılık geldiğine bakalım:

```
>>> unichr(287)
```

```
u'\u011f'
```

Gördüğünüz gibi “ğ” harfinin kod konumu olan “011F”yi elde ettik. Burada Python kendi iç işleyişi açısından “011F”yi “u'\u011f” şeklinde gösteriyor. Yukarıdaki kodu şöyle yazarak doğrudan “ğ” harfini elde edebilirsiniz:

```
>>> print unichr(287)
```

```
'ğ'
```

Peki doğrudan bir sayı vererek değil de, karakter vererek o karakterin unicode kod konumunu bulmak mümkün mü? Elbette. Bunun için yine ord() fonksiyonundan yararlanabiliriz:

```
>>> ord(u"ğ")
```

```
287
```

ord() fonksiyonu bir karakterin kod konumunu ondalık sayı olarak verecektir. Elde ettiğiniz bu ondalık sayıyı unichr() fonksiyonuna vererek onaltılık halini ve dolayısıyla unicode kod konumunu elde edebilirsiniz:

```
>>> unichr(287)
```

```
u'\u011f'
```

Bu arada ord() fonksiyonunu burada nasıl kullandığımıza dikkat edin. Sadece ord("ğ") yazmak Python’un hata vermesine sebep olacaktır. Çünkü özünde ord() fonksiyonu sadece 0-256 aralığındaki ascii karakterlerle çalışır. Eğer yukarıdaki kodu ord("ğ") şeklinde yazarsanız, şöyle bir hata alırsınız:

```
>>> ord("ğ")
```

```
ValueError: chr() arg not in range(256)
```

“ğ”nin değeri “287” olduğu ve bu sayı da 256’dan büyük olduğu için ord() fonksiyonu normal olarak bu karakteri gösteremeyecektir. Ama eğer siz bu kodu ord(u"ğ") şeklinde başına bir “u” harfi getirerek yaparsanız mutlu mesut yaşamaya devam edebilirsiniz... Biraz sonra bu “u” harfinin tam olarak ne işe yaradığını göreceğiz. Şimdilik biz yine yolumuza kaldığımız yerden devam edelim.

İsterseniz elimiz alışsın diye “ğ” dışındaki bir Türkçe karakteri de inceleyelim. Mesela “ı” harfini alalım. Bu da yalnızca Türk alfabesinde bulunan bir harftir...

```
>>> ord(u"ı")
```

```
305
```

Demek ki “ı” harfinin unicode kod konumu 305 imiş... Ancak bildiğiniz gibi bu ondalık bir değerdir. Unicode kod konumları ise onaltılık sisteme göre gösterilir. O halde kodumuzu yazalım:

```
>>> unichr(305)
```

```
u'\u0131'
```



Burada Python'un kendi eklediği "u\u" kısmını atarsak "0131" sayısını elde ederiz. Test etmek amacıyla bu "0131" sayısının ondalık karşılığını kontrol edebileceğinizi biliyorsunuz:

```
>>> int("0131", 16)
```

```
305
```

Şimdi eğer <http://www.unicode.org/charts/PDF/U0100.pdf> adresindeki unicode tablosuna bakacak olursanız "ı" harfinin kod konumunun gerçekten de "0131" olduğunu göreceksiniz. İsterseniz bakın...

Doğrudan "ı" harfini elde etmek isterseniz şöyle bir şey de yazabilirsiniz:

```
>>> print u'\u0131'
```

```
ı
```

Unicode sistemi bu kod konumlarından oluşan devasa bir tablodur. Unicode tablosuna <http://www.unicode.org/charts> adresinden ulaşabilirsiniz. Ancak unicode sadece bu kod konumlarından ibaret bir sistem değildir. Zaten unicode sistemi sadece bu kod konumlarından ibaret olsaydı pek bir işe yaramazdı. Çünkü unicode sistemindeki kod konumlarının bilgisayarlarda doğrudan depolanması mümkün değildir. Bilgisayarların bu kod konumlarını işleyebilmesi için bunların bayt olarak temsil edilebilmesi gerekir. İşte unicode denen sistem bir de bu kod konumlarını bilgisayarların işleyebilmesi için bayta çeviren "kod çözücülere" sahiptir.

Unicode sistemi içinde UTF-8, UTF-16 ve UTF-32 gibi kod çözücüler vardır. Bunlar arasında en yaygın kullanılanı UTF-8'dir ve bu kodlama sistemi GNU/Linux sistemlerinde de standart olarak kabul edilir. Python programlama dili de 3.x sürümlerinden itibaren varsayılan kodlama biçimi olarak UTF-8'i kullanır. Hemen bunu teyit edelim.

Python 3.x sürümünde şöyle bir komut verelim:

```
>>> import sys
>>> sys.getdefaultencoding()
```

```
'utf-8'
```

Yukarıda da gösterdiğimiz gibi, Python'un 2.x sürümlerinde bu komutun çıktısı 'ascii' idi... Kod çözücüler her kod konumunu alıp farklı bir biçimde kodlarlar. Ufak bir örnek yapalım:

```
>>> unicode.encode(u"ğ", "utf-8")
```

```
'\xc4\x9f'
```

```
>>> unicode.encode(u"ğ", "utf-16")
```

```
'\xff\xfe\x1f\x01'
```

```
>>> unicode.encode(u"ğ", "utf-32")
```

```
'\xff\xfe\x00\x00\x1f\x01\x00\x00'
```

Buradaki unicode.encode() yapısına kafanızı takmayın. Biraz sonra bunları iyice inceleyeceğiz. Burada benim amacım sadece farklı kod çözücülerin karakterleri nasıl kodladığını göstermek... Dedğim gibi, bu kod çözücüler içinde en gözde olanı utf-8'dir ve bu çözücü GNU/Linux'ta da standarttır.



## 18.4 Python’da unicode Desteği

ascii ve unicode’nin ne olduğunu öğrendik. Şimdi sıra geldi Python’daki unicode desteği konusunu işlemeye...

Esasında unicode konusundan bahsederken bunun Python’a yansımalarının bir kısmını da görmedik değil. Örneğin `unichr()` fonksiyonunu kullanarak bir kod konumunun hangi karaktere karşılık geldiğini bulabileceğimizi öğrendik:

```
>>> print unichr(351)
```

Ş

Ancak daha önce söylediğimiz şeyler parça parça bilgiler içeriyordu. Bu bölümde ise “gerçek hayatta” unicode ve Python’u nasıl bağdaştıracığımızı anlamaya çalışacağız.

Python’da karakterlere ilişkin iki farklı tip bulunur: “karakter dizileri” (strings) ve “unicode dizileri” (unicode strings). Mesela şu bir karakter dizisidir:

```
>>> "elma"
```

Hemen kontrol edelim:

```
>>> type("elma")
```

```
<type 'str'>
```

Şu ise bir unicode dizisidir:

```
>>> u"elma"
```

Bunu da kontrol edelim:

```
>>> type(u"elma")
```

```
<type 'unicode'>
```

Gördüğünüz gibi, unicode dizileri, karakter dizilerinin başına bir “u” harfi getirilerek kolayca elde edilebiliyor.

Peki bir karakter dizisini “unicode” olarak tanımlamanın bize ne faydası var? Bir örnek bin söze bedeldir!.. O yüzden isterseniz derdimizi bir örnekle anlatmaya çalışalım. Şuna bir bakın:

```
>>> kardiz = "ıışık"  
>>> len(kardiz)
```

7

Gördüğünüz gibi, “ışık” kelimesinde sadece dört karakter bulunduğu halde `len()` fonksiyonu “7” sonucunu veriyor... Bu bölümün en başında bahsettiğimiz *makalede* de söylediğimiz gibi `len()` fonksiyonu aslında bir karakter dizisinde kaç karakter olduğuna bakmaz. Bu fonksiyonun ilgilendiği şey o karakter dizisinin içerdiği bayt sayısıdır. “ışık” kelimesini etkileşimli kabukta yazacak olursanız sorunun nerede olduğunu görebilirsiniz:

```
>>> "ıışık"
```

```
'\xc4\xb1\xc5\x9f\xc4\xb1k'
```

Burada “xc4”, “xb1”, “xc5”, “x9f”, “xc4”, “xb1” ve “k” şeklinde gösterilen toplam 7 baytlık bir veri var. Dolayısıyla burada len() fonksiyonunun “4” çıktısı vermesini boşuna beklememek lazım. Bir de şuna bakın:

```
>>> unidiz = u"ıışık"
>>> len(unidiz)
```

```
4
```

Gördüğünüz gibi, “ışık” karakter dizisini unicode olarak tanımladığımızda Python bu dizideki gerçek karakter sayısını verecektir. İsterseniz biraz önce yaptığımız gibi u“ışık” adlı unicode dizisini etkileşimli kabukta yazarak durumu daha net görmeyi tercih edebilirsiniz:

```
>>> u"ıışık"
```

```
u'\u0131\u015f\u0131k'
```

Bu çıktıda görünen “unicode kod konumları”na özellikle dikkat edin. Python bu kod konumlarını birer baytmış gibi görecektir. Eğer unicode tablosunu açıp bakarsanız, bu kod konumlarının hangi harflere karşılık geldiğini görebilirsiniz. Veya daha önce söylediğimiz gibi, şu yolu da kullanabilirsiniz:

```
>>> ord(u"\u0131")
```

```
305
```

```
>>> unichr(305)
```

```
u'\u0131'
```

```
>>> print unichr(305)
```

```
ı
```

Ya da bu üç aşamayı tek adımda halletmek de isteyebilirsiniz:

```
>>> print unichr(ord(u"\u0131"))
```

```
ı
```

Demek ki karakter dizilerini “unicode” olarak tanımlamak programlarımızın çalışması açısından hayati önem taşıyabiliyormuş... Ama tabii bu daha işin ufak bir boyutu. Bir de şuna bakın:

```
>>> kardiz = "ıışık"
```

```
>>> print kardiz.upper()
```

```
ıışık
```

Gayet başarısız bir sonuç! Karakter dizilerinin metotlarından biri olan upper() metodu sadece “k” harfini büyütebildi... Geri kalan harfleri ise olduğu gibi bıraktı. Ama artık siz bu sorunun üstesinden nasıl gelebileceğimizi az çok tahmin ediyorsunuzdur:

```
>>> unidiz = u"ıışık"
```

```
>>> print unidiz.upper()
```

```
İŞİK
```

### 18.4.1 Python Betiklerinde unicode Desteği

Bu bölümün başında ascii konusundan bahsederken, şöyle bir örnek vermiştik:

```
print "Türkçe karakterler: şçöğüıI"
```

Dedik ki, eğer bu satırı bir metin dosyasına yazıp kaydeder ve ardından bu programı çalıştırırsanız şöyle bir hata mesajı alırsınız:

```
SyntaxError: Non-ascii character '\xfc' in file deneme.py on line 1, but no encoding declared; see http://www.python.org/peps/pep-0263.html for details
```

Esasında biz bu hatanın üstesinden nasıl gelebileceğimizi daha önceki derslerimizde edindiğimiz bilgiler sayesinde çok iyi biliyoruz. Python derslerinin ta en başından beri yazdığımız bir satır, bizim yukarıdaki gibi bir hata almamızı engelleyecektir:

```
# -*- coding: utf-8 -*-
```

Hatırlarsanız, unicode konusundan söz ederken unicode sistemi içinde bazı kod çözücülerin yer aldığını söylemiştik. Orada da dediğimiz gibi, “utf-8” de bu kod çözücülerden biri ve en gözde olanıdır. Yukarıdaki satır yardımıyla, yazdığımız programın “utf-8” ile kodlanmasını sağlıyoruz. Böylelikle programımız içinde geçen ascii dışı karakterler çıktıda düzgün gösterilebilecektir. GNU/Linux sistemlerinde “utf-8” kodlaması her zaman işe yarayacaktır. Ancak Windows’ta utf-8 yerine “cp1254” adlı özel bir kod çözücü kullanmanız gerekebilir. Dolayısıyla eğer Windows kullanıyorsanız yukarıdaki satırı şöyle yazmanız gerekebilir:

```
# -*- coding: cp1254 -*-
```

Yazacağınız betiklerin en başına yukarıdaki gibi bir satır koyarak, programlarınıza unicode desteği vermiş oluyorsunuz. Yazdığınız o satır sayesinde Python kendi varsayılan kod çözücüsü olan “ascii”yi kullanarak programınızın çökmesine sebep olmak yerine, sizin belirlediğiniz kod çözücü olan utf-8’i (veya cp-1254’ü) kullanarak programlarınızın içinde rahatlıkla İngilizce dışındaki karakterleri de kullanmanıza müsaade edecektir.

Hatırlarsanız, unicode sistemleri içindeki kod konumlarının bilgisayarlar açısından pek bir şey ifade etmediğini, bunların bilgisayarlarda depolanabilmesi için öncelikle uygun bir kod çözücü yardımı ile bilgisayarın anlayabileceği baytlara dönüştürülmesi gerektiğini söylemiştik. İsterseniz biraz da bunun ne demek olduğunu anlamamıza yardımcı olacak birkaç örnek yapalım.

Biliyoruz ki, Python’da unicode dizileri oluşturmanın en kolay yolu, karakter dizilerinin başına bir adet “u” harfi eklemektir:

```
>>> uni_diz = u"ışık"
```

Bu unicode dizisini etkileşimli kabukta yazdırdığımız zaman şöyle bir çıktı elde ediyoruz:

```
>>> uni_diz
u'\u0131\u015f\u0131k'
```

Burada gördüğümüz şey bir bayt dizisi değildir. Burada gördüğümüz şey bir dizi unicode kod konumudur... Dolayısıyla bu unicode dizisi bu haliyle bilgisayarda depolanamaz. İsterseniz deneyelim:

```
>>> f = open("deneme.txt", "w")
>>> f.write(uni_diz)
```

Bu kodlar şöyle bir hata verecektir:

```
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-2:
ordinal not in range(128)
```

Python, unicode kod konumlarından oluşmuş bir diziyi bilgisayara kaydetmek istediğimizde olanca iyi niyetiyle bu kod konumlarını bilgisayarın anlayabileceği bir biçime dönüştürmeye çalışır. Bunu yaparken de tabii ki varsayılan kod çözücüsü olan “ascii”yi kullanmayı dener. Eğer “uni\_diz” adlı değişkende sadece İngilizce karakterler olsaydı, Python bunları ascii yardımıyla bilgisayarın anlayabileceği bir biçime çevirir ve rahatlıkla dosyaya yazdırabilirdi. Ancak burada bulunan Türkçe karakterler nedeniyle ascii kod çözücüsü bu karakterleri çözemez ve programın çökmesine yol açar... Bu yüzden bizim yapmamız gereken şey, unicode dizilerini depolayabilmek için öncelikle bunları uygun bir kod çözücü yardımıyla bilgisayarın anlayabileceği bir biçime getirmektir. Biraz sonra bunu nasıl yapacağımızı göreceğiz. Ama biz şimdilik yine yolumuza devam edelim.

ascii, unicode ve bunların Python’la ilişkisi konusunda temel bilgileri edindiğimize göre Python’daki unicode desteğinin biraz daha derinlerine inmeye başlayabiliriz. Bunun için isterseniz öncelikle unicode ile ilgili metotlara bir göz atalım.

## 18.4.2 unicode() Fonksiyonu

Hatırlarsanız Python’da bir unicode dizisi oluşturmak için karakter dizisinin başına bir adet “u” harfi getirmemizin yeterli olacağını söylemiştik. Aynı şeyi “unicode()” adlı bir fonksiyonu kullanarak da yapabiliriz:

```
>>> unicode("elma")
>>> type(unicode("elma"))

<type 'unicode'>
```

Burada dikkat etmemiz gereken önemli bir nokta var. “unicode()” adlı fonksiyon ikinci (ve hatta üçüncü) bir parametre daha alır. Önce ikinci parametrenin ne olduğuna bakalım:

```
>>> unicode("elma", "utf-8")
```

Burada “elma” adlı karakter dizisini “utf-8” adlı kod çözücüyü kullanarak bir unicode dizisi haline getirdik. “ascii”nin kendisi unicode sistemi içinde yer almasa da, “ascii” tablosunda bulunan karakterler unicode sisteminde de aynı şekilde yer aldığı için, burada kullandığımız “unicode()” fonksiyonu bir karakter dizisini “ascii” ile kodlamamıza da müsaade eder:

```
>>> unicode("elma", "ascii")
```

Esasında eğer unicode() fonksiyonunun ikinci parametresini belirtmezseniz Python otomatik olarak sizin “ascii”yi kullanmak istediğinizi varsayacaktır:

```
>>> unicode("elma")
```

Peki karakter dizilerini bu şekilde kodlamak ne işimize yarar? Bunu anlamak için şöyle bir örnek verelim:

```
>>> a = "ş1r1nga"
>>> print a.upper()

Ş1R1NGA
```

Gördüğünüz gibi, karakter dizilerinin metotlarından biri olan `upper()`, içindeki Türkçe karakterlerden ötürü “şırına” kelimesini büyütemedi. Bu kelimeyi düzgün bir şekilde büyütebilmek için iki yöntem kullanabilirsiniz. Önce basit olanını görelim:

```
>>> a = u"şırına"  
>>> print a.upper()
```

```
ŞIRINA
```

Burada daha en baştan “şırına” karakter dizisini unicode olarak tanımladık. Bu işlemi, karakter dizisinin başına sadece bir “u” harfi getirerek yaptık. Peki ya baştaki karakter dizisini değiştirme imkanımız yoksa ne olacak? İşte bu durumda ikinci yolu tercih edebiliriz:

```
>>> a = "şırına"  
>>> b = unicode(a, "utf-8")  
>>> print b.upper()
```

```
ŞIRINA
```

`unicode()` fonksiyonu karakterleri daha esnek bir biçimde kodlamamızı sağlar. Eğer burada `unicode()` fonksiyonunu ikinci parametre olmadan çağırırsanız hata mesajı alırsınız:

```
>>> b = unicode(a)
```

```
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc5 in position 0:  
ordinal not in range(128)
```

Daha önce de dediğimiz gibi, ikinci parametrenin belirtilmemesi durumunda Python “ascii” kod çözücüsünü kullanacaktır. O yüzden, “ş” ve “ı” harflerini düzgün görüntüleyebilmek için bizim “utf-8” adlı kod çözücüsünü kullanmamız gerekiyor...

`unicode()` adlı fonksiyon üçüncü bir parametre daha alır. Bu parametre, kullanılan kod çözücünün, bazı karakterleri düzgün kodlayamaması durumunda Python’un ne yapması gerektiğini belirler. Bu parametre üç farklı değer alır: “strict”, “ignore” ve “replace”. Hemen bir örnek verelim:

```
>>> b = unicode(a, "ascii", errors = "strict")
```

```
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc5 in position 0:  
ordinal not in range(128)
```

Gördüğünüz gibi, biraz önce aldığımız hatanın aynısını almamıza sebep oldu bu kod... Çünkü “errors” adlı parametrenin varsayılan değeri “strict”tir. Peki “strict” ne anlama geliyor? Eğer “errors” adlı parametreye değer olarak “strict” verirse, kullanılan kod çözücünün düzgün kodlayamadığı karakterlerle karşılaşıldığında Python bize bir hata mesajı gösterecektir. Dediğimiz gibi, “errors”un varsayılan değeri “strict”tir. Dolayısıyla eğer “errors” parametresini hiç kullanmazsanız, ters durumlarda Python size bir hata mesajı gösterir. Bu parametre “strict” dışında “ignore” ve “replace” adlı iki değer daha alabilir. Önce “ignore”nin ne yaptığına bakalım:

```
>>> b = unicode(a, "ascii", errors = "ignore")  
>>> print b.upper()
```

```
RNGA
```

Burada “ignore” değeri, Python’un herhangi bir ters durumda, çözülemeyen karakteri es geçmesini sağlıyor. Bu sebeple Python çözülemeyen karakterleri tamamen ortadan kaldırıyor... Peki “replace” değeri ne işe yapıyor? Hemen bakalım:

```
>>> b = unicode(a, "ascii", errors = "replace")
>>> print b.upper()
```

```
????R??NGA
```

Burada da Python çözölemeyen karakterlerin yerine soru işaretleri yerleştirdi... Eğer Python'un her karakter yerine neden iki soru işareti koyduğunu merak ediyorsanız, bu bölümün en başında bağlantısını verdiğimiz makaleyi inceleyebilirsiniz (ipucu: len() fonksiyonunu kullanarak "şırınga" kelimesindeki Türkçe karakterlerin uzunluğuna bakın...)

Bu anlattığımız konu ile ilgili olarak şöyle basit bir örnek verebiliriz:

```
#-*-coding:utf-8-*-
```

```
a = raw_input("herhangi bir karakter yazınız: ")
print a.upper()
```

Eğer kodlarımızı bu şekilde yazarsak istediğimiz sonucu elde edemeyiz. Python ekranda girdiğimiz Türkçe karakterleri bu şekilde düzgün büyütemeyecektir. Yukarıdaki kodların çalışması için şöyle bir şey yazmalıyız:

```
#-*-coding:utf-8-*-
```

```
a = raw_input("herhangi bir karakter yazınız: ")
print unicode(a, "utf-8", errors = "replace").upper()
```

Böylece unicode'ye duyarlı bir program yazmış olduk. Artık kullanıcılarımız İngiliz alfabesindeki harflerin dışında bir harf girdiklerinde de programımız düzgün çalışacaktır. Kullanıcının, tanınmayan bir karakter girmesi ihtimaline karşılık da "errors" parametresine "replace" değerini verdik...

Yukarıdaki örneğe benzer şekilde, eğer kullanıcılarınızın girdiği karakterlerin sayısına göre işlem yapan bir uygulama yazıyorsanız ya kullanıcılarınızı "ascii" dışında karakter girmemeye zorlayacaksınız, ya da şöyle bir kod yazacaksınız:

```
#-*-coding:utf-8-*-
```

```
a = raw_input("Bir parola belirleyiniz: ")
print "Parolanız %s karakter içeriyor!"%len(unicode(a, "utf-8", errors="replace"))
```

Eğer burada bir unicode dizisi oluşturmazsanız ölçmeye çalıştığınız karakter uzunluğu beklediğiniz gibi çıkmayacaktır.

Şimdi tekrar unicode kod konumlarını dosyaya kaydetme meselesine dönelim. Bu kısımda öğrendiğimiz unicode() fonksiyonu da kod konumlarını dosyaya kaydetmemizi sağlamaya yetmeyecektir:

```
>>> kardiz = "ışık"
>>> unikod = unicode(kardiz, "utf-8")
>>> f = open("deneme.txt", "w")
>>> f.write(unikod)
```

Bu kodlar şu hatayı verir:

```
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-2:
ordinal not in range(128)
```

Çünkü `unicode()` fonksiyonu da aslında bir dönüştürme işlemi yapmaz. Bu fonksiyonun yaptığı şey bir karakter dizisini kod konumları şeklinde temsil etmekten ibarettir... Bunu şu şekilde teyit edebilirsiniz:

```
>>> kardiz = "ışık"
>>> unikod = unicode(kardiz, "utf-8")
>>> unikod

u'\u0131\u015f\u0131k'
```

Burada gördüğümüz şey byte değil `unicode` kod konumlarıdır... `unicode()` fonksiyonu içinde kullandığımız `utf-8` çözücüsünün görevi ise “ışık” kelimesini Python’un doğru okumasını sağlamaktır...

### 18.4.3 `encode()` ve `decode()` Metotları

Hatırlarsanız, `unicode` sisteminde her bir karakterin bir kod konumuna sahip olduğunu söylemiştik. Karakterlerin kod konumlarını ya `unicode` tablolarına bakarak, ya da Python’daki `ord()` ve `unichr()` fonksiyonlarını kullanarak bulabileceğinizi biliyorsunuz. Mesela:

```
>>> unichr(ord(u"ş"))

u'\u015f'
```

Benzer bir şeyi `unicode()` metodunu kullanarak daha da kolay bir şekilde yapabilirsiniz:

```
>>> unicode(u"ş")

u'\u015f'
```

Burada `unicode()` fonksiyonunu tek parametreyle kullandığımıza dikkat edin. Eğer “`utf-8`” veya başka bir kod çözücü kullanarak ikinci parametreyi de girersek Python burada bize bir hata mesajı gösterecektir.

Tekrar tekrar söylediğimiz gibi, bu kod konumu esasında bilgisayarlar için çok fazla şey ifade etmez. Bunların bir şey ifade edebilmesi için bir kod çözücü vasıtasıyla kodlanması gerekir. Mesela bu kod konumunu “`utf-8`” ile kodlamayı tercih edebilirsiniz:

```
>>> kod_konumu = unicode(u"ş")
>>> utfsekiz = kod_konumu.encode("utf-8")
```

Burada `encode()` adlı bir metottan yararlandığımıza dikkat edin.

Böylece `unicode` kod konumunu “`utf-8`” kod çözücüsü ile kodlamış oldunuz. Eğer özgün kod konumunu tekrar elde etmek isterseniz bu kez `decode()` adlı başka bir metottan yararlanabilirsiniz:

```
>>> utfsekiz.decode("utf-8")

u'\u015f'
```

İşte Python’da asıl dönüştürme işlemlerini yapanlar bu `encode()` ve `decode()` adlı metotlardır. Dolayısıyla artık şöyle bir şey yazabilirsiniz:

```
>>> kardiz = "ışık"
>>> unikod = unicode(kardiz, "utf-8")
>>> unibayt = unicode.encode(unikod, "utf-8")
>>> f = open("deneme.txt", "w")
```

```
>>> f.write(unibayt)
>>> f.close()
```

Bu kısmı kısaca özetleyecek olursak şöyle söyleyebiliriz:

Python'da karakterlere ilişkin iki farklı tip bulunur: karakter dizileri ve unicode dizileri

Mesela şu bir karakter dizisidir:

```
>>> kardiz = "elma"
```

Bu karakter dizisi 4 baytlık bir veri içerir:

```
>>> len(kardiz)
```

```
4
```

Şu da bir karakter dizisidir:

```
>>> kardiz = "ışık"
```

Bu karakter dizisi ise 7 baytlık bir veri içerir:

```
>>> len(kardiz)
```

```
7
```

Buradan çıkan sonuca bakarak karakter dizisi ile bayt sayısının aynı şey olmadığını anlamış oluyoruz. Ayrıca `len()` fonksiyonu da bir karakter dizisindeki karakter sayısını değil bayt sayısını veriyor.

"kardiz" adlı değişkeni etkileşimli kabukta yazdırırsak baytları temsil eden karakterleri tek tek görebiliriz:

```
>>> kardiz
```

```
'\xc4\xb1\xc5\x9f\xc4\xb1k'
```

Şu ise bir unicode dizisidir:

```
>>> unidiz = u"elma"
```

Gördüğünüz gibi, Python'da unicode dizisi oluşturmak oldukça kolay. Yapmamız gereken tek şey karakter dizisinin başına bir "u" harfi getirmek.

Şu da bir unicode dizisidir:

```
>>> unidiz = u"ışık"
```

Bu unicode dizisi, karakter dizilerinin aksine baytlardan değil unicode kod konumlarından oluşur. Eğer yukarıdaki unicode dizisini etkileşimli kabukta yazdıracak olursanız bu unicode kod konumlarını görebilirsiniz:

```
>>> unidiz
```

```
u'\u0131\u015f\u0131k'
```

Eğer `len()` fonksiyonunu bu unicode dizisi üzerine uygulayacak olursanız, `len()` fonksiyonu her bir kod konumunu bir baytmış gibi değerlendirecektir:



```
>>> len(unidiz)
```

```
4
```

Kod konumlarının bir işe yarayabilmesi için bunların uygun bir kod çözücü yardımıyla bayta çevrilmeleri gerekir. Unicode sistemi içinde UTF-8, UTF-16 ve UTF-32 gibi kod çözücüler vardır. Bu kod çözücülerin her biri, kod konumlarını farklı bir biçimde ve boyutta bayta çevirir. Bu kod çözücülerinde en esnek ve yaygın olanı utf-8 adlı kod çözücüdür. Python'da kod konumlarını bayta çevirmek için `encode()` adlı bir metottan yararlanabiliriz:

```
>>> unidiz.encode("utf-8")
```

```
'\xc4\xb1\xc5\x9f\xc4\xb1k'
```

Unicode kod konumları bayta çevrildikten sonra artık bir unicode dizisi değil, karakter dizisi olur:

```
>>> type(unidiz.encode("utf-8"))
```

```
<type 'str'>
```

Eğer bir karakter dizisini unicode kod konumları ile temsil etmek isterseniz `decode()` metotundan yararlanabilirsiniz:

```
>>> kardiz.decode("utf-8")
```

```
u'\u0131\u015f\u0131k'
```

Yukarıdaki işlem şununla aynıdır:

```
>>> unicode(kardiz, "utf-8")
```

```
u'\u0131\u015f\u0131k'
```

Böylece Python'daki "unicode" desteğine ilişkin en önemli metotları görmüş olduk. Artık Python'da "unicode"ye ilişkin bazı önemli modülleri de incelemeye geçebiliriz...

## 18.4.4 unicodedata Modülü

Bu modül unicode veritabanına erişerek, bu veritabanındaki bilgileri kullanmamızı sağlar. Küçük bir örnek verelim:

```
>>> import unicodedata
```

```
>>> unicodedata.name(u"ğ")
```

```
'LATIN SMALL LETTER G WITH BREVE'
```

"ğ" harfinin unicode sistemindeki uzun adı "LATIN SMALL LETTER G WITH BREVE"dir. Eğer isterseniz bu ismi ve `lookup()` metodunu kullanarak söz konusu karakterin unicode kod konumuna da ulaşabilirsiniz:

```
>>> unicodedata.lookup("LATIN SMALL LETTER G WITH BREVE")
```

```
u'\u011f'
```

Bir de şuna bakalım:

```
>>> unicodedata.category(u"ğ")  
'L1'
```

`category()` metodu ise bir unicode dizisinin unicode sistemindeki kategorisini gösteriyor. Yukarıda verdiğimiz örneğe göre “ğ” harfinin kategorisi “L1” yani “Latin-1”.

### 18.4.5 codecs Modülü

Bu modül Python’da bir dosyayı, kod çözücüyü de belirterek açmamızı sağlar:

```
import codecs  
f = codecs.open("unicode.txt", "r", encoding="utf-8")  
for i in f:  
    print i
```

Böylece dosya içindeki ascii dışı karakterleri doğru görüntüleyebiliriz...

Ayrıca bu metot, yukarıdaki `encode()` metodunu kullanarak yaptığımız dosyaya yazma işlemini biraz daha kolaylaştırabilir:

```
>>> import codecs  
>>> f = codecs.open("deneme.txt", "w", encoding="utf-8")  
>>> f.write(u"ışık")  
>>> f.close()
```

Elbette `codecs` modülünün `open()` metodunu sadece yeni dosya oluşturmak için değil, her türlü dosya işleminde kullanabilirsiniz. Bu metot, dosya işlemleri konusunu işlerken gördüğümüz `open()` metoduyla birbirine çok benzer. Tıpkı o metotta olduğu gibi `codecs.open()` metodunda da dosyaları “w”, “r”, “a” gibi kiplerde açabilirsiniz.

---

# Düzenli İfadeler (Regular Expressions)

---

Düzenli ifadeler Python programlama dilindeki en çetrefilli konulardan biridir. Hatta düzenli ifadelerin Python içinde ayrı bir dil olarak düşünülmesi gerektiğini söyleyenler dahi vardır. Bütün zorluklarına rağmen programlama deneyimimizin bir noktasında mutlaka karşımıza çıkacak olan bu yapıyı öğrenmemizde büyük fayda var. Düzenli ifadeleri öğrendikten sonra, elle yapılması saatler sürecektir bir işlemi saliseler içinde yapabildiğinizi gördüğünüzde eminim düzenli ifadelerin ne büyük bir nimet olduğunu anlayacaksınız. Tabii her güzel şey gibi, düzenli ifadelerin nimetlerinden yararlanabilecek düzeye gelmek de biraz kan ve gözyaşı istiyor.

Peki, düzenli ifadeleri kullanarak neler yapabiliriz? Çok genel bir ifadeyle, bu yapıyı kullanarak metinleri veya karakter dizilerini parmağımızda oynatabiliriz. Örneğin bir web sitesinde dağınık halde duran verileri bir çırpıda ayıklayabiliriz. Bu veriler, mesela, toplu halde görmek istediğimiz web adreslerinin bir listesi olabilir. Bunun dışında, örneğin, çok sayıda belge üzerinde tek hareketle istediğimiz değişiklikleri yapabiliriz.

Genel bir kural olarak, düzenli ifadelerden kaçabildiğimiz müddetçe kaçmamız gerekir. Eğer Python'daki karakter dizisi metotları, o anda yapmak istediğimiz şey için yeterli geliyorsa mutlaka o metotları kullanmalıyız. Çünkü karakter dizisi metotları, düzenli ifadelere kıyasla hem daha basit, hem de çok daha hızlıdır. Ama bir noktadan sonra karakter dizilerini kullanarak yazdığınız kodlar iyice karmaşıklaşmaya başlamışsa, kodların her tarafı if deyimleriyle dolmuşsa, hatta basit bir işlemi gerçekleştirmek için yazdığınız kod sayfa sınırlarını zorlamaya başlamışsa, işte o noktada artık düzenli ifadelerin dünyasına adım atmanız gerekiyor olabilir. Ama bu durumda Jamie Zawinski'nin şu sözünü de aklınızdan çıkarmayın: *"Bazıları, bir soruna karşı karşıya kaldıklarında şöyle der: 'Evet, düzenli ifadeleri kullanmam gerekiyor.' İşte onların bir sorunu daha vardır artık..."*

Başta da söylediğim gibi, düzenli ifadeler bize zorlukları unutturacak kadar büyük kolaylıklar sunar. Emin olun yüzlerce dosya üzerinde tek tek elle değişiklik yapmaktan daha zor değildir düzenli ifadeleri öğrenip kullanmak... Hem zaten biz de bu sayfalarda bu "sevimsiz" konuyu olabildiğince sevimli hale getirmek için elimizden gelen çabayı göstereceğiz. Sizin de çaba göstermeniz, bol bol alıştırma yapmanız durumunda düzenli ifadeleri kavramak o kadar da zorlayıcı olmayacaktır. Unutmayın, düzenli ifadeler ne kadar uğraştırıcı olsa da programcının en önemli silahlarından biridir. Hatta düzenli ifadeleri öğrendikten sonra onsuz geçen yıllarınıza acıyacaksınız.

Şimdi lafı daha fazla uzatmadan işimize koyulalım.

## 19.1 Düzenli İfadelerin Metotları

Python'daki düzenli ifadelere ilişkin her şey bir modül içinde tutuluyor. Bu modülün adı `re`. Tıpkı `os` modülünde, `sys` modülünde, `Tkinter` modülünde ve öteki bütün modüllerde olduğu gibi, düzenli ifadeleri kullanabilmemiz için de öncelikle bu `re` modülünü içe aktarmamız gerekecek. Bu işlemi nasıl yapacağımızı çok iyi biliyorsunuz:

```
>>> import re
```

Bir önceki bölümde söylediğimiz gibi, düzenli ifadeler bir programcının en önemli silahlarından biridir. Şu halde silahımızın özelliklerine bakalım. Yani bu yapının bize sunduğu araçları şöyle bir listeleyelim. Etkileşimli kabukta şu kodu yazıyoruz:

```
>>> dir(re)
```

Tabii yukarıdaki `dir(re)` fonksiyonunu yazmadan önce `import re` şeklinde modülümüzü içe aktarmış olmamız gerekiyor.

Gördüğünüz gibi, `re` modülü içinde epey metot/fonksiyon var. Biz bu sayfada ve ilerleyen sayfalarda, yukarıdaki metotların/fonksiyonların en sık kullanılanlarını size olabildiğince yalın bir şekilde anlatmaya çalışacağız. Eğer isterseniz, şu komutu kullanarak yukarıdaki metotlar/fonksiyonlar hakkında yardım da alabilirsiniz:

```
>>> help(metot_veya_fonksiyon_adı)
```

Bir örnek vermek gerekirse:

```
>>> help(re.match)
```

Help on function match in module re:

```
match(pattern, string, flags=0)
    Try to apply the pattern at the start of the string,
    returning a match object, or None if no match was found.
```

Ne yazık ki, Python'un yardım dosyaları hep İngilizce. Dolayısıyla eğer İngilizce bilmiyorsanız, bu yardım dosyaları pek işinize yaramayacaktır.

Bu arada yukarıdaki yardım bölümünden çıkmak için klavyedeki `q` düğmesine basmanız gerekir.

### 19.1.1 match() Metodu

Bir önceki bölümde metotlar hakkında yardım almaktan bahsederken ilk örneğimizi `match()` metoduyla vermiştik, o halde `match()` metodu ile devam edelim.

`match()` metodunu tarif etmek yerine, isterseniz bir örnek yardımıyla bu metodun ne işe yaradığını anlamaya çalışalım. Diyelim ki elimizde şöyle bir karakter dizisi var:

```
>>> a = "python güçlü bir programlama dilidir."
```

Varsayalım ki biz bu karakter dizisi içinde "python" kelimesi geçip geçmediğini öğrenmek istiyoruz. Ve bunu da düzenli ifadeleri kullanarak yapmak istiyoruz. Düzenli ifadeleri bu örneğe uygulayabilmek için yapmamız gereken şey, öncelikle bir düzenli ifade kalıbı oluşturup, daha sonra bu kalıbı yukarıdaki karakter dizisi ile karşılaştırmak. Biz bütün bu işlemleri `match()` metodunu kullanarak yapabiliriz:

```
>>> re.match("python",a)
```

Burada, python şeklinde bir düzenli ifade kalıbı oluşturduk. Düzenli ifade kalıpları `match()` metodunun ilk argümanıdır (yani parantez içindeki ilk değer). İkinci argümanımız ise (yani parantez içindeki ikinci değer), hazırladığımız kalıbı kendisiyle eşleştireceğimiz karakter dizisi olacaktır.

Klavyede ENTER tuşuna bastıktan sonra karşımıza şöyle bir çıktı gelecek:

```
<sre.SRE_Match object at 0xb7d111e0>
```

Bu çıktı, düzenli ifade kalıbının karakter dizisi ile eşleştiği anlamına geliyor. Yani aradığımız şey, karakter dizisi içinde bulunmuş. Python bize burada ne bulunduğunu söylemiyor. Bize söylediği tek şey, aradığımız şeyi buldu. Bunu söyleme tarzı da yukarıdaki gibi... Yukarıdaki çıktıda gördüğümüz ifadeye Python’cda eşleşme nesnesi (*match object*) adı veriliyor. Çünkü `match()` metodu yardımıyla yaptığımız şey aslında bir eşleştirme işlemidir (*match* kelimesi İngilizce’de eşleşmek anlamına geliyor). Biz burada python düzenli ifadesinin `a` değişkeniyle eşleşip eşleşmediğine bakıyoruz. Yani `re.match("python",a)` ifadesi aracılığıyla python ifadesi ile `a` değişkeninin tuttuğu karakter dizisinin eşleşip eşleşmediğini sorguluyoruz. Bizim örneğimizde python `a` değişkeninin tuttuğu karakter dizisi ile eşleştiği için bize bir eşleşme nesnesi döndürülüyor. Bir de şu örneğe bakalım:

```
>>> re.match("Java",a)
```

Burada ENTER tuşuna bastığımızda hiç bir çıktı almıyoruz. Aslında biz görmesek de Python burada `None` çıktısı veriyor. Eğer yukarıdaki komutu şöyle yazarsak `None` çıktısını biz de görebiliriz:

```
>>> print re.match("Java",a)
```

```
None
```

Gördüğünüz gibi, ENTER tuşuna bastıktan sonra `None` çıktısı geldi. Demek ki Java ifadesi, `a` değişkeninin tuttuğu karakter dizisi ile eşleşmiyormuş. Buradan çıkardığımız sonuca göre, Python `match()` metodu yardımıyla aradığımız şeyi eşleştirdiği zaman bir eşleşme nesnesi (*match object*) döndürüyor. Eğer eşleşme yoksa, o zaman da `None` değerini döndürüyor.

Biraz kafa karıştırmak için şöyle bir örnek verelim:

```
>>> a = "Python güçlü bir dildir"
>>> re.match("güçlü", a)
```

Burada `a` değişkeninde “güçlü” ifadesi geçtiği halde `match()` metodu bize bir eşleşme nesnesi döndürmedi. Aslında bu normal. Çünkü `match()` metodu bir karakter dizisinin sadece en başına bakar. Yani Python güçlü bir dildir ifadesini tutan `a` değişkenine `re.match("güçlü",a)` gibi bir fonksiyon uyguladığımızda, `match()` metodu `a` değişkeninin yalnızca en başına bakacağı için ve `a` değişkeninin en başında “güçlü” yerine “python” olduğu için, `match()` metodu bize olumsuz yanıt veriyor.

Aslında `match()` metodunun yaptığı bu işi, karakter dizilerinin `split()` metodu yardımıyla da yapabiliriz:

```
>>> a.split()[0] == "python"
```

```
True
```

Demek ki `a` değişkeninin en başında “python” ifadesi varmış. Bir de şuna bakalım:

```
>>> a.split()[0] == "güçlü"
```

```
False
```

Veya aynı işi sadece `startswith()` metodunu kullanarak dahi yapabiliriz:

```
>>> a.startswith("python")
```

Eğer düzenli ifadelerden tek beklentiniz bir karakter dizisinin en başındaki veriyle eşleştirme işlemi yapmaksa, `split()` veya `startswith()` metotlarını kullanmak daha mantıklıdır. Çünkü `split()` ve `startswith()` metotları `match()` metodundan çok daha hızlı çalışacaktır.

`match()` metodunu kullanarak bir kaç örnek daha yapalım:

```
>>> sorgu = "1234567890"
```

```
>>> re.match("1",sorgu)
```

```
<_sre.SRE_Match object at 0xb7d111e0>
```

```
>>> re.match("1234",sorgu)
```

```
<_sre.SRE_Match object at 0xb7d111e0>
```

```
>>> re.match("124",sorgu)
```

```
None
```

İsterseniz şimdiye kadar öğrendiğimiz şeyleri şöyle bir gözden geçirelim:

1. Düzenli ifadeler Python'un çok güçlü araçlarından biridir.
2. Python'daki düzenli ifadelere ilişkin bütün fonksiyonlar `re` adlı bir modül içinde yer alır.
3. Dolayısıyla düzenli ifadeleri kullanabilmek için öncelikle bu `re` modülünü `import re` diyerek içe aktarmamız gerekir.
4. `re` modülünün içeriğini `dir(re)` komutu yardımıyla listeleyebiliriz.
5. `match()` metodu `re` modülü içindeki fonksiyonlardan biridir.
6. `match()` metodu bir karakter dizisinin yalnızca en başına bakar.
7. Eğer aradığımız şey karakter dizisinin en başında yer alıyorsa, `match()` metodu bir eşleştirme nesnesi döndürür.
8. Eğer aradığımız şey karakter dizisinin en başında yer almıyorsa, `match()` metodu `None` değeri döndürür.

Daha önce söylediğimiz gibi, `match()` metodu ile bir eşleştirme işlemi yaptığımızda, eğer eşleşme varsa Python bize bir eşleşme nesnesi döndürecektir. Ama biz bu çıktıdan, `match()` metodu ile bulunan şeyin ne olduğunu göremiyoruz. Ama istersek tabii ki bulunan şeyi de görme imkânımız var. Bunun için `group()` metodunu kullanacağız:

```
>>> a = "perl, python ve ruby yüksek seviyeli dillerdir."
```

```
>>> b = re.match("perl",a)
```

```
>>> print b.group()
```

```
perl
```

Burada, `re.match("perl",a)` fonksiyonunu bir değişkene atadık. Hatırlarsanız, bu fonksiyonu komut satırına yazdığımızda bir eşleşme nesnesi elde ediyorduk. İşte burada değişkene

atadığımız şey aslında bu eşleşme nesnesinin kendisi oluyor. Bu durumu şu şekilde teyit edebilirsiniz:

```
>>> type(b)
<type '_sre.SRE_Match'>
```

Gördüğünüz gibi, b değişkeninin tipi bir eşleşme nesnesi (*match object*). İsterseniz bu nesnenin metotlarına bir göz gezdirebiliriz:

```
>>> dir(b)
```

Dikkat ederseniz yukarıda kullandığımız `group()` metodu listede görünüyor. Bu metot, doğrudan doğruya düzenli ifadelerin değil, eşleşme nesnelerinin bir metodudur. Listedeki diğer metotları da sırası geldiğinde inceleyeceğiz. Şimdi isterseniz bir örnek daha yapıp bu konuyu kapatalım:

```
>>> iddia = "Adana memleketlerin en güzelidir!"
>>> nesne = re.match("Adana", iddia)
>>> print nesne.group()
```

Peki, eşleştirmek istediğimiz düzenli ifade kalıbı bulunamazsa ne olur? Öyle bir durumda yukarıdaki kodlar hata verecektir. Hemen bakalım:

```
>>> nesne = re.match("İstanbul", iddia)
>>> print nesne.group()
```

Hata mesajımız:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'group'
```

Böyle bir hata, yazdığınız bir programın çökmesine neden olabilir. O yüzden kodlarımızı şuna benzer bir şekilde yazmamız daha mantıklı olacaktır:

```
>>> nesne = re.match("İstanbul", iddia)
>>> if nesne:
...     print "eşleşen ifade: %s" %nesne.group()
... else:
...     print "eşleşme başarısız!"
```

Şimdi isterseniz bu `match()` metoduna bir ara verip başka bir metodu inceleyelim.

### 19.1.2 search() Metodu

Bir önceki bölümde incelediğimiz `match()` metodu, karakter dizilerinin sadece en başına bakıyordu. Ama her zaman istediğimiz şey bununla sınırlı olmayacaktır. `match()` metodunun, karakter dizilerinin sadece başına bakmasını engellemenin yolları olmakla birlikte, bizim işimizi görecektir çok daha kullanışlı bir metodu vardır düzenli ifadelerin. Önceki bölümde `dir(re)` şeklinde gösterdiğimiz listeye tekrar bakarsanız, orada `re` modülünün `search()` adlı bir metodu olduğunu göreceksiniz. İşte bu yazımızda inceleyeceğimiz metot bu `search()` metodu olacaktır.

`search()` metodu ile `match()` metodu arasında çok önemli bir fark vardır. `match()` metodu bir karakter dizisinin en başına bakıp bir eşleştirme işlemi yaparken, `search()` metodu karakter

dizisinin genelinde bir arama işlemi yapar. Yani biri eşleştirir, öbürü arar. Zaten *search* kelimesi de İngilizce’de aramak anlamına gelir...

Hatırlarsanız, `match()` metodunu anlatırken şöyle bir örnek vermiştik:

```
>>> a = "Python güçlü bir dildir"
>>> re.match("güçlü", a)
```

Yukarıdaki kod, karakter dizisinin başında bir eşleşme bulamadığı için bize `None` değeri döndürüyordu. Ama eğer aynı işlemi şöyle yaparsak, daha farklı bir sonuç elde ederiz:

```
>>> a = "Python güçlü bir dildir"
>>> re.search("güçlü", a)

<sre.SRE_Match object at 0xb7704c98>
```

Gördüğünüz gibi, `search()` metodu “güçlü” kelimesini buldu. Çünkü `search()` metodu, `match()` metodunun aksine, bir karakter dizisinin sadece baş tarafına bakmakla yetinmiyor, karakter dizisinin geneli üzerinde bir arama işlemi gerçekleştiriyor. Tıpkı `match()` metodunda olduğu gibi, `search()` metodunda da `group()` metodundan faydalanarak bulunan şeyi görüntüleyebiliriz:

```
>>> a = "Python güçlü bir dildir"
>>> b = re.search("güçlü",a)
>>> print b.group()
```

Şimdiye kadar hep karakter dizileri üzerinde çalıştık. İsterseniz biraz da listeler üzerinde örnekler verelim.

Şöyle bir listemiz olsun:

```
>>> liste = ["elma", "armut", "kebap"]
>>> re.search("kebap", liste)
```

Ne oldu? Hata aldınız, değil mi? Bu normal. Çünkü düzenli ifadeler karakter dizileri üzerinde işler. Bunlar doğrudan listeler üzerinde işlem yapamaz. O yüzden bizim Python’a biraz yardımcı olmamız gerekiyor:

```
>>> for i in liste:
...     b = re.search("kebap", i)
...     if b:
...         print b.group()
...
kebap
```

Hatta şimdiye kadar öğrendiklerimizle daha karmaşık bir şeyler de yapabiliriz:

```
>>> import re
>>> import urllib

>>> f = urllib.urlopen("http://www.istihza.com")

>>> for i in f.readlines():
...     b = re.search("programlama", i)
...     if b:
...         print b.group()
...
programlama
programlama
```



Gördüğünüz gibi, <http://www.istihza.com> sayfasında kaç adet “programlama” kelimesi geçiyorsa hepsi ekrana dökülüyor. Siz isterseniz bu kodları biraz daha geliştirebilirsiniz:

```
#-*-coding:utf-8-*-

import re
import urllib

kelime = raw_input("istihza.com'da aramak \
istediğiniz kelime: ")

adres = urllib.urlopen("http://www.istihza.com")

kar_dizisi = "".join(adres)

nesne = re.search(kelime,kar_dizisi)

if nesne:
    print "kelime bulundu: %s"%nesne.group()
else:
    print "kelime bulunamadı!: %s"%kelime
```

İlerde bilginiz artınca daha yetkin kodlar yazabilecek duruma geleceğiz. Ama şimdilik elimizde olanlar ancak yukarıdaki kodu yazmamıza müsaade ediyor. Unutmayın, düzenli ifadeler sahasında ısınma turları atıyoruz daha...

### 19.1.3 findall() Metodu

Python komut satırında, yani etkileşimli kabukta, `dir(re)` yazdığımız zaman aldığımız listeye tekrar bakarsak orada bir de `findall()` adlı bir metodun olduğunu görürüz. İşte bu bölümde `findall()` adlı bu önemli metodu incelemeye çalışacağız.

Önce şöyle bir metin alalım elimize:

```
metin = """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..."""
```

Bu metin içinde geçen bütün “Python” kelimelerini bulmak istiyoruz:

```
>>> print re.findall("Python",metin)

['Python', 'Python', 'Python', 'Python', 'Python', 'Python']
```

Gördüğünüz gibi, metinde geçen bütün “Python” kelimelerini bir çırpıda liste olarak aldık. Aynı işlemi `search()` metodunu kullanarak yapmak istersek yolu biraz uzatmamız gerekir:

```
>>> liste = metin.split()
>>> for i in liste:
...     nesne = re.search("Python",i)
...     if nesne:
...         print nesne.group()
... 
```

```
Python
Python
Python
Python
Python
Python
```

Gördüğünüz gibi, metinde geçen bütün “Python” kelimelerini `search()` metodunu kullanarak bulmak için öncelikle “metin” adlı karakter dizisini, daha önce karakter dizilerini işlerken gördüğümüz `split()` metodu yardımıyla bir liste haline getiriyoruz. Ardından bu liste üzerinde bir for döngüsü kurduktan sonra `search()` ve `group()` metotlarını kullanarak bütün “Python” kelimelerini ayıklıyoruz. Eğer karakter dizisini yukarıdaki şekilde listeye dönüştürmezsek şöyle bir netice alırız:

```
>>> nesne = re.search("Python",metin)
>>> print nesne.group()
```

```
Python
```

Bu şekilde metinde geçen sadece ilk “Python” kelimesini alabiliyoruz. Eğer, doğrudan karakter dizisine, listeye dönüştürmeksizin, for döngüsü uygulamaya kalkarsak şöyle bir şey olur:

```
>>> for i in a:
...     nesne = re.search("Python",metin)
...     if nesne:
...         print nesne.group()
```

Gördüğünüz gibi, yukarıdaki kod ekranımızı “Python” çıktısıyla dolduruyor. Eğer en sonda `print nesne.group()` yazmak yerine `print i` yazarsanız, Python’un ne yapmaya çalıştığını ve neden böyle bir çıktı verdiğini anlayabilirsiniz.

## 19.2 Metakarakterler

Şimdiye kadar düzenli ifadelerle ilgili olarak verdiğimiz örnekler sizi biraz şaşırtmış olabilir. “Zor dediğin bunlar mıydı?” diye düşünmüş olabilirsiniz. Haklısınız, zira “zor” dediğim, buraya kadar olan kısımda verdiğim örneklerden ibaret değildir. Buraya kadar olan bölümde verdiğim örnekler için en temel kısmını gözler önüne sermek içindi. Şimdiye kadar olan bölümde, mesela, python karakter dizisiyle eşleştirme yapmak için “python” kelimesini kullandık. Esasında bu, düzenli ifadelerin en temel özelliğidir. Yani python karakter dizisini bir düzenli ifade sayacak olursak (ki zaten öyledir), bu düzenli ifade en başta kendisiyle eşleşecektir. Bu ne demek? Şöyle ki: Eğer aradığınız şey python karakter dizisi ise, kullanmanız gereken düzenli ifade de “python” olacaktır. Diyoruz ki: “*Düzenli ifadeler en başta kendileriyle eşleşirler*”. Buradan şu anlam çıkıyor: Demek ki bir de kendileriyle eşleşmeyen düzenli ifadeler var. İşte bu durum, Python’daki düzenli ifadelerle kişiliğini kazandıran şeydir. Biraz sonra ne demek istediğimizi daha açık anlayacaksınız. Artık gerçek anlamıyla düzenli ifadelerle giriş yapıyoruz!

Öncelikle, elimizde aşağıdaki gibi bir liste olduğunu varsayalım:

```
>>> liste = ["özcan", "mehmet", "süleyman", "selim",
... "kemal", "özkan", "esra", "dünder", "esin",
... "esma", "özhan", "özlem"]
```

Diyeelim ki, biz bu liste içinden *özcan*, *özkan* ve *özhan* öğelerini ayıklamak/almak istiyoruz. Bunu yapabilmek için yeni bir bilgiye ihtiyacımız var: Metakarakterler.

Metakarakterler; kabaca, programlama dilleri için özel anlam ifade eden sembollerdir. Örneğin daha önce gördüğümüz “\n” bir bakıma bir metakarakterdir. Çünkü “\n” sembolü Python için özel bir anlam taşır. Python bu sembolü gördüğü yerde yeni bir satıra geçer. Yukarıda “kendisiyle eşleşmeyen karakterler” ifadesiyle kastettiğimiz şey de işte bu metakarakterlerdir. Örneğin, “a” harfi yalnızca kendisiyle eşleşir. Tıpkı “istihza” kelimesinin yalnızca kendisiyle eşleşeceği gibi... Ama mesela “\t” ifadesi kendisiyle eşleşmez. Python bu işareti gördüğü yerde sekme (*tab*) düğmesine basılmış gibi tepki verecektir. İşte düzenli ifadelerde de buna benzer metakarakterlerden yararlanacağız. Düzenli ifadeler içinde de, özel anlam ifade eden pek çok sembol, yani metakarakter vardır. Bu metakarakterlerden biri de “[ ]” sembolüdür. Şimdi yukarıda verdiğimiz listeden *özcan*, *özhan* ve *özkan* öğelerini bu sembolden yararlanarak nasıl ayıklayacağımızı görelim:

```
>>> re.search("öz[chk]an",liste)
```

Bu kodu böyle yazmamamız gerektiğini artık biliyoruz. Aksi halde hata alırsınız. Çünkü daha önce de dediğimiz gibi, düzenli ifadeler karakter dizileri üzerinde işlem yapabilir. Listeler üzerinde değil. Dolayısıyla komutumuzu şu şekilde vermemiz gerekiyor:

```
>>> for i in liste:
...     nesne = re.search("öz[chk]an",i)
...     if nesne:
...         print nesne.group()
```

Aynı işlemi şu şekilde de yapabiliriz:

```
>>> for i in liste:
...     if re.search("öz[chk]an",i):
...         print i
```

Ancak, bu örnekte pek belli olmasa da, son yazdığımız kod her zaman istediğimiz sonucu vermez. Mesela listemiz şöyle olsaydı:

```
>>> liste = ["özcan demir", "mehmet", "süleyman",
... "selim", "kemal", "özkan nuri", "esra", "dünder",
... "esin", "esma", "özhan kamil", "özlem"]
```

Yukarıdaki kod bu liste üzerine uygulandığında, sadece almak istediğimiz kısım değil, ilgisiz kısımlar da gelecektir. Gördüğünüz gibi, uygun kodları kullanarak, *özcan*, *özkan* ve *özhan* öğelerini listeden kolayca ayıkladık. Bize bu imkânı veren şey ise “[ ]” adlı metakarakter oldu. Aslında “[ ]” metakaracterinin ne işe yaradığını az çok anlamış olmalısınız. Ama biz yine de şöyle bir bakalım bu metakaraktere.

“[ ]” adlı metakarakter, yukarıda verdiğimiz listedeki “öz” ile başlayıp, “c”, “h” veya “k” harflerinden herhangi biri ile devam eden ve “an” ile biten bütün öğeleri ayıklıyor. Gelin bununla ilgili bir örnek daha yapalım:

```
>>> for i in liste:
...     nesne = re.search("es[mr]a",i)
...     if nesne:
...         print nesne.group()
```

Gördüğünüz gibi, “es” ile başlayıp, “m” veya “r” harflerinden herhangi biriyle devam eden ve sonunda da “a” harfi bulunan bütün öğeleri ayıklatıldı. Bu da bize *esma* ve *esra* çıktılarını verdi...

Dediğimiz gibi, metakarakterler programlama dilleri için özel anlam ifade eden sembollerdir. “Normal” karakterlerden farklı olarak, metakarakterlerle karşılaşan bir bilgisayar normalden farklı bir tepki verecektir. Yukarıda metakarakterlere örnek olarak “\n” ve “\t” kaçış dizilerini

vermiştik. Örneğin Python’da `print "\n"` gibi bir komut verdiğimizde, Python ekrana “\n” yazdırmak yerine bir alt satıra geçecektir. Çünkü “\n” Python için özel bir anlam taşımaktadır. Düzenli ifadelerde de birtakım metakarakterlerin kullanıldığını öğrendik. Bu metakarakterler, düzenli ifadeleri düzenli ifade yapan şeydir. Bunlar olmadan düzenli ifadelerle yararlı bir iş yapmak mümkün olmaz. Bu giriş bölümünde düzenli ifadelerde kullanılan metakarakterlere örnek olarak “[ ]” sembolünü verdik. Herhangi bir düzenli ifade içinde “[ ]” sembolünü gören Python, doğrudan doğruya bu sembolle eşleşen bir karakter dizisi aramak yerine, özel bir işlem gerçekleştirecektir. Yani “[ ]” sembolü kendisiyle eşleşmeyecektir...

Python’da bulunan temel metakarakterleri topluca görelim:

[ ] . \* + ? { } ^ \$ | ( )

Doğrudur, yukarıdaki karakterler, çizgi romanlardaki küfürlere benziyor. Endişelenmeyin, biz bu metakarakterleri olabildiğince sindirilebilir hale getirmek için elimizden gelen çabayı göstereceğiz.

Bu bölümde düzenli ifadelerin zor kısmı olan metakarakterlere, okurlarımızı ürkütmeden, yumuşak bir giriş yapmayı amaçladık. Şimdi artık metakarakterlerin temelini attığımıza göre üste kat çıkmaya başlayabiliriz.

### 19.2.1 [ ] (Köşeli Parantez)

[ ] adlı metakaraktere önceki bölümde değinmiştik. Orada verdiğimiz örnek şuydu:

```
>>> for i in liste:
...     nesne = re.search("öz[chk]an",i)
...     if nesne:
...         print nesne.group()
```

Yukarıdaki örnekte, bir liste içinde geçen *özcan*, *özhan* ve *özkan* öğelerini ayıklıyoruz. Burada bu üç öğedeki farklı karakterleri (“c”, “h” ve “k”) köşeli parantez içinde nasıl belirttiğimize dikkat edin. Python, köşeli parantez içinde gördüğü bütün karakterleri tek tek liste öğelerine uyguluyor. Önce “öz” ile başlayan bütün öğeleri alıyor, ardından “öz” hecesinden sonra “c” harfiyle devam eden ve “an” hecesi ile biten öğeyi buluyor. Böylece *özcan* öğesini bulmuş oldu. Aynı işlemi, “öz” hecesinden sonra “h” harfini barındıran ve “an” hecesiyle biten öğeye uyguluyor. Bu şekilde ise *özhan* öğesini bulmuş oldu. En son hedef ise “öz” ile başlayıp “k” harfi ile devam eden ve “an” ile biten öğe. Yani listedeki *özkan* öğesi... En nihayetinde de elimizde *özcan*, *özhan* ve *özkan* öğeleri kalmış oluyor.

Bir önceki bölümde yine [ ] metakarakteriyle ilgili olarak şu örneği de vermiştik:

```
>>> for i in liste:
...     nesne = re.search("es[mr]a",i)
...     if nesne:
...         print nesne.group()
```

Bu örneğin de “*özcan*, *özkan*, *özhan*” örneğinden bir farkı yok. Burada da Python köşeli parantez içinde gördüğü bütün karakterleri tek tek liste öğelerine uygulayıp, *esma* ve *esra* öğelerini bize veriyor.

Şimdi bununla ilgili yeni bir örnek verelim.

Diyelim ki elimizde şöyle bir liste var:

```
>>> a = ["23BH56", "TY76Z", "4Y7UZ", "TYUDZ", "34534"]
```

Mesela biz bu listedeki öğeler içinde, sayıyla başlayanları ayıklayalım. Şimdi şu kodları dikkatlice inceleyin:

```
>>> for i in a:
...     if re.match("[0-9]",i):
...         print i
...
23BH56
4Y7UZ
34534
```

Burada parantez içinde kullandığımız ifadeye dikkat edin. “0” ile “9” arasındaki bütün öğeleri içeren bir karakter dizisi tanımladık. Yani kısaca, içinde herhangi bir sayı barındıran öğeleri kapsama alanımıza aldık. Burada ayrıca `search()` yerine `match()` metodunu kullandığımıza da dikkat edin. `match()` metodunu kullanmamızın nedeni, bu metodun bir karakter dizisinin sadece en başına bakması... Amacımız sayı ile başlayan bütün öğeleri ayıklamak olduğuna göre, yukarıda yazdığımız kod, liste öğeleri içinde yer alan ve sayı ile başlayan bütün öğeleri ayıklayacaktır. Biz burada Python’a şu emri vermiş oluyoruz: *“Bana sayı ile başlayan bütün öğeleri bul! Önemli olan bu öğelerin sayıyla başlamasıdır! Sayıyla başlayan bu öğeler ister harfle devam etsin, ister başka bir karakterle... Sen yeter ki bana sayı ile başlayan öğeleri bul!”*

Bu emri alan Python, hemen liste öğelerini gözden geçirecek ve bize `23BH56`, `4Y7UZ` ve `34534` öğelerini verecektir. Dikkat ederseniz, Python bize listedeki `TY76Z` ve `TYUDZ` öğelerini vermedi. Çünkü `TY76Z` içinde sayılar olsa da bunlar bizim ölçütümüze uyacak şekilde en başta yer almıyor. `TYUDZ` öğesinde ise tek bir sayı bile yok...

Şimdi de isterseniz listedeki `TY76Z` öğesini nasıl alabileceğimize bakalım:

```
>>> for i in a:
...     if re.match("[A-Z][A-Z][0-9]",i):
...         print i
```

Burada dikkat ederseniz düzenli ifademizin başında A-Z diye bir şey yazdık. Bu ifade “A” ile “Z” harfleri arasındaki bütün karakterleri temsil ediyor. Biz burada yalnızca büyük harfleri sorguladık. Eğer küçük harfleri sorgulamak isteseydik “A-Z” yerine “a-z” diyecektik. Düzenli ifademiz içinde geçen birinci “A-Z” ifadesi aradığımız karakter dizisi olan “TY76Z” içindeki “T” harfini, ikinci “A-Z” ifadesi “Y” harfini, “0-9” ifadesi ise “7” sayısını temsil ediyor. Karakter dizisi içindeki geri kalan harfler ve sayılar otomatik olarak eşleştirilecektir. O yüzden onlar için ayrı bir şey yazmaya gerek yok. Yalnız bu söylediğimiz son şey sizi aldatmasın. Bu “otomatik eşleştirme” işlemi bizim şu anda karşı karşıya olduğumuz karakter dizisi için geçerlidir. Farklı nitelikteki karakter dizilerinin söz konusu olduğu başka durumlarda işler böyle yürümeyebilir. Düzenli ifadeleri başarılı bir şekilde kullanabilmenin ilk şartı, üzerinde işlem yapılacak karakter dizisini tanımdır. Bizim örneğimizde yukarıdaki gibi bir düzenli ifade kalıbı oluşturmak işimizi görüyor. Ama başka durumlarda, duruma uygun başka kalıplar yazmak gerekebilir/gerekecektir. Dolayısıyla, tek bir düzenli ifade kalıbıyla hayatın geçmeyeceğini unutmamalıyız.

Şimdi yukarıdaki kodu `search()` ve `group()` metotlarını kullanarak yazmayı deneyin. Elde ettiğiniz sonuçları dikkatlice inceleyin. `match()` ve `search()` metotlarının ne gibi farklılıklara sahip olduğunu kavramaya çalışın... Sorunuz olursa bana nasıl ulaşacağınızı biliyorsunuz...

Bu arada, düzenli ifadelerle ilgili daha fazla şey öğrendiğimizde yukarıdaki kodu çok daha sade bir biçimde yazabileceğiz.

## 19.2.2 . (Nokta)

Bir önceki bölümde [ ] adlı metakarakteri incelemiştik. Bu bölümde ise farklı bir metakarakteri inceleyeceğiz. İnceleyeceğimiz metakarakter: "."

Bu metakarakter, yeni satır karakteri hariç bütün karakterleri temsil etmek için kullanılır. Mesela:

```
>>> for i in liste:
...     nesne = re.match("es.a",i)
...     if nesne:
...         print nesne.group()
...
esma
esra
```

Gördüğünüz gibi, daha önce [ ] metakarakterini kullanarak yazdığımız bir düzenli ifadeyi bu kez farklı şekilde yazıyoruz. Unutmayın, bir düzenli ifade birkaç farklı şekilde yazılabilir. Biz bunlar içinde en basit ve en anlaşılır olanını seçmeliyiz. Ayrıca yukarıdaki kodu birkaç farklı şekilde de yazabilirsiniz. Mesela şu yazım da bizim durumumuzda geçerli bir seçenek olacaktır:

```
>>> for i in liste:
...     if re.match("es.a",i):
...         print i
```

Tabii ki biz, o anda çözmek durumunda olduğumuz soruna en uygun olan seçeneği tercih etmeliyiz... Yalnız, unutmamamız gereken şey, bu "." adlı metakarakterin sadece tek bir karakterin yerini tutuyor olmasıdır. Yani şöyle bir kullanım bize istediğimiz sonucu vermez:

```
>>> liste = ["ahmet","kemal", "kamil", "mehmet"]

>>> for i in liste:
...     if re.match(".met",i):
...         print i
```

Burada "." sembolü "ah" ve "meh" hecelerinin yerini tutamaz. "." sembolünün görevi sadece tek bir karakterin yerini tutmaktır (yeni satır karakteri hariç). Ama biraz sonra öğreneceğimiz metakarakter yardımıyla "ah" ve "meh" hecelerinin yerini de tutabileceğiz.

"." sembolünü kullanarak bir örnek daha yapalım. Bir önceki bölümde verdiğimiz a listesini hatırlıyorsunuz:

```
>>> a = ['23BH56', 'TY76Z', '4Y7UZ', 'TYUDZ', '34534']
```

Önce bu listeye bir öge daha ekleyelim:

```
>>> a.append("1agAY54")
```

Artık elimizde şöyle bir liste var:

```
>>> a = ['23BH56', 'TY76Z', '4Y7UZ', 'TYUDZ',
... '34534', "1agAY54"]
```

Şimdi bu listeye şöyle bir düzenli ifade uygulayalım:

```
>>> for i in a:
...     if re.match("[0-9a-z]",i):
...         print i
...
```

23BH56  
34534  
1agAY54

Burada yaptığımız şey çok basit. Şu özelliklere sahip bir karakter dizisi arıyoruz:

1. Herhangi bir karakter ile başlayacak. Bu karakter sayı, harf veya başka bir karakter olabilir.
2. Ardından bir sayı veya alfabedeki küçük harflerden herhangi birisi gelecek.
3. Bu ölçütleri karşıladıktan sonra, aradığımız karakter dizisi herhangi bir karakter ile devam edebilir.

Yukarıdaki ölçütlere uyan karakter dizilerimiz: *23BH56, 34534, 1agAY54*

Yine burada da kendinize göre birtakım değişiklikler yaparak, farklı yazım şekilleri ve farklı metotlar kullanarak ne olup ne bittiğini daha iyi kavrayabilirsiniz. Düzenli ifadeleri gereği gibi anlayabilmek için bol bol uygulama yapmamız gerektiğini unutmamalıyız.

### 19.2.3 \* (Yıldız)

Bu metakaracter, kendinden önce gelen bir düzenli ifade kalıbını sıfır veya daha fazla sayıda eşleştirir. Tanımı biraz karışık olsa da örnek yardımıyla bunu da anlayacağız:

```
>>> yeniliste = ["st", "sat", "saat", "saaat", "falanca"]
>>> for i in yeniliste:
...     if re.match("sa*t",i):
...         print i
```

Burada "\*" sembolü kendinden önce gelen "a" karakterini sıfır veya daha fazla sayıda eşleştiriyor. Yani mesela "st" içinde sıfır adet "a" karakteri var. Dolayısıyla bu karakter yazdığımız düzenli ifadeyle eşleşiyor. "sat" içinde bir adet "a" karakteri var. Dolayısıyla bu da eşleşiyor. "saat" ve "saaat" karakter dizilerinde sırasıyla iki ve üç adet "a" karakteri var. Tabii ki bunlar da yazdığımız düzenli ifadeyle eşleşiyor. Listemizin en son ögesi olan "falanca" da ilk hecede bir adet "a" karakteri var. Ama bu öğedeki sorun, bunun "s" harfiyle başlamaması. Çünkü biz yazdığımız düzenli ifadede, aradığımız şeyin "s" harfi ile başlamasını, sıfır veya daha fazla sayıda "a" karakteri ile devam etmesini ve ardından da "t" harfinin gelmesini istemiştik. "falanca" ögesi bu koşulları karşılamadığı için süzgecimizin dışında kaldı.

Burada dikkat edeceğimiz nokta, "\*" metakaracterinin kendinden önce gelen yalnızca bir karakterle ilgileniyor olması... Yani bizim örneğimizde "\*" sembolü sadece "a" harfinin sıfır veya daha fazla sayıda bulunup bulunmamasıyla ilgileniyor. Bu ilgi, en baştaki "s" harfini kapsamıyor. "s" harfinin de sıfır veya daha fazla sayıda eşleşmesini istersek düzenli ifademizi "s\*a\*t" veya "[sa]\*t" biçiminde yazmamız gerekir... Bu iki seçenek içinde "[sa]\*t" şeklindeki yazımı tercih etmenizi tavsiye ederim. Burada, daha önce öğrendiğimiz "[ ]" metakaracteri yardımıyla "sa" harflerini nasıl gruptladığımıza dikkat edin...

Şimdi "." metakaracterini anlatırken istediğimiz sonucu alamadığımız listeye dönelim. Orada "ahmet" ve "mehmet" öğelerini listeden başarıyla ayıklayamamıştık. O durumda bizim başarısız olmamıza neden olan kullanım şöyleydi:

```
>>> liste = ["ahmet", "kemal", "kamil", "mehmet"]
>>> for i in liste:
...     if re.match(".met",i):
...         print i
```

Ama artık elimizde “\*” gibi bir araç olduğuna göre şimdi istediğimiz şeyi yapabiliriz. Yapmamız gereken tek şey “.” sembolünden sonra “\*” sembolünü getirmek:

```
>>> for i in liste:
...     if re.match(".*met",i):
...         print i
```

Gördüğünüz gibi “ahmet” ve “mehmet” öğelerini bu kez başarıyla ayıkladık. Bunu yapmamızı sağlayan şey de “\*” adlı metakarakter oldu... Burada Python’a şu emri verdik: “Bana kelime başında herhangi bir karakteri (.” sembolü herhangi bir karakterin yerini tutuyor) sıfır veya daha fazla sayıda içeren ve sonu da “met” ile biten bütün öğeleri ver!”

Bir önceki örneğimizde “a” harfinin sıfır veya daha fazla sayıda bulunup bulunmamasıyla ilgilenmiştik. Bu son örneğimizde ise herhangi bir harfin/karakterin sıfır veya daha fazla sayıda bulunup bulunmamasıyla ilgilendik. Dolayısıyla “.\*met” şeklinde yazdığımız düzenli ifade, “ahmet”, “mehmet”, “muhammet”, “ismet”, “kısmet” ve hatta tek başına “met” gibi bütün öğeleri kapsayacaktır. Kısaca ifade etmek gerekirse, sonu “met” ile biten her şey (“met” ifadesinin kendisi de dâhil olmak üzere) kapsama alanımıza girecektir. Bunu günlük hayatta nerede kullanabileceğinizi hemen anlamış olmalısınız. Mesela bir dizin içindeki bütün “mp3” dosyalarını bu düzenli ifade yardımıyla listeleyebiliriz:

```
>>> import os
>>> import re

>>> dizin = os.listdir(os.getcwd())

>>> for i in dizin:
...     if re.match(".*mp3",i):
...         print i
```

match() metodunu anlattığımız bölümde bu metodun bir karakter dizisinin yalnızca başlangıcıyla ilgilendiğini söylemiştik. Mesela o bölümde verdiğimiz şu örneği hatırlıyorsunuzdur:

```
>>> a = "python güçlü bir dildir"
>>> re.match("güçlü", a)
```

Bu örnekte Python bize çıktı olarak “None” değerini vermişti. Yani herhangi bir eşleşme bulamamıştı. Çünkü dediğimiz gibi, match() metodu bir karakter dizisinin yalnızca en başına bakar. Ama geldiğimiz şu noktada artık bu kısıtlamayı nasıl kaldıracağınızı biliyorsunuz:

```
>>> re.match(".*güçlü",a)
```

Ama match() metodunu bu şekilde zorlamak yerine performans açısından en doğru yol bu tür işler için search() metodunu kullanmak olacaktır.

Bunu da geçtiğimize göre artık yeni bir metakarakterini incelemeye başlayabiliriz.

## 19.2.4 + (Artı)

Bu metakarakter, bir önceki metakarakterimiz olan “\*” ile benzerdir. Hatırlarsanız, “\*” metakarakterini kendisinden önceki sıfır veya daha fazla sayıda tekrar eden karakterleri ayıklıyordu. “+” metakarakterini ise kendisinden önceki bir veya daha fazla sayıda tekrar eden karakterleri ayıklar. Bildiğiniz gibi, önceki örneklerimizden birinde “ahmet” ve “mehmet” öğelerini şu şekilde ayıklamıştık:



```
>>> for i in liste:
...     if re.match(".*met",i):
...         print i
```

Burada “ahmet” ve “mehmet” dışında “met” şeklinde bir öge de bu düzenli ifadenin kapsamına girecektir. Mesela listemiz şöyle olsa idi:

```
>>> liste = ["ahmet","mehmet","met","kezban"]
```

Yukarıdaki düzenli ifade bu listedeki “met” ögesini de içine alacaktı. Çünkü “\*” adlı metakarakter sıfır sayıda tekrar eden karakterleri de ayıklıyor. Ama bizim istediğimiz her zaman bu olmayabilir. Bazen de, ilgili karakterin en az bir kez tekrar etmesini isteriz. Bu durumda yukarıdaki düzenli ifadeyi şu şekilde yazmamız gerekir:

```
>>> for i in liste:
...     if re.match("."+met",i):
...         print i
```

Burada şu komutu vermiş olduk: *” Bana sonu ‘met’ ile biten bütün öğeleri ver! Ama bana ‘met’ ögesini yalnız başına verme!”* Aynı işlemi `search()` metodunu kullanarak da yapabileceğimizi biliyorsunuz:

```
>>> for i in liste:
...     nesne = re.search("."+met",i)
...     if nesne:
...         nesne.group()
...
ahmet
mehmet
```

Bir de daha önce verdiğimiz şu örneğe bakalım:

```
>>> yeniliste = ["st", "sat", "saat", "saaat", "falanca"]

>>> for i in yeniliste:
...     if re.match("sa*t",i):
...         print i
```

Burada yazdığımız düzenli ifadenin özelliği nedeniyle “st” de kapsama alanı içine giriyordu. Çünkü burada “\*” sembolü “a” karakterinin hiç bulunmadığı durumları da içine alıyor. Ama eğer biz “a” karakteri en az bir kez geçsin istiyorsak, düzenli ifademizi şu şekilde yazmalıyız:

```
>>> for i in yeniliste:
...     if re.match("sa+t",i):
...         print i
```

Hatırlarsanız önceki derslerimizden birinde köşeli parantezi anlatırken şöyle bir örnek vermiştik:

```
>>> a = ["23BH56","TY76Z","4Y7UZ","TYUDZ","34534"]

>>> for i in a:
...     if re.match("[A-Z][A-Z][0-9]",i):
...         print i
```

Burada amacımız sadece TY76Z ögesini almaktı. Dikkat ederseniz, ögenin başındaki “T” ve “Y” harflerini bulmak için iki kez “[A-Z]” yazdık. Ama artık “+” metakarakterini öğrendiğimize göre aynı işi daha basit bir şekilde yapabiliriz:

```
>>> for i in a:
...     if re.match("[A-Z]+[0-9]",i):
...         print i
...
TY76Z
```

Burada “[A-Z]” düzenli ifade kalıbını iki kez yazmak yerine bir kez yazıp yanına da “+” sembolünü koyarak, bu ifade kalıbının bir veya daha fazla sayıda tekrar etmesini istediğimizi belirttik...

“+” sembolünün ne iş yaptığını da anladığımıza göre, artık yeni bir metakarakter incelemeye başlayabiliriz.

### 19.2.5 ? (Soru İşareti)

Hatırlarsanız, “\*” karakteri sıfır ya da daha fazla sayıda eşleşmeleri; “+” ise bir ya da daha fazla sayıda eşleşmeleri kapsıyordu. İşte şimdi göreceğimiz “?” sembolü de eşleşme sayısının sıfır veya bir olduğu durumları kapsıyor. Bunu daha iyi anlayabilmek için önceden verdiğimiz şu örneğe bakalım:

```
>>> yeniliste = ["st", "sat", "saat", "saaat", "falanca"]

>>> for i in yeniliste:
...     if re.match("sa*t",i):
...         print i
...
st
sat
saat
saaat

>>> for i in yeniliste:
...     if re.match("sa+t",i):
...         print i
...
sat
saat
saaat
```

“\*” ve “+” sembollerinin hangi karakter dizilerini ayıkladığını görüyoruz. Şimdi de “?” sembolünün ne yaptığına bakalım:

```
>>> for i in yeniliste:
...     if re.match("sa?t",i):
...         print i
...
st
sat
```

Gördüğümüz gibi, “?” adlı metakarakterimiz, kendisinden önce gelen karakterin hiç bulunmadığı (yani sıfır sayıda olduğu) ve bir adet bulunduğu durumları içine alıyor. Bu yüzden de çıktı olarak bize sadece “st” ve “sat” öğelerini veriyor.

Şimdi bu metakarakter kullanarak gerçek hayatta karşımıza çıkabilecek bir örnek verelim. Bu metakarakterin tanımına tekrar bakarsak, “olsa da olur olmasa da olur” diyebileceğimiz durumlar için bu metakarakterin rahatlıkla kullanılabileceğini görürüz.

Şöyle bir örnek verelim: Diyelim ki bir metin üzerinde arama yapacaksınız. Aradığınız kelime “uluslararası”:

```
metin = """Uluslararası hukuk, uluslar arası ilişkiler altında bir disiplindir. Uluslararası ilişkiler
```

**Not:** Bu metin [http://tr.wikipedia.org/wiki/Uluslararası\\_hukuk](http://tr.wikipedia.org/wiki/Uluslararası_hukuk) adresinden alınıp üzerinde ufak değişiklikler yapılmıştır.

Şimdi yapmak istediğimiz şey “uluslararası” kelimesini bulmak. Ama dikkat ederseniz metin içinde “uluslararası” kelimesi aynı zamanda “uluslar arası” şeklinde de geçiyor. Bizim bu iki kullanımı da kapsayacak bir düzenli ifade yazmamız gerekecek.

```
>>> nesne = re.findall("[Uu]luslar ?arası",metin)
>>> for i in nesne:
...     print i
```

Verdiğimiz düzenli ifade kalıbını dikkatlice inceleyin. Bildiğiniz gibi, “?” metakarakteri, kendinden önce gelen karakterin (düzenli ifade kalıbını) sıfır veya bir kez geçtiği durumları arıyor. Burada “?” sembolünü “ ” karakterinden, yani “boşluk” karakterinden sonra kullandık. Dolayısıyla, “boşluk karakterinin sıfır veya bir kez geçtiği durumları” hedefledik. Bu şekilde hem “uluslar arası” hem de “uluslararası” kelimesini ayıklamış olduk. Düzenli ifademizde ayrıca şöyle bir şey daha yazdık: “[Uu]”. Bu da gerekiyor. Çünkü metnimiz içinde “uluslararası” kelimesinin büyük harfle başladığı yerler de var... Bildiğiniz gibi, “uluslar” ve “Uluslar” kelimeleri asla aynı değildir. Dolayısıyla hem “u” harfini hem de “U” harfini bulmak için, daha önce öğrendiğimiz “[ ]” metakarakterini kullanıyoruz.

### 19.2.6 { } (Küme Parantezi)

{ } adlı metakarakterimiz yardımıyla bir eşleşmeden kaç adet istediğimizi belirtebiliyoruz. Yine aynı örnek üzerinden gidelim:

```
>>> for i in yeniliste:
...     if re.match("sa{3}t",i):
...         print i
...
saaat
```

Burada “a” karakterinin 3 kez tekrar etmesini istediğimizi belirttik. Python da bu emrimizi hemen yerine getirdi. Bu metakarakterin ilginç bir özelliği daha vardır. Küme içinde iki farklı sayı yazarak, bir karakterin en az ve en çok kaç kez tekrar etmesini istediğimizi belirtebiliriz. Örneğin:

```
>>> for i in yeniliste:
...     if re.match("sa{0,3}t",i):
...         print i
...
st
sat
saat
saaat
```

sa{0,3}t ifadesiyle, “a” harfinin en az sıfır kez, en çok da üç kez tekrar etmesini istediğimiz söyledik. Dolayısıyla, “a” harfinin sıfır, bir, iki ve üç kez tekrar ettiği durumlar ayıklanmış oldu. Bu sayı çiftlerini değiştirerek daha farklı sonuçlar elde edebilirsiniz. Ayrıca hangi sayı çiftinin daha önce öğrendiğimiz “?” metakarakteriyle aynı işi yaptığını bulmaya çalışın...

### 19.2.7 ^ (Şapka)

^ sembolünün iki işlevi var. Birinci işlevi, bir karakter dizisinin en başındaki veriyi sorgulamaktır. Yani aslında `match()` metodunun varsayılan olarak yerine getirdiği işlevi bu metakarakter yardımıyla açıkça belirterek yerine getirebiliyoruz. Şu örneğe bakalım:

```
>>> a = ['23BH56', 'TY76Z', '4Y7UZ', 'TYUDZ',
...      '34534', '1agAY54']

>>> for i in a:
...     if re.search("[A-Z]+[0-9]",i):
...         print i
...
23BH56
TY76Z
4Y7UZ
1agAY54
```

Bir de şuna bakalım:

```
>>> for i in a:
...     nesne = re.search("[A-Z]+[0-9]",i)
...     if nesne:
...         print nesne.group()
...
BH5
TY7
Y7
AY5
```

Dikkat ederseniz, şu son verdiğimiz kod oldukça hassas bir çıktı verdi bize. Çıktıdaki bütün değerler, aynen düzenli ifademizde belirttiğimiz gibi, yan yana bir veya daha fazla harf içeriyor ve sonra da bir sayı ile devam ediyor. Bu farklılığın nedeni, ilk kodlarda `print i` ifadesini kullanmamız. Bu durumun çıktılarımızı nasıl değiştirdiğine dikkat edin. Bir de şu örneğe bakalım:

```
>>> for i in a:
...     if re.match("[A-Z]+[0-9]",i):
...         print i
...
TY76Z
```

Burada sadece `TY76Z` çıktısını almamızın nedeni, `match()` metodunun karakter dizilerinin en başına bakıyor olması. Aynı etkiyi `search()` metoduyla da elde etmek için, başlıkta geçen “^” (şapka) sembolünden yararlanacağız:

```
>>> for i in a:
...     nesne = re.search("^[A-Z]+[0-9]",i)
...     if nesne:
...         print nesne.group()
...
TY7
```

Gördüğümüz gibi, “^” (şapka) metakarakteri `search()` metodunun, karakter dizilerinin sadece en başına bakmasını sağladı. O yüzden de bize sadece, `TY7` çıktısını verdi. Hatırlarsanız aynı kodu, şapkasız olarak, şu şekilde kullanmıştık yukarıda:

```
>>> for i in a:
...     nesne = re.search("[A-Z]+[0-9]",i)
...     if nesne:
```

```
...     print nesne.group()
...
BH5
TY7
Y7
AY5
```

Gördüğünüz gibi, şapka sembolü olmadığında `search()` metodu karakter dizisinin başına bakmakla yetinmiyor, aynı zamanda karakter dizisinin tamamını tarıyor. Biz yukarıdaki koda bir “^” sembolü ekleyerek, metodumuzun sadece karakter dizisinin en başına bakmasını istedik. O da emrimize sadakatle uydu. Burada dikkatimizi çekmesi gereken başka bir nokta da `search()` metodundaki çıktının kırılmış olması. Dikkat ettiyseniz, `search()` metodu bize öğenin tamamını vermedi. Öğelerin yalnızca “[A-Z]+[0-9]” kalıbına uyan kısımlarını kesip attı önümüze. Çünkü biz ona tersini söylemedik. Eğer öğelerin tamamını istiyorsak bunu açık açık belirtmemiz gerekir:

```
>>> for i in a:
...     nesne = re.search("[A-Z]+[0-9].*",i)
...     if nesne:
...         print nesne.group()
...
BH56
TY76Z
Y7UZ
AY54
```

Veya metodumuzun karakter dizisinin sadece en başına bakmasını istersek:

```
>>> for i in a:
...     nesne = re.search("^([A-Z]+[0-9]).*",i)
...     if nesne:
...         print nesne.group()
...
TY76Z
```

Bu kodlarda düzenli ifade kalıbının sonuna “.” sembolünü eklediğimize dikkat edin. Böylelikle metodumuzun sonu herhangi bir şekilde biten öğeleri bize vermesini sağladık...

Başta da söylediğimiz gibi, “^” metakarakterinin, karakter dizilerinin en başına demir atmak dışında başka bir görevi daha vardır: “Hariç” anlamına gelmek... Bu görevini sadece “[ ]” metakarakterinin içinde kullanıldığı zaman yerine getirir. Bunu bir örnekle görelim. Yukarıdaki listemiz üzerinde öyle bir süzgeç uygulayalım ki, *1agAY54* öğesi çıktılarımız arasında görünmesin... Bu öğeyi avlayabilmek için kullanmamız gereken düzenli ifade şöyle olacaktır: `[0-9A-Z][^a-z]+`

```
>>> for i in a:
...     nesne = re.match("[0-9A-Z][^a-z]+",i)
...     if nesne:
...         print nesne.group()
```

Burada şu ölçütlere sahip bir öğe arıyoruz:

1. Aradığımız öğe bir sayı veya büyük harf ile başlamalı
2. En baştaki sayı veya büyük harften sonra küçük harf GELMEMELİ (Bu ölçütü “^” işareti sağlıyor)
3. Üstelik bu “küçük harf gelmeme durumu” bir veya daha fazla sayıda tekrar etmeli... Yani baştaki sayı veya büyük harften sonra kaç tane olursa olsun asla küçük harf gelmemeli

(Bu ölçütü de “+” işareti sağlıyor)

Bu ölçütlere uymayan tek öge “1agAY54” olacaktır. Dolayısıyla bu öge çıktıda görünmeyecek...

Burada, “^” işaretinin nasıl kullanıldığına ve küçük harfleri nasıl dışarıda bıraktığına dikkat edin. Unutmayalım, bu “^” işaretinin “hariç” anlamı sadece “[ ]” metakarakterinin içinde kullanıldığı zaman geçerlidir.

### 19.2.8 \$ (Dolar)

Bir önceki bölümde “^” işaretinin, karakter dizilerinin en başına demir attığını söylemiştik. Yani bu sembol arama/eşleştirme işleminin karakter dizisinin en başından başlamasını sağlıyordu. Bu sembol bir bakıma karakter dizilerinin nasıl başlayacağını belirliyordu. İşte şimdi göreceğimiz “dolar işareti” de (\$) karakter dizilerinin nasıl biteceğini belirliyor. Bu soyut açıklamaları somut bir örnekle bağlayalım:

```
>>> liste = ["at", "katkı", "fakat", "atkı", "rahat",  
... "mat", "yat", "sat", "satılık", "katılım"]
```

Gördüğünüz gibi, elimizde on ögelik bir liste var. Diyelim ki biz bu listeden, “at” hecesiyle biten kelimeleri ayıklamak istiyoruz:

```
>>> for i in liste:  
...     if re.search("at$",i):  
...         print i  
...  
at  
fakat  
rahat  
mat  
yat  
sat
```

Burada “\$” metakarakteri sayesinde aradığımız karakter dizisinin nasıl bitmesi gerektiğini belirleyebildik. Eğer biz “at” ile başlayan bütün öğeleri ayıklamak isteseydik ne yapmamız gerektiğini biliyorsunuz:

```
>>> for i in liste:  
...     if re.search("^at",i):  
...         print i  
...  
at  
atkı
```

Gördüğünüz gibi, “^” işareti bir karakter dizisinin nasıl başlayacağını belirlerken, “\$” işareti aynı karakter dizisinin nasıl biteceğini belirliyor. Hatta istersek bu metakarakterleri birlikte de kullanabiliriz:

```
>>> for i in liste:  
...     if re.search("^at$",i):  
...         print i  
...  
at
```

Sonuç tam da beklediğimiz gibi oldu. Verdiğimiz düzenli ifade kalıbı ile “at” ile başlayan ve aynı şekilde biten karakter dizilerini ayıkladık. Bu da bize “at” çıktısını verdi.

### 19.2.9 \ (Ters Bölü)

Bu işaret bildiğimiz “kaçış dizisi”dir... Peki burada ne işi var? Şimdiye kadar öğrendiğimiz konulardan gördüğünüz gibi, Python’daki düzenli ifadeler açısından özel anlam taşıyan bir takım semboller/metakarakterler var. Bunlar kendileriyle eşleşmiyorlar. Yani bir karakter dizisi içinde bu sembolleri arıyorsak eğer, bunların taşıdıkları özel anlam yüzünden bu sembolleri ayıklamak hemencecik mümkün olmayacaktır. Yani mesela biz “\$” sembolünü arıyor olsak, bunu Python’a nasıl anlatacağız? Çünkü bu sembolü yazdığımız zaman Python bunu farklı algılıyor. Lafı dolandırmadan hemen bir örnek verelim...

Diyelim ki elimizde şöyle bir liste var:

```
>>> liste = ["10$", "25€", "20$", "10TL", "25f"]
```

Amacımız bu listedeki dolarlı değerleri ayıklamaksa ne yapacağız? Şunu deneyelim önce:

```
>>> for i in liste:
...     if re.match("[0-9]+$",i):
...         print i
```

Python “\$” işaretinin özel anlamından dolayı, bizim sayıyla biten bir karakter dizisi aradığımızı zannedecek, dolayısıyla da herhangi bir çıktı vermeyecektir. Çünkü listemizde sayıyla biten bir karakter dizisi yok... Peki biz ne yapacağız? İşte bu noktada “\” metakarakteri devreye girecek... Hemen bakalım:

```
>>> for i in liste:
...     if re.match("[0-9]+\$",i):
...         print i
...
10$
20$
```

Gördüğünüz gibi, “\” sembolünü kullanarak “\$” işaretinin özel anlamından kaçtık... Bu metakarakteri de kısaca anlattığımıza göre yeni bir metakarakterle yolumuza devam edebiliriz...

### 19.2.10 | (Dik Çizgi)

Bu metakarakter, birden fazla düzenli ifade kalıbını birlikte eşleştirmemizi sağlar. Bu ne demek? Hemen görelim:

```
>>> liste = ["at", "katkı", "fakat", "atkı", "rahat",
... "mat", "yat", "sat", "satılık", "katılım"]

>>> for i in liste:
...     if re.search("^at|at$",i):
...         print i
...
at
fakat
atkı
rahat
mat
yat
sat
```

Gördüğünüz gibi “[” metakarakterini kullanarak başta ve sonda “at” hecesini içeren kelimeleri ayıkladık. Aynı şekilde, mesela, renkleri içeren bir listeden belli renkleri de ayıklayabiliriz bu metakarakter yardımıyla...

```
...     if re.search("kırmızı|mavi|sarı",i):  
...         print i
```

Sırada son metakarakterimiz olan “()” var...

### 19.2.11 ( ) (Parantez)

Bu metakarakter yardımıyla düzenli ifade kalıplarını gruplayacağız. Bu metakarakter bizim bir karakter dizisinin istediğimiz kısımlarını ayıklamamızda çok büyük kolaylıklar sağlayacak.

Diyelim ki biz [http://www.istihza.com/py2/icindekiler\\_python.html](http://www.istihza.com/py2/icindekiler_python.html) adresindeki bütün başlıkları ve bu başlıklara ait html dosyalarını bir liste halinde almak istiyoruz. Bunun için şöyle bir şey yazabiliriz:

```
import re  
import urllib  
  
url = "http://www.istihza.com/py2/icindekiler_python.html"  
  
f = urllib.urlopen(url)  
  
for i in f:  
    nesne = re.search('<a href="+.html">.+</a>',i)  
    if nesne:  
        print nesne.group()
```

Burada yaptığımız şey şu:

1. Öncelikle “[http://www.istihza.com/py2/icindekiler\\_python.html](http://www.istihza.com/py2/icindekiler_python.html)” sayfasını urllib modülü yardımıyla açtık. Amacımız bu sayfadaki başlıkları ve bu başlıklara ait html dosyalarını listelemek
2. Ardından, bütün sayfayı taramak için basit bir for döngüsü kurduk
3. Düzenli ifade kalıbımızı şöyle yazdık: `<a href="+.html">.+</a>`. Çünkü bahsi geçen web sayfasındaki html uzantılı dosyalar “a href=” tag’ı içinde yer alıyor. Bu durumu, web tarayıcınızda [http://www.istihza.com/py2/icindekiler\\_python.html](http://www.istihza.com/py2/icindekiler_python.html) sayfasını açıp sayfa kaynağını görüntüleyerek teyit edebilirsiniz. (Firefox’ta CTRL+U’ya basarak sayfa kaynağını görebilirsiniz)
4. Yazdığımız düzenli ifade kalıbı içinde dikkatimizi çekmesi gereken bazı noktalar var: Kalıbın “(.+html)” kısmında geçen “+” metakarakteri kendisinden önce gelen düzenli ifadenin bir veya daha fazla sayıda tekrar eden eşleşmelerini buluyor. Burada “+” metakarakterinden önce gelen düzenli ifade, kendisi de bir metakarakter olan “.” sembolü... Bu sembol bildiğiniz gibi, “herhangi bir karakter” anlamına geliyor. Dolayısıyla “.+” ifadesi şu demek oluyor: “*Bana bir veya daha fazla sayıda tekrar eden bütün karakterleri bul!*” Dolayısıyla burada “(.+html)” ifadesini birlikte düşünersek, yazdığımız şey şu anlama geliyor: “*Bana ‘html’ ile biten bütün karakter dizilerini bul!*”
5. “[http://www.istihza.com/py2/icindekiler\\_python.html](http://www.istihza.com/py2/icindekiler_python.html)” adresinin kaynağına baktığımız zaman aradığımız bilgilerin hep şu şekilde olduğunu görüyoruz: `<a href="tbilgi_giris.html">Temel Bilgiler</a>` Dolayısıyla aslında düzenli ifade kalıbımızı yazarken yaptığımız şey, düzenli ifademizi kaynakta görünen şablona uydurmak...



Yukarıda verdiğimiz kodları çalıştırdığımız zaman aldığımız çıktı şu şekilde oluyor:

```
<a href="index.html">ANA SAYFA</a>
...
...
...
```

Hemen hemen amacımıza ulaştık sayılır. Ama gördüğünüz gibi çıktımız biraz karmaşık. Bunları istediğimiz gibi düzenleyebilirsek iyi olurdu, değil mi? Mesela bu çıktıları şu şekilde düzenleyebilmek hoş olurdu:

```
Başlık: ANA SAYFA; Bağlantı: index.html
```

İşte bu bölümde göreceğimiz “( )” metakarakteri istediğimiz şeyi yapmada bize yardımcı olacak.

Dilerseniz en başta verdiğimiz kodlara tekrar dönelim:

```
import re
import urllib

url = "http://www.istihza.com/py2/icindekiler_python.html"

f = urllib.urlopen(url)

for i in f:
    nesne = re.search('<a href=".*html">.*</a>', i)
    if nesne:
        print nesne.group()
```

Şimdi bu kodlarda şu değişikliği yapıyoruz:

```
#-*-coding:utf-8-*-

import re
import urllib

url = "http://www.istihza.com/py2/icindekiler_python.html"

f = urllib.urlopen(url)

for i in f:
    nesne = re.search('<a href="(.*html)">(.*?)</a>', i)
    if nesne:
        print "Başlık: %s;\nBağlantı: %s\n" \
%(nesne.group(2), nesne.group(1))
```

Kodlarda yaptığımız değişikliklere dikkat edin ve anlamaya çalışın. Bazı noktalar gözünüze karanlık göründüyse hiç endişe etmeyin, çünkü bir sonraki bölümde bütün karanlık noktaları tek tek açıklayacağız. Burada en azından, “( )” metakarakterini kullanarak düzenli ifadenin bazı bölümlerini nasıl grupladığımıza dikkat edin.

Bu arada, elbette [www.istihza.com](http://www.istihza.com) sitesinde herhangi bir değişiklik olursa yukarıdaki kodların istediğiniz çıktıyı vermeyeceğini bilmelisiniz. Çünkü yazdığımız düzenli ifade [istihza.com](http://www.istihza.com) sitesinin sayfa yapısıyla sıkı sıkıya bağlantılıdır.

## 19.3 Eşleşme Nesnelerinin Metotları

### 19.3.1 group() metodu

Bu bölümde doğrudan düzenli ifadelerin değil, ama düzenli ifadeler kullanılarak üretilen eşleşme nesnelerinin bir metodu olan `group()` metodundan bahsedeceğiz. Esasında biz bu metodu önceki bölümlerde de kullanmıştık. Ama burada bu metoda biraz daha ayrıntılı olarak bakacağız.

Daha önceki bölümlerden hatırlayacağınız gibi, bu metot düzenli ifadeleri kullanarak eşleştirdiğimiz karakter dizilerini görme imkanı sağlıyordu. Bu bölümde bu metodu “()” metakarakteri yardımıyla daha verimli bir şekilde kullanacağız. İsterseniz ilk olarak şöyle basit bir örnek verelim:

```
>>> a = "python bir program:lama dilidir"

>>> nesne = re.search("(python) (bir) (programlama) (dilidir)",a)
>>> print nesne.group()

python bir programlama dilidir
```

Burada düzenli ifade kalıbımızı nasıl grupladığımıza dikkat edin. `print nesne.group()` komutunu verdiğimizde eşleşen karakter dizileri ekrana döküldü. Şimdi bu grupladığımız bölümlere tek tek erişelim:

```
>>> print nesne.group(0)

python bir programlama dilidir
```

Gördüğünüz gibi, “0” indeksi eşleşen karakter dizisinin tamamını veriyor. Bir de şuna bakalım:

```
>>> print nesne.group(1)

python
```

Burada 1 numaralı grubun ögesi olan “python”u aldık. Gerisinin nasıl olacağını tahmin edebilirsiniz:

```
>>> print nesne.group(2)

bir

>>> print nesne.group(3)

programlama

>>> print nesne.group(4)

dilidir
```

Bu metodun bize ilerde ne büyük kolaylıklar sağlayacağını az çok tahmin ediyorsunuzdur. İsterseniz kullanabileceğimiz metotları tekrar listeleyelim:

```
>>> dir(nesne)
```

Bu listede `group()` dışında bir de `groups()` adlı bir metodun olduğunu görüyoruz. Şimdi bunun ne iş yaptığına bakalım.

### 19.3.2 groups() metodu

Bu metod, bize kullanabileceğimiz bütün grupları bir demet halinde sunar:

```
>>> print nesne.groups()

('python', 'bir', 'programlama', 'dilidir')
```

Şimdi isterseniz bir önceki bölümde yaptığımız örneğe geri dönelim:

```
#-*-coding:utf-8-*-

import re
import urllib

url = "http://www.istihza.com/py2/icindekiler_python.html"

f = urllib.urlopen(url)

for i in f:
    nesne = re.search('<a href="(.*?)>(.*?)</a>', i)
    if nesne:
        print "Başlık: %s;\nBağlantı: %s\n" \
              %(nesne.group(2), nesne.group(1))
```

Bu kodlarda son satırı şöyle değiştirelim:

```
#-*-coding:utf-8-*-

import re
import urllib

url = "http://www.istihza.com/py2/icindekiler_python.html"

f = urllib.urlopen(url)

for i in f:
    nesne = re.search('<a href="(.*?)>(.*?)</a>', i)
    if nesne:
        print nesne.groups()
```

Gördüğünüz gibi şuna benzer çıktılar elde ediyoruz:

```
('index.html', 'ANA SAYFA')
('icindekiler_python.html', 'PYTHON')
('icindekiler_tkinter.html', 'TKINTER')
('makaleler.html', 'MAKALELER')
...
...
...
```

Demek ki `nesne.groups()` komutu bize “( )” metakarakteri ile daha önceden gruplamış olduğumuz öğeleri bir demet olarak veriyor. Biz de bu demetin öğelerine daha sonradan rahatlıkla erişebiliyoruz...

Böylece eşleştirme nesnelerinin en sık kullanılan iki metodunu görmüş olduk. Bunları daha sonraki örneklerimizde de bol bol kullanacağız. O yüzden şimdilik bu konuya ara verelim.

### 19.3.3 Özel Diziler

Düzenli ifadeler içinde metakarakterler dışında, özel anlamlar taşıyan bazı başka ifadeler de vardır. Bu bölümde bu özel dizileri inceleyeceğiz: Boşluk karakterinin yerini tutan özel dizi: \s

Bu sembol, bir karakter dizisi içinde geçen boşlukları yakalamak için kullanılır. Örneğin:

```
>>> a = ["5 Ocak", "27Mart", "4 Ekim", "Nisan 3"]

>>> for i in a:
...     nesne = re.search("[0-9]\s[A-Za-z]+",i)
...     if nesne:
...         print nesne.group()
...
5 Ocak
4 Ekim
```

Yukarıdaki örnekte, bir sayı ile başlayan, ardından bir adet boşluk karakteri içeren, sonra da bir büyük veya küçük harfle devam eden karakter dizilerini ayıkladık. Burada boşluk karakterini “s” simgesi ile gösterdiğimizize dikkat edin.

#### Ondalık Sayıların Yerini Tutan Özel Dizi: \d

Bu sembol, bir karakter dizisi içinde geçen ondalık sayıları eşleştirmek için kullanılır. Buraya kadar olan örneklerde bu işlevi yerine getirmek için “[0-9]” ifadesinden yararlanıyorduk. Şimdi artık aynı işlevi daha kısa yoldan, “\d” dizisi ile yerine getirebiliriz. İsterseniz yine yukarıdaki örnekten gidelim:

```
>>> a = ["5 Ocak", "27Mart", "4 Ekim", "Nisan 3"]

>>> for i in a:
...     nesne = re.search("\d\s[A-Za-z]+",i)
...     if nesne:
...         print nesne.group()
...
5 Ocak
4 Ekim
```

Burada, “[0-9]” yerine “\d” yerleştirerek daha kısa yoldan sonuca vardık.

#### Alfanümerik Karakterlerin Yerini Tutan Özel Dizi: \w

Bu sembol, bir karakter dizisi içinde geçen alfanümerik karakterleri ve buna ek olarak “\_” karakterini bulmak için kullanılır. Şu örneğe bakalım:

```
>>> a = "abc123_$$%+"
>>> print re.search("\w*",a).group()

abc123_
```

“\w” özel dizisinin hangi karakterleri eşlediğine dikkat edin. Bu özel dizi şu ifadeyle aynı anlamı gelir:

```
[A-Za-z0-9_]
```

Düzenli ifadeler içindeki özel diziler genel olarak bunlardan ibarettir. Ama bir de bunların büyük harfli versiyonları vardır ki, önemli oldukları için onları da inceleyeceğiz.

Gördüğünüz gibi;

1. “\s” özel dizisi boşluk karakterlerini avlıyor

2. “\d” özel dizisi ondalık sayıları avlıyor
3. “\w” özel dizisi alfanümerik karakterleri ve “\_” karakterini avlıyor

Dedik ki, bir de bunların büyük harfli versiyonları vardır. İşte bu büyük harfli versiyonlar da yukarıdaki dizilerin yaptığı işin tam tersini yapar. Yani:

1. “\S” özel dizisi boşluk olmayan karakterleri avlar
2. “\D” özel dizisi ondalık sayı olmayan karakterleri avlar. Yani “[^0-9]” ile eşdeğerdir.
3. “\W” özel dizisi alfanümerik olmayan karakterleri ve “\_” olmayan karakterleri avlar. Yani “[^A-Z-a-z\_]” ile eşdeğerdir. “\D” ve “\W” dizilerinin yeterince anlaşılır olduğunu zannediyorum. Burada sanırım sadece “\S” dizisi bir örnekle somutlaştırılmayı hakediyor:

```
>>> a = ["5 Ocak", "27Mart", "4 Ekim", "Nisan 3"]

>>> for i in a:
...     nesne = re.search("\d+\S\w+", i)
...     if nesne:
...         print nesne.group()
...
27Mart
```

Burada “\S” özel dizisinin listede belirtilen konumda boşluk içermeyen ögeyi nasıl bulduğuna dikkat edin.

Şimdi bu özel diziler için genel bir örnek verip konuyu kapatalım...

Bilgisayarımızda şu bilgileri içeren “adres.txt” adlı bir dosya olduğunu varsayıyoruz:

```
esra : istinye 05331233445
esma : levent 05322134344
sevgi : dudullu 05354445434
kemal : sanayi 05425455555
osman : tahtakale 02124334444
metin : taksim 02124344332
kezban : caddebostan 02163222122
```

Amacımız bu dosyada yer alan isim ve telefon numaralarını “isim > telefon numarası” şeklinde almak:

```
# -*- coding: utf-8 -*-

import re

dosya = open("adres.txt")

for i in dosya.readlines():
    nesne = re.search("(\w+)\s+:\s(\w+)\s+(\d+)", i)
    if nesne:
        print "%s > %s"%(nesne.group(1), nesne.group(3))
```

Burada formülümüz şu şekilde: “Bir veya daha fazla karakter” + “bir veya daha fazla boşluk” + “:” işareti + “bir adet boşluk” + “bir veya daha fazla sayı”

İsterseniz bu bölümü çok basit bir soruyla kapatalım. Sorumuz şu:

Elimizde şu adresteki gibi bir yığın var: <http://www.istihza.com/denemeler/yigin.txt>

Yapmanız gereken, bu yığın içindeki gizli mesajı düzenli ifadeleri kullanarak bulmak... Cevaplarınızı [kistihza\[at\]yahoo\[nokta\]com](mailto:kistihza[at]yahoo[nokta]com) adresine gönderebilirsiniz.

## 19.4 Düzenli İfadelerin Derlenmesi

### 19.4.1 compile() metodu

En başta da söylediğimiz gibi, düzenli ifadeler karakter dizilerine göre biraz daha yavaş çalışırlar. Ancak düzenli ifadelerin işleyişini hızlandırmanın da bazı yolları vardır. Bu yollardan biri de compile() metodunu kullanmaktır. “compile” kelimesi İngilizce’de “derlemek” anlamına gelir. İşte biz de bu compile() metodu yardımıyla düzenli ifade kalıplarımızı kullanmadan önce derleyerek daha hızlı çalışmalarını sağlayacağız. Küçük boyutlu projelerde compile() metodu pek hissedilir bir fark yaratmasa da özellikle büyük çaplı programlarda bu metodu kullanmak oldukça faydalı olacaktır. Basit bir örnekle başlayalım:

```
>>> liste = ["Python2.4", "Python2.5", "Python2.6",
... "Python3.0", "Java"]
>>> derli = re.compile("[A-Za-z]+[0-9]\.[0-9]")

>>> for i in liste:
...     nesne = derli.search(i)
...     if nesne:
...         print nesne.group()
...
Python2.4
Python2.5
Python2.6
Python3.0
```

Burada öncelikle düzenli ifade kalıbımızı derledik. Derleme işlemini nasıl yaptığımıza dikkat edin. Derlenecek düzenli ifade kalıbını compile() metodunda parantez içinde belirtiyoruz. Daha sonra search() metodunu kullanırken ise, re.search() demek yerine, derli.search() şeklinde bir ifade kullanıyoruz. Ayrıca dikkat ederseniz derli.search() kullanımında parantez içinde sadece eşleşecek karakter dizisini kullandık (i). Eğer derleme işlemi yapmamış olsaydık, hem bu karakter dizisini, hem de düzenli ifade kalıbını yan yana kullanmamız gerekecektir. Ama düzenli ifade kalıbımızı yukarıda derleme işlemi esnasında belirttiğimiz için, bu kalıbı ikinci kez yazmamıza gerek kalmadı. Ayrıca burada kullandığımız düzenli ifade kalıbına da dikkat edin. Nasıl bir şablon oturttuğumuzu anlamaya çalışın. Gördüğünüz gibi, liste öğelerinde bulunan “.” işaretini eşleştirmek için düzenli ifade kalıbı içinde “\.” ifadesini kullandık. Çünkü bildiğiniz gibi, tek başına “.” işaretinin Python açısından özel bir anlamı var. Dolayısıyla bu özel anlamdan kaçmak için “\” işaretini de kullanmamız gerekiyor.

Şimdi isterseniz küçük bir örnek daha verelim. Diyelim ki amacımız günlük dolar kurunu almak olsun. Bu iş için <http://www.tcmb.gov.tr/kurlar/today.html> adresini kullanacağız:

```
#-*-coding:utf8-*-

import re
import urllib

f = urllib.urlopen("http://www.tcmb.gov.tr/kurlar/today.html")

derli = re.compile("ABD\SDOLARI\s+[0-9]\.[0-9]+")

nesne = derli.search(f.read())

print nesne.group()
```

Burada kullandığımız “\s” adlı özel diziyi hatırlıyorsunuz. Bu özel dizinin görevi karakter dizileri

içindeki boşlukların yerini tutmaktı. Mesela bizim örneğimizde “ABD” ve “DOLARI” kelimeleri arasındaki boşluğun yerini tutuyor. Ayrıca yine “DOLARI” kelimesi ile 1 doların TL karşılığı olan değerin arasındaki boşlukların yerini de tutuyor. Yalnız dikkat ederseniz “\s” ifadesi tek başına sadece bir adet boşluğun yerini tutar. Biz onun birden fazla boşluğun yerini tutması için yanına bir adet “+” metakarakteri yerleştirdik. Ayrıca burada da “.” işaretinin yerini tutması için “\.” ifadesinden yararlandık.

## 19.5 compile() ile Derleme Seçenekleri

Bir önceki bölümde compile() metodunun ne olduğunu, ne işe yaradığını ve nasıl kullanıldığını görmüştük. Bu bölümde ise “compile” (derleme) işlemi sırasında kullanılabilecek seçenekleri anlatacağız.

### 19.5.1 re.IGNORECASE veya re.I

Bildiğiniz gibi, Python’da büyük-küçük harfler önemlidir. Yani eğer “python” kelimesini arıyorsanız, alacağınız çıktılar arasında “Python” olmayacaktır. Çünkü “python” ve “Python” birbirlerinden farklı iki karakter dizisidir. İşte re.IGNORECASE veya kısaca re.I adlı derleme seçenekleri bize büyük-küçük harfe dikkat etmeden arama yapma imkanı sağlar. Hemen bir örnek verelim:

```
#-*-coding:utf8-*-

import re

metin = """Programlama dili, programcının bir bilgisayara ne yapmasını
istediğini anlatmasının standartlaştırılmış bir yoludur. Programlama
dilleri, programcının bilgisayara hangi veri üzerinde işlem yapacağını,
verinin nasıl depolanıp iletileceğini, hangi koşullarda hangi işlemlerin
yapılacağını tam olarak anlatmasını sağlar. Şu ana kadar 2500’den fazla
programlama dili yapılmıştır. Bunlardan bazıları: Pascal, Basic, C, C#,
C++, Java, Cobol, Perl, Python, Ada, Fortran, Delphi programlama
dilleridir."""

derli = re.compile("programlama",re.IGNORECASE)
print derli.findall(metin)

['Programlama', 'Programlama', 'programlama', 'programlama']
```

---

**Not:** Bu metin [http://tr.wikipedia.org/wiki/Programlama\\_dili](http://tr.wikipedia.org/wiki/Programlama_dili) adresinden alınmıştır.

---

Gördüğünüz gibi, metinde geçen hem “programlama” kelimesini hem de “Programlama” kelimesini ayıklayabildik. Bunu yapmamızı sağlayan şey de re.IGNORECASE adlı derleme seçeneği oldu. Eğer bu seçeneği kullanmasaydık, çıktıda yalnızca “programlama” kelimesini görürdük. Çünkü aradığımız şey aslında “programlama” kelimesi idi. Biz istersek re.IGNORECASE yerine kısaca re.I ifadesini de kullanabiliriz. Aynı anlama gelecektir..

### 19.5.2 re.DOTALL veya re.S

Bildiğiniz gibi, metakarakterler arasında yer alan “.” sembolü herhangi bir karakterin yerini tutuyordu. Bu metakarakter bütün karakterlerin yerini tutmak üzere kullanılabilir. Hatırlarsanız,

“.” metakarakterini anlatırken, bu metakarakterin, yeni satır karakterinin yerini tutmayacağını söylemiştik. Bunu bir örnek yardımıyla görelim. Diyelim ki elimizde şöyle bir karakter dizisi var:

```
>>> a = "Ben Python,\nMonty Python"
```

Bu karakter dizisi içinde “Python” kelimesini temel alarak bir arama yapmak istiyorsak eğer, kullanacağımız şu kod istediğimiz şeyi yeterince yerine getiremeyecektir:

```
>>> print re.search("Python.*",a).group()
```

Bu kod şu çıktıyı verecektir:

```
Python,
```

Bunun sebebi, “.” metakarakterinin “\n” (yeni satır) kaçış dizisini dikkate almamasıdır. Bu yüzden bu kaçış dizisinin ötesine geçip orada arama yapmıyor. Ama şimdi biz ona bu yeteneği de kazandıracğıız:

```
>>> derle = re.compile("Python.*",re.DOTALL)
>>> nesne = derle.search(a)
>>> if nesne:
...     print nesne.group()
```

re.DOTALL seçeneğini sadece re.S şeklinde de kısaltabilirsiniz...

### 19.5.3 re.UNICODE veya re.U

Bu derleme seçeneği, Türkçe programlar yazmak isteyenlerin en çok ihtiyaç duyacağı seçeneklerden biridir. Ne demek istediğimizi tam olarak anlatabilmek için şu örneğe bir bakalım:

```
>>> liste = ["çilek","fıstık", "kebab"]

>>> for i in liste:
...     nesne=re.search("\w*",i)
...     if nesne:
...         print nesne.group()
...
f
kebab
```

Burada alfanümerik karakterler içeren öğeleri ayıklamaya çalıştık. Normalde çıktımız “çilek fıstık kebab” şeklinde olmalıydı. Ama “çilek”teki “ç” ve “fıstık”taki “ı” harfleri Türkçe’ye özgü harfler olduğu için düzenli ifade motoru bu harfleri tanımadı.

İşte burada imdadımıza re.UNICODE seçeneği yetişecek.

Önce GNU/Linux kullanıcılarının bu seçeneği nasıl kullanabileceğine bir örnek verelim:

```
# -*- coding: utf-8 -*-

import re
import locale
locale.setlocale(locale.LC_ALL, "")

liste = ["çilek","fıstık", "kebab"]

for i in liste:
```



```
liste[liste.index(i)] = unicode(i,"utf8")

derle = re.compile("\w*",re.UNICODE)

for i in liste:
    nesne=derle.search(i)
    if nesne:
        print nesne.group()
```

Burada hangi işlemleri yaptığımıza çok dikkat edin. Yukarıdaki kodlarda geçen `import locale` ve ardından gelen satır özellikle önemli. Daha sonra liste öğelerini nasıl “unicode” haline getirdiğimize bakın. Liste öğelerini unicode haline getirmesek alacağımız çıktı yine istediğimiz gibi olmayacaktır. (Benzer bir konu için bkz: <http://www.istihza.com/py2/lenascii.html>). Ayrıca düzenli ifade kalıbımızı derlerken de, çıktıyı düzgün alabilmek için `re.UNICODE` seçeneğini kullandık...

Şimdi aynı şeyi Windows kullanıcılarının nasıl yapacağına bakalım:

```
#-*-coding:cp1254-*-

import locale
locale.setlocale(locale.LC_ALL, "")

import re

liste = ["çilek", "fıstık", "kebab"]

derle = re.compile("\w*",re.UNICODE)

for i in liste:
    nesne = derle.search(i)
    if nesne:
        print nesne.group()
```

Gördüğünüz gibi, GNU/Linux ve Windows’taki kodlar arasında bazı ufak tefek farklılıklar var. Bu farklılıkların nedeni iki sistemde Türkçe karakterleri göstermek için takip edilen yolun farklı olması...

## 19.6 Düzenli İfadelerle Metin/Karakter Dizisi Değiştirme İşlemleri

### 19.6.1 sub() metodu

Şimdiye kadar hep düzenli ifadeler yoluyla bir karakter dizisini nasıl eşleştireceğimizi inceledik. Ama tabii ki düzenli ifadeler yalnızca bir karakter dizisi “bulmak”la ilgili değildir. Bu araç aynı zamanda bir karakter dizisini “değiştirmeyi” de kapsar. Bu iş için temel olarak iki metot kullanılır. Bunlardan ilki `sub()` metodudur. Bu bölümde `sub()` metodunu inceleyeceğiz. En basit şekliyle `sub()` metodunu şu şekilde kullanabiliriz:

```
>>> a = "Kırmızı başlıklı kız, kırmızı elma dolu sepetiyle \
... anneannesinin evine gidiyormuş!"

>>> derle = re.compile("kırmızı",re.IGNORECASE|re.UNICODE)
```

```
>>> print derle.sub("yeşil",a)
```

Burada karakter dizimiz içinde geçen bütün “kırmızı” kelimelerini “yeşil” kelimesiyle değiştirdik. Bunu yaparken de re.IGNORECASE ve re.UNICODE adlı derleme seçeneklerinden yararlandık. Bu ikisini yan yana kullanırken “|” adlı metakarakterden faydalandığımıza dikkat edin.

Elbette sub() metoduyla daha karmaşık işlemler yapılabilir. Bu noktada şöyle bir hatırlatma yapalım. Bu sub() metodu karakter dizilerinin replace() metoduna çok benzer. Ama tabii ki sub() metodu hem kendi başına replace() metodundan çok daha güçlüdür, hem de beraber kullanılabilecek derleme seçenekleri sayesinde replace() metodundan çok daha esnektir. Ama tabii ki, eğer yapmak istediğiniz iş replace() metoduyla halledilebiliyorsa en doğru yol, replace() metodunu kullanmaktır...

Şimdi bu sub() metodunu kullanarak biraz daha karmaşık bir işlem yapacağız. Aşağıdaki metne bakalım:

```
metin = """Karadeniz Ereğlisi denince akla ilk olarak kömür ve demir-çelik gelir. Kokusu ve tadıyla dünyaya nam salmış meşhur Osmanlı çileği ise ismini verdiği festival günleri dışında pek hatırlanmaz. Oysa Çin’ den Arnavutköy’e oradan da Ereğli’ye getirilen kralların meyvesi çilek, burada geçirdiği değişim sonucu tadına doyulmaz bir hal alır. Ereğli’nin havasından mı suyundan mı bilinmez, kokusu, tadı bambaşka bir hale dönüşür ve meşhur Osmanlı çileği unvanını hak eder. Bu nazik ve aromalı çilekten yapılan reçel de likör de bir başka olur.
```

```
Bu yıl dokuzuncusu düzenlenen Uluslararası Osmanlı Çileği Kültür Festivali’nde 36 üretici arasında yetiştirdiği çileklerle birinci olan Kocaali Köyü’nden Güner Özdemir, yılda bir ton ürün alıyor. 60 yaşındaki Özdemir, çileklerinin sırrını yoğun ilgiye ve içten duyduğu sevgiye bağlıyor: "Erkekler bahçemize giremez. Koca ayaklarıyla ezerler çileklerimizi"
```

```
Çileği toplamanın zor olduğunu söyleyen Ayşe Özhan da çocukluğundan bu yana çilek bahçesinde çalışıyor. Her sabah 04.00’te kalkan Özhan, çileklerini özenle suluyor. Kasım başında ektiği çilek fideleri haziran başında meyve veriyor."""
```

---

**Not:** Bu metin <http://www.radikal.com.tr/haber.php?haberno=40130> adresinden alınmıştır.

---

Gelin bu metin içinde geçen “çilek” kelimelerini “erik” kelimesi ile değiştirelim. Ama bunu yaparken, metin içinde “çilek” kelimesinin “Çilek” şeklinde de geçtiğine dikkat edelim. Ayrıca Türkçe kuralları gereği bu “çilek” kelimesinin bazı yerlerde ünsüz yumuşamasına uğrayarak “çileğ-” şekline dönüştüğünü de unutmayalım.

Bu metin içinde geçen “çilek” kelimelerini “erik”le değiştirmek için birkaç yol kullanabilirsiniz. Birinci yolda, her değişiklik için ayrı bir düzenli ifade oluşturulabilir. Ancak bu yolun dezavantajı, metnin de birkaç kez kopyalanmasını gerektirmesidir. Çünkü ilk düzenli ifade oluşturulup buna göre metinde bir değişiklik yapıldıktan sonra, ilk değişiklikleri içeren metnin, farklı bir metin olarak kopyalanması gerekir (metin2 gibi...). Ardından ikinci değişiklik yapılacağı zaman, bu değişikliğin metin2 üzerinden yapılması gerekir. Aynı şekilde bu metin de, mesela, metin3 şeklinde tekrar kopyalanmalıdır. Bundan sonraki yeni bir değişiklik de bu metin3 üzerinden yapılacaktır... Bu durum bu şekilde uzar gider... Metni tekrar tekrar kopyalamak yerine, düzenli ifadeleri kullanarak şöyle bir çözüm de üretebiliriz:

```
#-*-coding:utf-8-*-
```

```
import locale
locale.setlocale(locale.LC_ALL, "")
```

```

import re

metin_uni=unicode(metin,"utf8")

derle = re.compile(u"çile[kğ]",re.UNICODE|re.IGNORECASE)

def degistir(nesne):
    a = {u"çileğ":u"eriğ", u"Çileğ":u"Eriğ",
          u"Çilek":u"Erik", u"çilek":u"erik"}

    b = nesne.group().split()
    for i in b:
        return a[i]

print derle.sub(degistir,metin_uni)

```

Yukarıdaki çözüm GNU/Linux kullanıcıları içindir. Windows kullanıcıları aynı kodu ufak değişikliklerle şöyle yazabilir:

```

#-*-coding:cp1254-*-

import locale
locale.setlocale(locale.LC_ALL,"")

import re

derle = re.compile("çile[kğ]",re.UNICODE|re.IGNORECASE)

def degistir(nesne):
    a = {"çileğ":"eriğ", "Çileğ":"Eriğ",
          "Çilek":"Erik", "çilek":"erik"}

    b = nesne.group().split()
    for i in b:
        return a[i]

print derle.sub(degistir,metin)

```

Gördüğünüz gibi, sub() metodu, argüman olarak bir fonksiyon da alabiliyor. Yukarıdaki kodlar biraz karışık görünmüş olabilir. Tek tek açıklayalım...

Öncelikle şu satıra bakalım:

```

derle = re.compile("çile[kğ]",re.UNICODE|re.IGNORECASE)

```

Burada amacımız, metin içinde geçen “çilek” ve “çileğ” kelimelerini bulmak. Neden “çileğ”? Çünkü “çilek” kelimesi bir sesli harften önce geldiğinde sonundaki “k” harfi “ğ”ye dönüşüyor. Bu seçenekli yapıyı, daha önceki bölümlerde gördüğümüz “[ ]” adlı metakarakter yardımıyla oluşturduk. Düzenli ifade kalıbımızın hem büyük harfleri hem de küçük harfleri aynı anda bulması için re.IGNORECASE seçeneğinden yararlandık. Ayrıca Türkçe harfler üzerinde işlem yapacağımız için de re.UNICODE seçeneğini kullanmayı unutmadık. Bu iki seçeneği “[ ]” adlı metakarakterle birbirine bağladığımıza dikkat edin... Şimdi de şu satırlara bakalım:

```

def degistir(nesne):
    a = {"çileğ":"eriğ", "Çileğ":"Eriğ",
          "Çilek":"Erik", "çilek":"erik"}

    b = nesne.group().split()

```

```
for i in b:
    return a[i]
```

Burada, daha sonra sub() metodu içinde kullanacağımız fonksiyonu yazıyoruz. Fonksiyonu, def degistir(nesne) şeklinde tanımladık. Burada “nesne” adlı bir argüman kullanmamızın nedeni, fonksiyon içinde group() metodunu kullanacak olmamız. Bu metodu fonksiyon içinde “nesne” adlı argümana bağlayacağız. Bu fonksiyon, daha sonra yazacağımız sub() metodu tarafından çağrıldığında, yaptığımız arama işlemi sonucunda ortaya çıkan “eşleşme nesnesi” fonksiyona atanacaktır (eşleşme nesnesinin ne demek olduğunu ilk bölümlerden hatırlıyorsunuz). İşte “nesne” adlı bir argüman kullanmamızın nedeni de, eşleşme nesnelerinin bir metodu olan group() metodunu fonksiyon içinde kullanabilmek...

Bir sonraki satırda bir adet sözlük görüyoruz:

```
a = {"çileğ":"eriğ", "Çileğ":"Eriğ",
     "çilek":"Erik", "Çilek":"erik"}
```

Bu sözlüğü oluşturmamızın nedeni, metin içinde geçen bütün “çilek” kelimelerini tek bir “erik” kelimesiyle değiştiremeyecek olmamız... Çünkü “çilek” kelimesi metin içinde pek çok farklı biçimde geçiyor. Başta da dediğimiz gibi, yukarıdaki yol yerine metni birkaç kez kopyalayıp ve her defasında bir değişiklik yaparak da sorunu çözebilirsiniz. (Mesela önce “çilek” kelimelerini bulup bunları “erik” ile değiştirirsiniz. Daha sonra “çileğ” kelimelerini arayıp bunları “eriğ” ile değiştirirsiniz, vb...) Ama metni tekrar tekrar oluşturmak pek performanslı bir yöntem olmayacaktır. Bizim şimdi kullandığımız yöntem metin kopyalama zorunluluğunu ortadan kaldırıyor. Bu sözlük içinde “çilek” kelimesinin alacağı şekilleri sözlük içinde birer anahtar olarak, “erik” kelimesinin alacağı şekilleri ise birer “değer” olarak belirliyoruz. Bu arada GNU/Linux kullanıcıları Türkçe karakterleri düzgün görüntüleyebilmek için bu sözlükteki öğeleri unicode olarak belirliyor (u“erik” gibi...) Sonraki satırda iki metot birden var:

```
b = nesne.group().split()
```

Burada, fonksiyonumuzun argümanı olarak vazife gören eşleşme nesnesine ait metotlardan biri olan group() metodunu kullanıyoruz. Böylece derle = re.compile("çile[kğ]", re.UNICODE|re.IGNORECASE) satırı yardımıyla metin içinde bulduğumuz bütün “çilek” ve çeşnilerini alıyoruz. Karakter dizilerinin split() metodunu kullanmamızın nedeni ise group() metodunun verdiği çıktıyı liste haline getirip daha kolay manipüle etmek. Burada for i in b:print i komutunu vererseniz group() metodu yardımıyla ne bulduğumuzu görebilirsiniz:

```
çileğ
çilek
çileğ
çilek
Çileğ
çilek
çilek
çilek
çilek
Çileğ
çilek
çilek
çilek
```

Bu çıktıyı gördükten sonra, kodlarda yapmaya çalıştığımız şey daha anlamlı görünmeye başlamış olmalı... Şimdi sonraki satıra geçiyoruz:

```
for i in b:
    return a[i]
```

Burada, `group()` metodu yardımıyla bulduğumuz eşleşmeler üzerinde bir for döngüsü oluşturduk. Ardından da `return a[i]` komutunu vererek “a” adlı sözlük içinde yer alan öğeleri yazdırıyoruz. Bu arada, buradaki “i”nin yukarıda verdiğimiz `group()` çıktılarını temsil ettiğine dikkat edin. `a[i]` gibi bir komut verdiğimizde aslında sırasıyla şu komutları vermiş oluyoruz:

```
a["çilek"]  
a["çileğ"]  
a["çilek"]  
a["çileğ"]  
a["çilek"]  
a["çilek"]  
a["çilek"]  
a["çileğ"]  
a["çilek"]  
a["çilek"]
```

Bu komutların çıktıları sırasıyla *erik, eriğ, erik, Eriğ, erik, erik, erik, Eriğ, erik, erik* olacaktır. İşte bu `return` satırı bir sonraki kod olan `print derle.sub(degistir,metin)` ifadesinde etkinlik kazanacak. Bu son satırımız sözlük öğelerini tek tek metne uygulayacak ve mesela `a["çilek"]` komutu sayesinde metin içinde “çilek” gördüğü yerde “erik” kelimesini yapıştıracak ve böylece bize istediğimiz şekilde değiştirilmiş bir metin verecektir...

Bu kodların biraz karışık gibi göründüğünü biliyorum, ama aslında çok basit bir mantığı var: `group()` metodu ile metin içinde aradığımız kelimeleri ayıklıyor. Ardından da “a” sözlüğü içinde bunları anahtar olarak kullanarak “çilek” ve çeşitleri yerine “erik” ve çeşitlerini koyuyor...

Yukarıda verdiğimiz düzenli ifadeyi böyle ufak bir metinde kullanmak çok anlamlı olmayabilir. Ama çok büyük metinler üzerinde çok çeşitli ve karmaşık değişiklikler yapmak istediğinizde bu kodların işinize yarayabileceğini göreceksiniz.

### 19.6.2 subn() metodu

Bu metodu çok kısa bir şekilde anlatıp geçeceğiz. Çünkü bu metot `sub()` metoduyla neredeyse tamamen aynıdır. Tek farkı, `subn()` metodunun bir metin içinde yapılan değişiklik sayısını da göstermesidir. Yani bu metodu kullanarak, kullanıcılarınıza “*toplam şu kadar sayıda değişiklik yapılmıştır*” şeklinde bir bilgi verebilirsiniz. Bu metot çıktı olarak iki öğeli bir demet verir. Birinci öğe değiştirilen metin, ikinci öğe ise yapılan değişiklik sayısıdır. Yani kullanıcıya değişiklik sayısını göstermek için yapmanız gereken şey, bu demetin ikinci öğesini almaktır. Mesela `sub()` metodunu anlatırken verdiğimiz kodların son satırını şöyle değiştirebilirsiniz:

```
ab = derle.subn(degistir,metin_uni)  
  
print "Toplam %s değişiklik yapılmıştır."%ab[1]
```

## 19.7 Sonuç

Böylelikle düzenli ifadeler konusunu bitirmiş olduk. Buradaki amacımız, size düzenli ifadeler konusunda genel bir bakış sunabilmektir. Bu yazıları okuduktan sonra kafanızda düzenli ifadelerle ilgili kabataslak da olsa bir resim oluştuysa bu yazılar amacına ulaşmış demektir. Elbette düzenli ifadeler burada anlattıklarımızdan ibaret değildir. Bu konunun üzerine eğildiğinizde aslında düzenli ifadelerin dipsiz bir kuyu gibi olduğunu göreceksiniz. Esasında en başta da dediğimiz gibi, düzenli ifadeler apayrı bir dil gibidir. Doğrusu şu ki, düzenli ifadeler başlı başına bağımsız bir sistemdir. Hemen hemen bütün programlama dilleri öyle ya da böyle düzenli ifadeleri destekler. Python’da düzenli ifadeleri bünyesine adapte etmiş dillerden biridir.

Bizim düzenli ifadeler konusundaki yaklaşımımız, her zaman bunları “gerektiğinde” kullanmak olmalıdır. Dediğimiz gibi, eğer yapmak istediğiniz bir işlemi karakter dizilerinin metotları yardımıyla yapabiliyorsanız düzenli ifadelere girişmemek en iyisidir. Çünkü karakter dizisi metotları hem daha hızlıdır hem de anlaması daha kolaydır.

---

# Nesne Tabanlı Programlama – OOP (NTP)

---

Bu yazımızda çok önemli bir konuyu işlemeye başlayacağız: Python’da “Nesne Tabanlı Programlama” (Object Oriented Programming). Yabancılar bu ifadeyi “OOP” olarak kısaltıyor. Gelin isterseniz biz de bunu Türkçe’de NTP olarak kısaltalım. . .

Şimdilik bu “Nesne Tabanlı Programlama”nın ne olduğu ve tanımı bizi ilgilendirmiyor. Biz şimdilik işin teorisiyle pek uğraşmayıp pratiğine bakacağız. NTP’nin pratikte nasıl işlediğini anlarsak, teorisini araştırıp öğrenmek de daha kolay olacaktır.

## 20.1 Neden Nesne Tabanlı Programlama?

İsterseniz önce kendimizi biraz yüreklendirip cesaretlendirelim. Şu soruyu soralım kendimize: Nesne Tabanlı Programlama’ya hiç girmesem olmaz mı?

Bu soruyu cevaplandırmadan önce bakış açımızı şöyle belirleyelim. Daha doğrusu bu soruyu iki farklı açıdan inceleyelim: NTP’yi öğrenmek ve NTP’yi kullanmak...

Eğer yukarıdaki soruya, “NTP’yi kullanmak” penceresinden bakarsak, cevabımız, “Evet,” olacaktır. Yani, “Evet, NTP’yi kullanmak zorunda değilsiniz”. Bu bakımdan NTP’yle ilgilenmek istemeyebilirsiniz, çünkü Python başka bazı dillerin aksine NTP’yi dayatmaz. İyi bir Python programcısı olmak için NTP’yi kullanmasanız da olur. NTP’yi kullanmadan da gayet başarılı programlar yazabilirsiniz. Bu bakımdan önünüzde bir engel yok.

Ama eğer yukarıdaki soruya “NTP’yi öğrenmek” penceresinden bakarsak, cevabımız, “Hayır”, olacaktır. Yani, “Hayır, NTP’yi öğrenmek zorundasınız!”. Bu bakımdan NTP’yle ilgilenmeniz gerekir, çünkü siz NTP’yi kullanmasanız da başkaları bunu kullanıyor. Dolayısıyla, NTP’nin bütün erdemlerini bir kenara bıraksak dahi, sırf başkalarının yazdığı kodları anlayabilmek için bile olsa, elinizi NTP’yle kirletmeniz gerekecektir... Bir de şöyle düşünün: Gerek internet üzerinde olsun, gerekse basılı yayınlarda olsun, Python’a ilişkin pek çok kaynakta kodlar bir noktadan sonra NTP yapısı içinde işlenmektedir. Bu yüzden özellikle başlangıç seviyesini geçtikten sonra karşınıza çıkacak olan kodları anlayabilmek için bile NTP’ye bir aşinalığınızın olması gerekir.

Dolayısıyla en başta sorduğumuz soruya karşılık ben de size şu soruyu sormak isterim:

“Daha nereye kadar kaçacaksınız bu NTP’den?”

Dikkat ederseniz, bildik anlamda NTP'nin faydalarından, bize getirdiği kolaylıklardan hiç bahsetmiyoruz. Zira şu anda içinde bulunduğumuz noktada bunları bilmenin bize pek faydası dokunmayacaktır. Çünkü daha NTP'nin ne olduğunu dahi bilmiyoruz ki cicili bicili cümlelerle bize anlatılacak “faydaları” özümseyebilelim... NTP'yi öğrenmeye çalışan birine birkaç sayfa boyunca “NTP şöyle iyidir, NTP böyle hoştur,” demenin pek faydası olmayacaktır. Çünkü böyle bir çaba, konuyu anlatan kişiyi ister istemez okurun henüz bilmediği kavramları kullanarak bazı şeyleri açıklamaya çalışmaya itecektir. Bu da okurun zihninde birtakım fantastik cümlelerin uçuşmasından başka bir işe yaramayacaktır. Dolayısıyla, NTP'nin faydalarını size burada bir çırpıda saymak yerine, öğrenme sürecine bırakıyoruz bu “özümseme” işini... NTP'yi öğrendikten sonra, bu programlama tekniğinin Python deneyiminize ne tür bir katkı sağlayacağını, size ne gibi bir fayda getireceğini kendi gözlerinizle göreceksiniz.

En azından biz bu noktada şunu rahatlıkla söyleyebiliriz: NTP'yi öğrendiğinizde Python Programlama'da bir anlamda “boyut atlamış” olacaksınız. Sonunda özgüveniniz artacak, orada burada Python'a ilişkin okuduğunuz şeyler zihninizde daha anlamlı izler bırakmaya başlayacaktır.

## 20.2 Sınıflar

NTP'de en önemli kavram “sınıflar”dır. Zaten NTP denince ilk akla gelen şey de genellikle “sınıflar” olmaktadır. Sınıflar yapı olarak “fonksiyonlara” benzetilebilir. Hatırlarsanız, fonksiyonlar yardımıyla farklı değişkenleri ve veri tiplerini, tekrar kullanılmak üzere bir yerde toplayabiliyorduk. İşte sınıflar yardımıyla da farklı fonksiyonları, veri tiplerini, değişkenleri, metotları gruplandırabiliyoruz.

### 20.2.1 Sınıf Tanımlamak

Öncelikle bir sınıfı nasıl tanımlayacağımıza bakmamız gerekiyor. Hemen, bir sınıfı nasıl tanımlayacağımızı bir örnekle görmeye çalışalım:

Python'da bir sınıf oluşturmak için şu yapıyı kullanıyoruz:

```
class IlkSinif:
```

Böylece sınıfları oluşturmak için ilk adımı atmış olduk. Burada dikkat etmemiz gereken bazı noktalar var:

Hatırlarsanız fonksiyonları tanımlarken def parçacığından yararlanıyorduk. Mesela:

```
def deneme():
```

Sınıfları tanımlarken ise class parçacığından faydalanıyoruz:

```
class IlkSinif:
```

Tıpkı fonksiyonlarda olduğu gibi, isim olarak herhangi bir kelimeyi seçebiliriz. Mesela yukarıdaki fonksiyonda “deneme” adını seçmiştik. Yine yukarıda gördüğümüz sınıf örneğinde de “IlkSinif” adını kullandık. Tabii isim belirlerken Türkçe karakter kullanamıyoruz. . .

Sınıf adlarını belirlerken kullanacağımız kelimenin büyük harf veya küçük harf olması önemli değildir. Ama seçilen kelimelerin ilk harflerini büyük yazmak adettendir. Mesela “class Sinif” veya “class HerhangiBirKelime”. Gördüğünüz gibi sınıf adı birden fazla kelimeden oluşuyorsa her kelimenin ilk harfi büyük yazılıyor. Bu bir kural değildir, ama her zaman adetlere uymak yerinde bir davranış olacaktır...



Son olarak, sınıfımızı tanımladıktan sonra parantez işareti kullanmak zorunda olmadığımıza dikkat edin. En azından şimdilik... Bu parantez meselesine tekrar döneceğiz.

İlk adımı attığımıza göre ilerleyebiliriz:

```
class İlkSinif:  
    mesele = "Olmak ya da olmamak"
```

Böylece eksiksiz bir sınıf tanımlamış olduk. Aslında tabii ki normalde sınıflar bundan biraz daha karmaşıktır. Ancak yukarıdaki örnek, gerçek hayatta bu haliyle karşımıza çıkmayacak da olsa, hem yapı olarak kurallara uygun bir sınıftır, hem de bize sınıflara ilişkin pek çok önemli ipucu vermektedir. Sırasıyla bakalım:

İlk satırda doğru bir şekilde sınıfımızı tanımladık.

İkinci satırda ise “mesele” adlı bir değişken oluşturduk.

Böylece ilk sınıfımızı başarıyla tanımlamış olduk.

## 20.2.2 Sınıfları Çalıştırmak

Şimdi güzel güzel yazdığımız bu sınıfı nasıl çalıştıracığımıza bakalım:

Herhangi bir Python programını nasıl çalıştırıyorsak sınıfları da öyle çalıştırabiliriz. Yani pek çok farklı yöntem kullanabiliriz. Örneğin yazdığımız şey arayüzü olan bir Tkinter programıysa “python programadı.py” komutuyla bunu çalıştırabilir, yazdığımız arayüzü görebiliriz. Hatta gerekli ayarlamaları yaptıktan sonra programın simgesine çift tıklayarak veya GNU/Linux sistemlerinde konsol ekranında programın sadece adını yazarak çalıştırabiliriz programımızı. Eğer komut satırından çalışan bir uygulama yazdıysak, yine “python programadı.py” komutuyla programımızı çalıştırıp konsol üzerinden yönetebiliriz. Ancak bizim şimdilik yazdığımız kodun bir arayüzü yok. Üstelik bu sadece NTP’yi öğrenmek için yazdığımız, tam olmayan bir kod parçasından ibaret. Dolayısıyla sınıfımızı tecrübe etmek için biz şimdilik doğrudan Python komut satırı içinden çalışacağız.

Şu halde herkes kendi platformuna uygun şekilde Python komut satırını başlatsın! Python’u başlattıktan sonra bütün platformlarda şu komutu vererek bu kod parçasını çalıştırılabilir duruma getirebiliriz:

```
from sinif import *
```

Burada sizin bu kodları “sinif.py” adlı bir dosyaya kaydettiğinizi varsaydım. Dolayısıyla bu şekilde dosyamızı bir modül olarak içe aktarabiliyoruz (import). Bu arada Python’un bu modülü düzgün olarak içe aktarabilmesi için komut satırını, bu modülün bulunduğu dizin içinde açmak gerekir. Python içe aktarılacak modülleri ararken ilk olarak o anda içinde bulunulan dizine bakacağı için modülümüzü rahatlıkla bulabilecektir.

GNU/Linux kullanıcıları komut satırıyla daha içli dışlı oldukları için etkileşimli kabuğu modülün bulunduğu dizinde nasıl açacaklarını zaten biliyorlardır... Ama biz yine de hızlıca üzerinden geçelim...(Modülün masaüstünde olduğunu varsayıyoruz):

ALT+F2 tuşlarına basıp açılan pencereye “konsol” (KDE) veya “gnome-terminal” (GNOME) yazıyoruz. Ardından konsol ekranında “cd Desktop” komutunu vererek masaüstüne erişiyoruz. Windows kullanıcılarının komut satırına daha az aşina olduğunu varsayarak biraz daha detaylı anlatalım bu işlemi...

Windows kullanıcıları ise Python komut satırını modülün olduğu dizin içinde açmak için şu yolu izleyebilir (yine modülün masaüstünde olduğunu varsayarsak...): *Başlat > Çalıştır* yolunu takip edip açılan kutuya “cmd” yazıyoruz (parantezler olmadan). Komut ekranı karşımıza gelecek.

Muhtemelen içinde bulunduğunuz dizin “C:Documents and Settingsİsminiz” olacaktır. Orada şu komutu vererek masaüstüne geçiyoruz:

```
cd Desktop
```

Şimdi de şu komutu vererek Python komut satırını başlatıyoruz:

```
C:/python25/python
```

Tabii kullandığınız Python sürümünün 2.5 olduğunu varsaydım. Sizde sürüm farklıysa komutu ona göre değiştirmelisiniz.

Eğer herhangi bir hata yapmadıysanız karşınıza şuna benzer bir ekran gelmeli:

```
C:\Documents and Settings\İsminiz>c:/python25/Python
Python 2.5.1 (r251:54863, Apr 18 2007, 08:51:08) [MSC v.1310 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Şimdi bu ekrandaki “>>>” satırından hemen sonra şu komutu verebiliriz:

```
>>> from sinif import *
```

Artık sınıfımızı çalıştırmamızın önünde hiç bir engel kalmadı sayılır. Bu noktada yapmamız gereken tek bir işlem var: Örneklem

## 20.3 Örneklem (Instantiation)

Şimdi şöyle bir şey yazıyoruz:

```
deneme = IlkSinif()
```

Böylece oluşturduğumuz sınıfı bir değişkene atadık. NTP kavramlarıyla konuşacak olursak, “sınıfımızı örneklemiş olduk”.

Peki bu “örneklem” denen şey de ne oluyor? Hemen bakalım:

İngilizce’de “instantiation” olarak ifade edilen “örneklem” kavramı sayesinde sınıfımızı kullanırken belli bir kolaylık sağlamış oluyoruz. Gördüğünüz gibi, “örneklem” (instantiation) aslında şekil olarak yalnızca bir değişken atama işleminden ibarettir. Nasıl daha önce gördüğümüz değişkenler uzun ifadeleri kısaca adlandırmamızı sağlıyorsa, burada da “örneklem” işlemi hemen hemen aynı vazifeyi görüyor. Yani böylece ilk satırda tanımladığımız sınıfa daha kullanışlı bir isim vermiş oluyoruz. Dediğimiz gibi, bu işleme “örneklem” (instantiation) adı veriliyor. Bu örneklemelerin her birine ise “örnek” (instance) deniyor. Yani, IlkSinif adlı sınıfa bir isim verme işlemine “örneklem” denirken, bu işlem sonucu ortaya çıkan değişkene de, “örnek” (instance) diyoruz. Buna göre, burada “deneme” adlı değişken, “IlkSinif” adlı sınıfın bir örneğidir (“deneme” is an instance of the class “IlkSinif”). Daha soyut bir ifadeyle, örneklem işlemi “Class” (sınıf) nesnesini etkinleştirmeye yarar. Yani sınıfın bütününe alır ve onu paketleyip, istediğimiz şekilde kullanabileceğimiz bir nesne haline getirir. Şöyle de diyebiliriz:

Biz bir sınıf tanımlıyoruz. Bu sınıfın içinde birtakım değişkenler, fonksiyonlar, vb. olacaktır. Hayli kaba bir benzetme olacak ama, biz bunları bir internet sayfasının içeriğine benzetebiliriz. İşte biz bu sınıfı “örneklediğimiz” zaman, sınıf içeriğini bir bakıma erişilebilir hale getirmiş oluyoruz. Tıpkı bir internet sayfasının, “www...” şeklinde gösterilen adresi gibi... Mesela [www.python.quotaless.com](http://www.python.quotaless.com) adresi içindeki bütün bilgileri bir sınıf olarak düşünürsek, “www.python.quotaless.com” ifadesi bu sınıfın bir örneğidir... Durum tam olarak böyle olmasa

bile, bu benzetme, “örnekleme” işlemine ilişkin en azından zihnimizde bir kıvılcım çakmasını sağlayabilir.

Daha yerinde bir benzetme şöyle olabilir: “İnsan”ı büyük bir sınıf olarak kabul edelim. İşte “siz” (yani Ahmet, Mehmet, vb...) bu büyük sınıfın bir örneği, yani ete kemiğe bürünmüş hali oluyor-sunuz... Buna göre “insan” sınıfı insanın ne tür özellikleri olduğuna dair tanımlar (fonksiyonlar, veriler) içeriyor. “Mehmet” örneği (instance) ise bu tanımları, nitelikleri, özellikleri taşıyan bir “nesne” oluyor...

## 20.4 Çöp Toplama (Garbage Collection)

Peki biz bir sınıfı örneklemezsek ne olur? Eğer bir sınıfı örneklemezsek, o örneklenmeyen sınıf program tarafından otomatik olarak “çöp toplama” (garbage collection) adı verilen bir süreç tabi tutulacaktır. Burada bu sürecin ayrıntılarına girmeyeceğiz. Ama kısaca şöyle anlatabiliriz: Python’da (ve bir çok programlama dilinde) yazdığımız programlar içindeki “işe yaramayan” veriler bellekten silinir. Böylece etkili bir hafıza yönetimi uygulanmış ve programların performansı artırılmış olur. Mesela:

```
>>> a = 5
>>> a = a + 6
>>> print a
```

11

Burada “a” değişkeninin gösterdiği “5” verisi, daha sonra gelen “a = a + 6” ifadesi nedeniyle boşa düşmüş, ıskartaya çıkmış oluyor. Yani “a = a + 6” ifadesi nedeniyle, “a” değişkeni artık “5” verisini göstermiyor. Dolayısıyla “5” verisi o anda bellekte boşu boşuna yer kaplamış oluyor. Çünkü “a = a + 6” ifadesi yüzünden, “5” verisine gönderme yapan, onu gösteren, bu veriye bizim ulaşmamızı sağlayacak hiç bir işaret kalmamış oluyor ortada. İşte Python, bir veriye işaret eden hiç bir referans kalmadığı durumlarda, yani o veri artık işe yaramaz hale geldiğinde, otomatik olarak “çöp toplama” işlemini devreye sokar ve bu örnekte “5” verisini çöpe gönderir. Yani artık o veriyi bellekte tutmaktan vazgeçer. İşte eğer biz de yukarıda olduğu gibi sınıflarımızı “örneklemezsek”, bu sınıflara hiçbir yerde işaret edilmediği, yani bu sınıfı gösteren hiçbir “referans” olmadığı için, sınıfımız oluşturulduğu anda çöp toplama işlemine tabi tutulacaktır. Dolayısıyla artık bellekte tutulmayacaktır.

“Çöp Toplama” işlemini de kısaca anlattığımıza göre artık kaldığımız yerden yolumuza devam edebiliriz...

Bu arada dikkat ettiyseniz sınıfımızı örneklerken parantez kullandık. Yani şöyle yaptık:

```
>>> deneme = IlkSinif()
```

Eğer parantezleri kullanmazsak, yani “deneme = IlkSinif” gibi bir şey yazarsak, yaptığımız şey “örnekleme” olmaz. Böyle yaparak sınıfı sadece kopyalamış oluruz... Bizim yapmak istediğimiz bu değil. O yüzden, “parantezlere dikkat!” diyoruz...

Artık şu komut yardımıyla, sınıf örneğimizin “niteliklerine” ulaşabiliriz:

```
>>> deneme.mesele
```

Olmak ya da olmamak

## 20.5 Niteliklere Değınme (Attribute References)

Biraz önce “nitelik” diye bir şeyden söz ettik. İngilizce’de “attribute” denen bu “nitelik” kavramı, Python’daki nesnelerin özelliklerine işaret eder. Python’un yazarı Guido Van Rossum bu kavram için şöyle diyor:

“I use the word attribute for any name following a dot” (Noktadan sonra gelen bütün isimler için ben “nitelik” kelimesini kullanıyorum) kaynak: <http://docs.python.org/tut/node11.html>

Bu tanıma göre, örneğın:

```
>>> deneme.mesele
```

dediğımız zaman, buradaki “mesele”; “deneme” adlı sınıf örneğının (instance) bir niteliğı (attribute) oluyor. Biraz karışık gibi mi? Hemen bir örnek yapalım o halde:

```
class Toplama:  
    a = 15  
    b = 20  
    c = a + b
```

İlk satırda “Toplama” adlı bir sınıf tanımladık. Bunu yapmak için “class” parçacığından yararlandık. Sırasıyla; a, b ve c adlı üç adet değışken oluşturduk. c değışkeni a ve b değışkenlerinin toplamıdır.

Bu sınıftaki a, b ve c değışkenleri ise, “Toplama” sınıf örneğının (örneğı biraz sonra tanımlayacağız) birer niteliğı oluyor. Bundan önceki örneğımızde ise “mesele” adlı değışken, “deneme” adlı sınıf örneğının bir niteliğı idi...

Bu sınıfı yazıp kaydettiğımız dosyamızın adının “matematik.py” olduğunu varsayarsak;

```
>>> from matematik import *
```

komutunu verdikten sonra şunu yazıyoruz:

```
>>> sonuc = Toplama()
```

Böylece “Toplama” adlı sınıfımızı “örnekliyoruz”. Bu işleme “örnekleme” (instantiation) adı veriyoruz. “sonuc” kelimesine ise Python’cada “örnek” (instance) adı veriliyor. Yani “sonuc”, “Toplama” sınıfının bir örneğidir, diyoruz. . .

Artık,

```
>>> sonuc.a  
>>> sonuc.b  
>>> sonuc.c
```

biçiminde, “sonuc” örneğının niteliklerine tek tek erişebiliriz.

Peki kodları şöyle çalıştırırsak ne olur?

```
>>> import matematik
```

Eğer modül bu şekilde içe aktarırsak (import), sınıf örneğının niteliklerine ulaşmak için şu yapıyı kullanmamız gerekir:

```
>>> matematik.sonuc.a  
>>> matematik.sonuc.b  
>>> matematik.sonuc.c
```

Yani her defasında dosya adını (ya da başka bir ifadeyle “modülün adını”) da belirtmemiz gerekir. Bu iki kullanım arasında, özellikle sağladıkları güvenlik avantajları/dezavantajları açısından başka bazı temel farklılıklar da vardır, ama şimdilik konumuzu dağıtmamak için bunlara girmiyoruz... Ama temel olarak şunu bilmekte fayda var: Genellikle tercih edilmesi gereken yöntem “from modül import \*” yerine “import modül” biçimini kullanmaktır. Eğer “from modül import \*” yöntemini kullanarak içe aktardığınız modül içindeki isimler (değişkenler, nitelikler), bu modülü kullanacağınız dosya içinde de bulunuyorsa isim çakışmaları ortaya çıkabilir... Esasında, “from modül import \*” yapısını sadece ne yaptığımızı çok iyi biliyorsak ve modülle ilgili belgelerde modülün bu şekilde içe aktarılması gerektiği bildiriliyorsa kullanmamız yerinde olacaktır. Mesela Tkinter ile programlama yaparken rahatlıkla “from Tkinter import \*” yapısını kullanabiliriz, çünkü Tkinter bu kullanımda problem yaratmayacak şekilde tasarlanmıştır. Yukarıda bizim verdiğimiz örnekte de “from modül import \*” yapısını rahatlıkla kullanıyoruz, çünkü şimdilik tek bir modül üzerinde çalışıyoruz. Dolayısıyla isim çakışması yaratacak başka bir modülümüz olmadığı için “ne yaptığımızı biliyoruz!”...

Yukarıda anlattığımız kod çalıştırma biçimleri tabii ki, bu kodları komut ekranından çalıştırdığınızı varsaymaktadır. Eğer siz bu kodları IDLE ile çalıştırmak isterseniz, bunları hazırladıktan sonra F5 tuşuna basmanız, veya “Run > Run Module” yolunu takip etmeniz yeterli olacaktır. F5’e bastığınızda veya “Run > Run Module” yolunu takip ettiğinizde IDLE sanki komut ekranında “from matematik import \*” komutunu vermişsiniz gibi davranacaktır.

Veya GNU/Linux sistemlerinde sistem konsolunda:

```
python -i sinif.py
```

komutunu vererek de bu kod parçalarını çalıştırılabilir duruma getirebiliriz. Bu komutu verdiğimizde “from sinif import \*” komutu otomatik olarak verilip hemen ardından Python komut satırı açılacaktır. Bu komut verildiğinde ekranda göreceğiniz “>>>” işaretinden, Python’un sizden hareket beklediğini anlayabilirsiniz...

Şimdi isterseniz buraya kadar söylediklerimizi şöyle bir toparlayalım. Bunu da yukarıdaki örnek üzerinden yapalım:

```
class Toplama:
    a = 15
    b = 20
    c = a + b
```

“Toplama” adlı bir sınıf tanımlıyoruz. Sınıfımızın içine istediğimiz kod parçalarını ekliyoruz. Biz burada üç adet değişken ekledik. Bu değişkenlerin her birine, “nitelik” adını veriyoruz. Bu kodları kullanabilmek için Python komut satırında şu komutu veriyoruz:

```
>>> from matematik import *
```

Burada modül adının (yani dosya adının) matematik olduğunu varsaydık.

Şimdi yapmamız gereken şey, Toplama adlı sınıfı “örneklemek” (instantiation). Yani bir nevi, sınıfın kendisini bir değişkene atamak. Bu değişkene biz Python’da “örnek” (instance) adını veriyoruz. Yani, “sonuc” adlı değişken, “Toplama” adlı sınıfın bir örneğidir diyoruz (sonuc is an instance of Toplama)”

```
>>> sonuc = Toplama()
```

Bu komutu verdikten sonra niteliklerimize erişebiliriz:

```
>>> sonuc.a
>>> sonuc.b
>>> sonuc.c
```

Dikkat ederseniz, niteliklerimize erişirken “örnek”ten (instance), yani “sonuc” adlı değişkenden yararlanıyoruz.

Şimdi bir an bu sınıfımızı örneklemediğimizi düşünelim. Dolayısıyla bu sınıfı şöyle kullanmamız gerekecek:

```
>>> Toplama().a
>>> Toplama().b
>>> Toplama().c
```

Ama daha önce de anlattığımız gibi, siz “Toplama().a” der demez sınıf çalıştırılacak ve çalıştırdıktan hemen sonra ortada bu sınıfa işaret eden herhangi bir referans kalmadığı için Python tarafından “işe yaramaz” olarak algılanan sınıfımız çöp toplama işlemine tabi tutularak derhal belleği terketmesi sağlanacaktır. Bu yüzden bunu her çalıştırdığınızda yeniden belleğe yüklemiş olacaksınız sınıfı. Bu da bir hayli verimsiz bir çalışma şeklidir.

Böylelikle zor kısmı geride bırakmış olduk. Artık önümüze bakabiliriz. Zira en temel bazı kavramları gözden geçirdiğimiz ve temelimizi oluşturduğumuz için, daha karışık şeyleri anlamak kolaylaşacaktır.

## 20.6 \_\_init\_\_ Nedir?

Eğer daha önce etrafta sınıfları içeren kodlar görmüşseniz, bu \_\_init\_\_ fonksiyonuna en azından bir göz aşinalığınız vardır. Genellikle şu şekilde kullanıldığını görürüz bunun:

```
def __init__(self):
```

Biz şimdilik bu yapıdaki \_\_init\_\_ kısmıyla ilgileneneceğiz. “self”in ne olduğunu şimdilik bir kenara bırakıp, onu olduğu gibi kabul edelim. İşe hemen bir örnekle başlayalım. İsterseniz kendimizce ufak bir oyun tasarlayalım:

```
#!/usr/bin/env python
#-*- coding:utf8 -*-

class Oyun:
    def __init__(self):
        enerji = 50
        para = 100
        fabrika = 4
        isci = 10
        print "enerji:", enerji
        print "para:", para
        print "fabrika:", fabrika
        print "işçi:", isci

macera = Oyun()
```

Gayet güzel. Dikkat ederseniz “örnekleme” (instantiation) işlemini doğrudan dosya içinde hal-lettik. Komut satırına bırakmadık bu işi.

Şimdi bu kodları çalıştıracacağız. Bir kaç seçeneğimiz var:

Üzerinde çalıştığımız platforma göre Python komut satırını, yani etkileşimli kabuğu açıyoruz. Orada şu komutu veriyoruz:

```
>>> from deneme import *
```

Burada dosya adının “deneme.py” olduğunu varsaydık. Eğer örnekleme işlemini dosya içinden halletmemiş olsaydık, “from deneme import \*” komutunu verdikten sonra “macera = Oyun()” satırı yardımıyla ilk olarak sınıfımızı örneklendirmemiz gerekecekti.

GNU/Linux sistemlerinde başka bir seçenek olarak, ALT+F2 tuşlarına basıyoruz ve açılan pencerede “konsole” (KDE) veya “gnome-terminal” (GNOME) yazıp enter’e bastıktan sonra açtığımız komut satırında şu komutu veriyoruz:

```
python -i deneme.py
```

Eğer Windows’ta IDLE üzerinde çalışıyorsak, F5 tuşuna basarak veya “Run>Run Module” yolunu takip ederek kodlarımızı çalıştırıyoruz.

Bu kodları yukarıdaki seçeneklerden herhangi biriyle çalıştırdığımızda, `__init__` fonksiyonu içinde tanımlanmış olan bütün değişkenlerin, yani “niteliklerin”, ilk çalışma esnasında ekrana yazdırıldığını görüyoruz. İşte bu niteliklerin başlangıç değeri olarak belirlenebilmesi hep `__init__` fonksiyonu sayesinde olmaktadır. Dolayısıyla şöyle bir şey diyebiliriz:

Python’da bir programın ilk kez çalıştırıldığı anda işlemlerini istediğimiz şeyleri bu `__init__` fonksiyonu içine yazıyoruz. Mesela yukarıdaki ufak oyun çalışmasında, oyuna başlandığı anda bir oyuncunun sahip olacağı özellikleri `__init__` fonksiyonu içinde tanımladık. Buna göre bu oyunda bir oyuncu oyuna başladığında:

enerjisi 50, parası 100 fabrika sayısı 4, işçi sayısı ise 10 olacaktır.

Yalnız hemen uyaralım: Yukarıdaki örnek aslında pek de düzgün sayılmaz. Çok önemli eksiklikleri var bu kodun. Ama şimdi konumuz bu değil... Olayın iç yüzünü kavrayabilmek için öyle bir örnek vermemiz gerekiyordu. Bunu biraz sonra açıklayacağız. Biz okumaya devam edelim...

Bir de Tkinter ile bir örnek yapalım. Zira sınıflı yapıların en çok ve en verimli kullanıldığı yer arayüz programlama çalışmalarıdır:

```
from Tkinter import *

class Arayuz:
    def __init__(self):
        pencere = Tk()
        dugme = Button(text="tamam")
        dugme.pack()

uygulama = Arayuz()
```

Bu kodları da yukarıda saydığımız yöntemlerden herhangi biri ile çalıştırıyoruz. Tabii ki bu kod da eksiksiz değildir. Ancak şimdilik amacımıza hizmet edebilmesi için kodlarımızı bu şekilde yazmamız gerekiyordu. Ama göreceğiniz gibi yine de çalışıyor bu kodlar... Dikkat ederseniz burada da örnekleme işlemini dosya içinden hallettik. Eğer örnekleme satırını dosya içine yazmazsak, Tkinter penceresinin açılması için komut satırında “uygulama = Arayuz()” gibi bir satır yazmamız gerekir.

Buradaki `__init__` fonksiyonu sayesinde “Arayuz” adlı sınıf her çağrıldığında bir adet Tkinter penceresi ve bunun içinde bir adet düğme otomatik olarak oluşacaktır. Zaten bu `__init__` fonksiyonuna da İngilizce’de çoğu zaman “constructor” (oluşturan, inşa eden, meydana getiren) adı verilir. Gerçi `__init__` fonksiyonuna “constructor” demek pek doğru bir ifade sayılmaz, ama biz bunu şimdi bir kenara bırakalım. Sadece aklımızda olsun, `__init__` fonksiyonu gerçek anlamda bir “constructor” değildir, ama ona çok benzer...

Şöyle bir yanlış anlaşılma olmamasına dikkat edin:

“\_\_init\_\_” fonksiyonunun, “varsayılan değerleri belirleme”, yani “inşa etme” özelliği konumundan kaynaklanmıyor. Yani bu \_\_init\_\_ fonksiyonu, işlevini sırf ilk sırada yer aldığı için yerine getirmiyor. Bunu test etmek için, isterseniz yukarıdaki kodları “\_\_init\_\_” fonksiyonunun adını değiştirerek çalıştırmayı deneyin. Aynı işlevi elde edemezsiniz... Mesela \_\_init\_\_ yerine \_\_simit\_\_ deyin. Çalışmaz...

\_\_init\_\_ konusuna biraz olsun ışık tuttuğumuza göre artık en önemli bileşenlerden ikincisine gelebiliriz: self

## 20.7 self Nedir?

Bu küçücük kelime Python’da sınıfların en can alıcı noktasını oluşturur. Esasında çok basit bir işlevi olsa da, bu işlevi kavrayamazsak neredeyse bütün bir sınıf konusunu kavramak imkansız hale gelecektir. Self’i anlamaya doğru ilk adımı atmak için yukarıda kullandığımız kodlardan faydalanarak bir örnek yapmaya çalışalım. Kodumuz şöyleydi:

```
class Oyun:
    def __init__(self):
        enerji = 50
        para = 100
        fabrika = 4
        isci = 10
        print "enerji:", enerji
        print "para:", para
        print "fabrika:", fabrika
        print "işçi:", isci
```

```
macera = Oyun()
```

Diyelim ki biz burada “enerji, para, fabrika, işçi” değişkenlerini ayrı bir fonksiyon içinde kullanmak istiyoruz. Yani mesela göster() adlı ayrı bir fonksiyonumuz olsun ve biz bu değişkenleri ekrana yazdırmak istediğimizde bu “göster” fonksiyonundan yararlanalım. Kodların şu anki halinde olduğu gibi, bu kodlar tanımlansın, ama doğrudan ekrana dökülmesin. Şöyle bir şey yazmayı deneyelim. Bakalım sonuç ne olacak?

```
class Oyun:
    def __init__(self):
        enerji = 50
        para = 100
        fabrika = 4
        isci = 10

    def goster():
        print "enerji:", enerji
        print "para:", para
        print "fabrika:", fabrika
        print "işçi:", isci
```

```
macera = Oyun()
```

Öncelikle bu kodların sahip olduğu “niteliklere” bir bakalım:

enerji, para, fabrika, işçi ve goster()

Burada “örneğimiz” (instance) “macera” adlı değişken. Dolayısıyla bu niteliklere şu şekilde ulaşabiliriz:



```
>>> macera.enerji
>>> macera.para
>>> macera.fabrika
>>> macera.isci
>>> macera.goster()
```

Hemen deneyelim. Ama o da ne? Mesela macera.goster() dediğimizde şöyle bir hata alıyoruz:

```
Traceback (most recent call last):
  File "<pyshell0>", line 1, in <module>
    macera.goster()
TypeError: goster() takes no arguments (1 given)
```

Belli ki bir hata var kodlarımızda. goster() fonksiyonuna bir "self" ekleyerek tekrar deneyelim. Belki düzelir:

```
class Oyun:
    def __init__(self):
        enerji = 50
        para = 100
        fabrika = 4
        isci = 10

    def goster(self):
        print "enerji:", enerji
        print "para:", para
        print "fabrika:", fabrika
        print "işçi:", isci

macera = Oyun()
```

Tekrar deniyoruz:

```
>>> macera.goster()
```

Olmadı... Bu sefer de şöyle bir hata aldık:

```
enerji:
Traceback (most recent call last):
  File "<pyshell0>", line 1, in <module>
    macera.goster()
  File "xxxxxxxxxxxxxxxxxxxx", line 9, in goster
    print "enerji:", enerji
NameError: global name 'enerji' is not defined
```

Hmm... Sorunun ne olduğu az çok ortaya çıktı. Hatırlarsanız buna benzer hata mesajlarını Fonksiyon tanımlarken "global" değişkeni yazmadığımız zamanlarda da alıyorduk. . . İşte "self" burada devreye giriyor. Yani bir bakıma, fonksiyonlardaki global ifadesinin yerini tutuyor. Daha doğru bir ifadeyle, burada "macera" adlı sınıf örneğini temsil ediyor. Artık kodlarımızı düzeltebiliriz:

```
class Oyun:
    def __init__(self):
        self.enerji = 50
        self.para = 100
        self.fabrika = 4
        self.isci = 10
```

```
def goster(self):  
    print "enerji:", self.enerji  
    print "para:", self.para  
    print "fabrika:", self.fabrika  
    print "işçi:", self.isci
```

```
macera = Oyun()
```

Gördüğümüz gibi, kodlar içinde yazdığımız değişkenlerin, fonksiyon dışından da çağrılabilmesi için, yani bir bakıma “global” bir nitelik kazanması için “self” olarak tanımlanmaları gerekiyor. Yani mesela, “enerji” yerine “self.enerji” diyerek, bu “enerji” adlı değişkenin yalnızca içinde bulunduğu fonksiyonda değil, o fonksiyonun dışında da kullanılabilmesini sağlıyoruz. İyice somutlaştırmak gerekirse, “\_\_init\_\_” fonksiyonu içinde tanımladığımız “enerji” adlı değişken, bu haliyle “goster” adlı fonksiyonun içinde kullanılamaz. Daha da önemlisi bu kodları bu haliyle tam olarak çalıştıramayız da. Mesela şu temel komutları işletemeyiz:

```
>>> macera.enerji  
>>> macera.para  
>>> macera.isci  
>>> macera.fabrika
```

Eğer biz “enerji” adlı değişkeni goster() fonksiyonu içinde kullanmak istersek değişkeni sadece “enerji” değil, “self.enerji” olarak tanımlamamız gerekir. Ayrıca bunu “goster” adlı fonksiyon içinde kullanırken de sadece “enerji” olarak değil, “self.enerji” olarak yazmamız gerekir. Üstelik mesela “enerji” adlı değişkeni herhangi bir yerden çağırmak istediğimiz zaman da bunu önceden “self” olarak tanımlamış olmamız gerekir.

Şimdi tekrar deneyelim:

```
>>> macera.goster  
  
enerji: 50  
para: 100  
fabrika: 4  
işçi: 10  
  
>>> macera.enerji  
  
50  
  
>>> macera.para  
  
100  
  
>>> macera.fabrika  
  
4  
  
>>> macera.isci  
  
10
```

Sınıfın niteliklerine tek tek nasıl erişebildiğimizi görüyorsunuz. . . Bu arada, isterseniz “self”i, “macera” örneğinin yerini tutan bir kelime olarak da kurabilirsiniz zihninizde. Yani kodları çalıştırırken “macera.enerji” diyebilmek için, en başta bunu “self.enerji” olarak tanımlamamız gerekiyor... Bu düşünme tarzı işimizi biraz daha kolaylaştırabilir.

Bir de Tkinter’li örneğimize bakalım:

```

from Tkinter import *

class Arayuz:
    def __init__(self):
        pencere = Tk()
        dugme = Button(text="tamam")
        dugme.pack()

uygulama = Arayuz()

```

Burada tanımladığımız düğmenin bir iş yapmasını sağlayalım. Mesela düğmeye basılınca komut ekranında bir yazı çıksın. Önce şöyle deneyelim:

```

from Tkinter import *

class Arayuz:
    def __init__(self):
        pencere = Tk()
        dugme = Button(text="tamam", command=yaz)
        dugme.pack()

    def yaz():
        print "Hadi eyvallah!"

uygulama = Arayuz()

```

Tabii ki bu kodları çalıştırdığımızda şöyle bir hata mesajı alırız:

```

Traceback (most recent call last):
  File "xxxxxxxxxxxxxxxxxxxx", line 13, in <module>
    uygulama = Arayuz()
  File "xxxxxxxxxxxxxxxxxxxx", line 7, in __init__
    dugme = Button(text="tamam", command=yaz)
NameError: global name 'yaz' is not defined

```

Bunun sebebini bir önceki örnekte öğrenmiştik. Kodlarımızı şu şekilde yazmamız gerekiyor:

```

from Tkinter import *

class Arayuz:
    def __init__(self):
        pencere = Tk()
        dugme = Button(text="tamam", command=self.yaz)
        dugme.pack()

    def yaz(self):
        print "Hadi eyvallah!"

uygulama = Arayuz()

```

Gördüğünüz gibi, eğer programın farklı noktalarında kullanacağımız değişkenler veya fonksiyonlar varsa, bunları “self” öneki ile birlikte tanımlıyoruz. “def self.yaz” şeklinde bir fonksiyon tanımlama yöntemi olmadığına göre bu işlemi “def yaz(self)” şeklinde yapmamız gerekiyor. Bu son örnek aslında yine de tam anlamıyla kusursuz bir örnek değildir. Ama şimdilik elimizden bu kadarı geliyor. Daha çok bilginiz olduğunda bu kodları daha düzgün yazmayı da öğreneceğiz.

Bu iki örnek içinde, “self”lerle oynayarak olayın iç yüzünü kavramaya çalışın. Mesela yaz() fonksiyonundaki self parametresini silince ne tür bir hata mesajı alıyorsunuz, “com-

mand=self.yaz” içindeki “self” ifadesini silince ne tür bir hata mesajı alıyorsunuz? Bunları iyice inceleyip, “self”in nerede ne işe yaradığını kavramaya çalışın.

Bu noktada küçük bir sır verelim. Siz bu kelimeyi bütün sınıflı kodlamalarda bu şekilde görüyor olsanız da aslında illa ki “self” kelimesini kullanacaksınız diye bir kaide yoktur. Self yerine başka kelimeler de kullanabilirsiniz. Mesela yukarıdaki örneği şöyle de yazabilirsiniz:

```
from Tkinter import *

class Arayuz:
    def __init__(armut):
        pencere = Tk()
        dugme = Button(text="tamam",command=armut.yaz)
        dugme.pack()

    def yaz(armut):
        print "Hadi eyvallah!"

uygulama = Arayuz()
```

Ama siz böyle yapmayın. “self” kelimesinin kullanımı o kadar yaygınlaşmış ve yerleşmiştir ki, sizin bunu kendi kodlarınızda dahi olsa değiştirmeye kalkmanız pek hoş karşılanmayacaktır. Ayrıca sizin kodlarınızı okuyan başkaları, ne yapmaya çalıştığınızı anlamakta bir an da olsa tereddüt edecektir. Hatta birkaç yıl sonra dönüp siz dahi aynı kodlara baktığınızda, “Ben burada ne yapmaya çalışmışım,” diyebilirsiniz... O yüzden, “self” iyidir, “self” kullanın!...

Sizi “self” kullanmaya ikna ettiğimizi kabul edersek, artık yolumuza devam edebiliriz.

Hatırlarsanız yukarıda ufacık bir oyun çalışması yapmaya başlamıştık. Gelin isterseniz oyunumuzu biraz ayrıntılandıralım. Elimizde şimdilik şunlar vardı:

```
class Oyun:
    def __init__(self):
        self.enerji = 50
        self.para = 100
        self.fabrika = 4
        self.isci = 10

    def goster(self):
        print "enerji:", self.enerji
        print "para:", self.para
        print "fabrika:", self.fabrika
        print "işçi:", self.isci

macera = Oyun()
```

Buradaki kodlar yardımıyla bir oyuncu oluşturduk. Bu oyuncunun oyuna başladığında sahip olacağı enerji, para, fabrika ve işçi bilgilerini de girdik. Kodlarımız arasındaki goster() fonksiyonu yardımıyla da her an bu bilgileri görüntüleyebiliyoruz.

Şimdi isterseniz oyunumuza biraz hareket getirelim. Mesela kodlara yeni bir fonksiyon ekleyerek oyuncumuza yeni fabrikalar kurma olanağı tanıyalım:

```
class Oyun:
    def __init__(self):
        self.enerji = 50
        self.para = 100
        self.fabrika = 4
        self.isci = 10
```

```

def goster(self):
    print "enerji:", self.enerji
    print "para:", self.para
    print "fabrika:", self.fabrika
    print "işçi:", self.isci

def fabrikakur(self, miktar):
    if self.enerji > 3 and self.para > 10:
        self.fabrika = miktar + self.fabrika
        self.enerji = self.enerji - 3
        self.para = self.para - 10
        print miktar, "adet fabrika kurdunuz! Tebrikler!"

    else:
        print "Yeni fabrika kuramazsınız. Yeterli enerjiniz/paranız yok!"

```

```
macera = Oyun()
```

Burada fabrikakur() fonksiyonuyla ne yapmaya çalıştığımız aslında çok açık. Hemen bunun nasıl kullanılacağını görelim:

```
>>> macera.fabrikakur(5)
```

Bu komutu verdiğimizde, “5 adet fabrika kurdunuz! Tebrikler!” şeklinde bir kutlama mesajı gösterecektir bize programımız. . . Kodlarımız içindeki “def fabrikakur(self,miktar)” ifadesinde gördüğümüz “miktar” kelimesi, kodları çalıştırırken vereceğimiz parametreyi temsil ediyor. Yani burada “5” sayısını temsil ediyor. Eğer “macera.fabrikakur()” fonksiyonunu kullanırken herhangi bir sayı belirtmezseniz, hata alırsınız. Çünkü kodlarımızı tanımlarken fonksiyon içinde “miktar” adlı bir ifade kullanarak, kullanıcıdan fonksiyona bir parametre vermesini beklediğimizi belirttik. Dolayısıyla Python kullanıcıdan parantez içinde bir parametre girmesini bekleyecektir. Eğer fonksiyon parametresiz çalıştırılırsa da, Python’un beklentisi karşılanmadığı için, hata verecektir. Burada dikkat edeceğimiz nokta, kodlar içinde bir fonksiyon tanımlarken ilk parametrenin her zaman “self” olması gerektiğidir. Yani “def fabrikakur(miktar)” değil, “def fabrikakur(self,miktar)” dememiz gerekiyor.

Şimdi de şu komutu verelim:

```
>>> macera.goster()
```

```

enerji: 47
para: 90
fabrika: 9
işçi: 10

```

Gördüğümüz gibi oyuncumuz 5 adet fabrika kazanmış, ama bu işlem enerjisinde ve parasında bir miktar kayba neden olmuş (fabrika kurmayı bedava mı sandınız!).

Yazdığımız kodlara dikkatlice bakarsanız, oradaki if deyimi sayesinde oyuncunun enerjisi 3’ün altına, parası da 10’un altına düşerse şöyle bir mesaj verilecektir:

Yeni fabrika kuramazsınız. Yeterli enerjiniz/paranız yok!

Art arda fabrikalar kurarak bunu kendiniz de test edebilirsiniz.

## 20.8 Miras Alma (Inheritance)

Şimdiye kadar bir oyuncu oluşturduk ve bu oyuncuya oyuna başladığı anda sahip olacağı bazı özellikler verdik. Oluşturduğumuz oyuncu isterse oyun içinde fabrika da kurabiliyor. Ama böyle, “kendin çal, kendin oyna” tarzı bir durumun sıkıcı olacağı belli. O yüzden gelin oyuna biraz hareket katalım! Mesela oyunumuzda bir adet oyuncu dışında bir adet de düşman olsun. O halde hemen bir adet düşman oluşturalım:

```
class Dusman:
```

“Düşman”ımızın gövdesini oluşturduk. Şimdi sıra geldi onun kolunu bacağını oluşturmaya, ona bir kişilik kazandırmaya. . .

Hatırlarsanız, oyunun başında oluşturduğumuz oyuncunun bazı özellikleri vardı. (enerji, para, fabrika, işçi gibi. . .) İsterseniz düşmanımızın da buna benzer özellikleri olsun. Mesela düşmanımız da oyuncunun sahip olduğu özelliklerin aynıyla oyuna başlasın. Yani onun da:

enerjisi 50, parası 100 fabrika sayısı 4, işçi sayısı ise 10

olsun. Şimdi hatırlarsanız oyuncu için bunu şöyle yapmıştık:

```
class Oyun:
    def __init__(self):
        enerji = 50
        para = 100
        fabrika = 4
        isci = 10
```

Şimdi aynı şeyi “Dusman” sınıfı için de yapacağız. Peki bu özellikleri yeniden tek tek “düşman” için de yazacak mıyız? Tabii ki hayır. O halde nasıl yapacağız bunu? İşte burada imdadımıza Python sınıflarının “miras alma” özelliği yetişiyor. Yabancılar bu kavrama “inheritance” adını veriyorlar. Yani, nasıl Mısır’daki dedenizden size miras kaldığında dedenizin size bıraktığı mirasın nimetlerinden her yönüyle yararlanabiliyorsanız, bir sınıf başka bir sınıftan miras aldığı da aynı şekilde miras alan sınıf miras aldığı sınıfın özelliklerini kullanabiliyor. Az laf, çok iş. Hemen bir örnek yapalım. Yukarıda “Dusman” adlı sınıfımızı oluşturmuştuk:

```
class Dusman:
```

Dusman sınıfı henüz bu haliyle hiçbir şey miras almış değil. Hemen miras aldıralım. Bunun için sınıfımızı şöyle tanımlamamız gerekiyor:

```
class Dusman(Oyun):
```

Böylelikle daha en başta tanımladığımız “Oyun” adlı sınıfı, bu yeni oluşturduğumuz “Dusman” adlı sınıfa miras verdik. Dusman sınıfının durumunu Python’cada şöyle ifade edebiliriz:

“Dusman sınıfı Oyun sınıfını miras aldı” (Dusman inherits from Oyun)

Bu haliyle kodlarımız henüz eksik. Şimdilik şöyle bir şey yazıp sınıfımızı kitabına uyduralım:

```
class Dusman(Oyun):
    pass

dsman = Dusman()
```

Yukarıda pass ifadesini neden kullandığımızı biliyorsunuz. Sınıfı tanımladıktan sonra iki nokta üst üstenin ardından aşağıya bir kod bloğu yazmamız gerekiyor. Ama şu anda oraya yazacak bir kodumuz yok. O yüzden idareten oraya bir pass ifadesi yerleştirerek gerekli kod bloğunu geçiştirmiş oluyoruz. O kısmı boş bırakamayız. Yoksa sınıfımız kullanılamaz durumda olur.

Daha sonra oraya yazacağımız kod bloklarını hazırladıktan sonra oradaki pass ifadesini sileceğiz.

Şimdi bakalım bu sınıfla neler yapabiliyoruz?

Bu kodları, yazının başında anlattığımız şekilde çalıştıralım. Dediğimiz gibi, “Dusman” adlı sınıfımız daha önce tanımladığımız “Oyun” adlı sınıfı miras alıyor. Dolayısıyla “Dusman” adlı sınıf “Oyun” adlı sınıfın bütün özelliklerine sahip. Bunu hemen test edelim:

```
>>> dsman.goster()
```

```
enerji: 50  
para: 100  
fabrika: 4  
işçi: 10
```

Gördüğünüz gibi, Oyun sınıfının bir fonksiyonu olan `goster()`’i “Dusman” sınıfı içinden de çalıştırabildik. Üstelik Dusman içinde bu değişkenleri tekrar tanımlamak zorunda kalmadan... İstersek bu değişkenlere teker teker de ulaşabiliriz:

```
>>> dsman.enerji
```

```
50
```

```
>>> dsman.isci
```

```
10
```

Dusman sınıfı aynı zamanda Oyun sınıfının `fabrikakur()` adlı fonksiyonuna da erişebiliyor:

```
dsman.fabrikakur(4)
```

4 adet fabrika kurdunuz! Tebrikler!

Gördüğünüz gibi düşmanımız kendisine 4 adet fabrika kurdu!.. Düşmanımızın durumuna bakalım:

```
>>> dsman.goster()
```

```
enerji: 47  
para: 90  
fabrika: 8  
işçi: 10
```

Evet, düşmanımızın fabrika sayısı artmış, enerjisi ve parası azalmış. Bir de kendi durumumuzu kontrol edelim:

```
>>> macera.goster()
```

```
enerji: 50  
para: 100  
fabrika: 4  
işçi: 10
```

Dikkat ederseniz, Oyun ve Dusman sınıfları aynı değişkenleri kullandıkları halde birindeki değişiklik öbürünü etkilemiyor. Yani düşmanımızın yeni bir fabrika kurması bizim değerlerimizi değişikliğe uğratmıyor.

Şimdi şöyle bir şey yapalım:

Düşmanımızın, oyuncunun özelliklerine ek olarak bir de “ego” adlı bir niteliği olsun. Mesela düşmanımız bize her zarar verdiğinde egosu büyüsün!...

Önce şöyle deneyelim:

```
class Dusman(Oyun):
    def __init__(self):
        self.ego = 0
```

Bu kodları çalıştırdığımızda hata alırız. Çünkü burada yeni bir “\_\_init\_\_” fonksiyonu tanımladığımız için, bu yeni fonksiyon kendini Oyun sınıfının \_\_init\_\_ fonksiyonunun üzerine yazıyor. Dolayısıyla Oyun sınıfından miras aldığımız bütün nitelikleri kaybediyoruz. Bunu önlemek için şöyle bir şey yapmamız gerekir:

```
class Dusman(Oyun):
    def __init__(self):
        Oyun.__init__(self)
        self.ego = 0
```

Burada “Oyun.\_\_init\_\_(self)” ifadesiyle “Oyun” adlı sınıfın “\_\_init\_\_” fonksiyonu içinde yer alan bütün nitelikleri, “Dusman” adlı sınıfın \_\_init\_\_ fonksiyonu içine kopyalıyoruz. Böylece “self.ego” değişkenini tanımlarken, “enerji, para, vb.” niteliklerin kaybolmasını engelliyoruz. Aslında bu haliyle kodlarımız düzgün şekilde çalışır. Kodlarımızı çalıştırdığımızda biz ekranda göremesek de aslında “ego” adlı niteliğe sahiptir düşmanımız. Ekranda bunu göremememizin nedeni tabii ki kodlarımızda henüz bu niteliği ekrana yazdıracak bir “print” deyiminin yer almaması... İsterseniz bu özelliği daha önce de yaptığımız gibi ayrı bir fonksiyon ile halledelim:

```
class Dusman(Oyun):
    def __init__(self):
        Oyun.__init__(self)
        self.ego = 0

    def goster(self):
        Oyun.goster(self)
        print "ego:", self.ego
```

```
dsman = Dusman()
```

Tıpkı “\_\_init\_\_” fonksiyonunda olduğu gibi, burada da “Oyun.goster(self)” ifadesi yardımıyla “Oyun” sınıfının goster() fonksiyonu içindeki değişkenleri “Dusman” sınıfının goster() fonksiyonu içine kopyaladık. Böylece “ego” değişkenini yazdırırken, öteki değişkenlerin de yazdırılmasını sağladık.

Şimdi artık düşmanımızın bütün niteliklerini istediğimiz şekilde oluşturmuş olduk. Hemen deneyelim:

```
>>> dsman.goster()
```

```
enerji: 50
para: 100
fabrika: 4
işçi: 10
ego: 0
```

Gördüğümüz gibi düşmanımızın özellikleri arasında oyuncumuza ilave olarak bir de “ego” adlı bir nitelik var. Bunun başlangıç değerini “0” olarak ayarladık. Daha sonra yazacağımız fonksiyonda düşmanımız bize zarar verdikçe egosu büyüyecek... Şimdi gelin bu fonksiyonu yazalım:



```

class Dusman(Oyun):
    def __init__(self):
        Oyun.__init__(self)
        self.ego = 0

    def goster(self):
        Oyun.goster(self)
        print "ego:", self.ego

    def fabrikayik(self,miktar):
        macera.fabrika = macera.fabrika - miktar
        self.ego = self.ego + 2
        print "Tebrikler. Oyuncunun", miktar, "adet fabrikasını yıktınız!"
        print "Üstelik egonuz da tavana vurdu!"

dsman = Dusman()

```

Dikkat ederseniz, fabrikayik() fonksiyonu içindeki değişkeni “macera.fabrika” şeklinde yazdık. Yani bir önceki “Oyun” adlı sınıfın “örneğini” (instance) kullandık. “Dusman” sınıfının değil... Neden? Çok basit. Çünkü kendi fabrikalarımızı değil oyuncunun fabrikalarını yıkmak istiyoruz!.. Burada, şu kodu çalıştırarak oyuncumuzun kurduğu fabrikaları yıkabiliriz:

```
>>> dsman.fabrikayik(2)
```

Biz burada “2” adet fabrika yıkmayı tercih ettik..

Kodlarımızın en son halini topluca görelim isterseniz:

```

class Oyun:
    def __init__(self):
        self.enerji = 50
        self.para = 100
        self.fabrika = 4
        self.isci = 10

    def goster(self):
        print "enerji:", self.enerji
        print "para:", self.para
        print "fabrika:", self.fabrika
        print "işçi:", self.isci

    def fabrikakur(self,miktar):
        if self.enerji > 3 and self.para > 10:
            self.fabrika = miktar + self.fabrika
            self.enerji = self.enerji - 3
            self.para = self.para - 10
            print miktar, "adet fabrika kurdunuz! Tebrikler!"

        else:
            print "Yeni fabrika kuramazsınız. Yeterli enerjiniz/paranız yok!"

macera = Oyun()

class Dusman(Oyun):
    def __init__(self):
        Oyun.__init__(self)
        self.ego = 0

    def goster(self):

```

```

Oyun.goster(self)
print "ego:", self.ego

def fabrikayik(self,miktar):
    macera.fabrika = macera.fabrika - miktar
    self.ego = self.ego + 2
    print "Tebrikler. Oyuncunun", miktar, "adet fabrikasını yıktınız!"
    print "Üstelik egonuz da tavana vurdu!"

dsman = Dusman()

```

En son oluşturduğumuz fonksiyonda nerede “Oyun” sınıfını doğrudan adıyla kullandığımıza ve nerede bu sınıfın “örneğinden” (instance) yararlandığımıza dikkat edin. Dikkat ederkeniz, fonksiyon başlıklarını çağırırken doğrudan sınıfın kendi adını kullanıyoruz (mesela “Oyun.\_\_init\_\_(self)”). Bir fonksiyon içindeki değişkenleri çağırırken ise (mesela “macera.fabrika”), “örneği” (instance) kullanıyoruz. Eğer bir fonksiyon içindeki değişkenleri çağırırken de sınıf isminin kendisini kullanmak isterseniz, ifadeyi “Oyun().\_\_init\_\_(self)” şeklinde yazmanız gerekir. Ama siz böyle yapmayın... Yani değişkenleri çağırırken örneği kullanın.

Artık kodlarımız didiklenmek üzere sizi bekliyor. Burada yapılan şeyleri iyice anlayabilmek için kafanıza göre kodları değiştirin. Neyi nasıl değiştirdiğinizde ne gibi bir sonuç elde ettiğinizi dikkatli bir şekilde takip ederek, bu konunun zihninizde iyice yer etmesini sağlayın.

Aslında yukarıdaki kodları daha düzenli bir şekilde de yazmamız mümkün. Örneğin, “enerji, para, fabrika” gibi nitelikleri ayrı bir sınıf halinde düzenleyip, öteki sınıfların doğrudan bu sınıftan miras almasını sağlayabiliriz. Böylece sınıfımız daha derli toplu bir görünüm kazanmış olur. Aşağıdaki kodlar içinde, isimlendirmeleri de biraz değiştirerek standartlaştırdığımıza dikkat edin:

```

class Oyun:
    def __init__(self):
        self.enerji = 50
        self.para = 100
        self.fabrika = 4
        self.isci = 10

    def goster(self):
        print "enerji:", self.enerji
        print "para:", self.para
        print "fabrika:", self.fabrika
        print "işçi:", self.isci

oyun = Oyun()

class Oyuncu(Oyun):
    def __init__(self):
        Oyun.__init__(self)

    def fabrikakur(self,miktar):
        if self.enerji > 3 and self.para > 10:
            self.fabrika = miktar + self.fabrika
            self.enerji = self.enerji - 3
            self.para = self.para - 10
            print miktar, "adet fabrika kurdunuz! Tebrikler!"

        else:
            print "Yeni fabrika kuramazsınız. Yeterli enerjiniz/paranız yok!"

```

```

oyuncu = Oyuncu()

class Dusman(Oyun):
    def __init__(self):
        Oyun.__init__(self)
        self.ego = 0

    def goster(self):
        Oyun.goster(self)
        print "ego:", self.ego

    def fabrikayik(self,miktar):
        oyuncu.fabrika = oyuncu.fabrika - miktar
        self.ego = self.ego + 2
        print "Tebrikler. Oyuncunun", miktar, "adet fabrikasını yıktınız!"
        print "Üstelik egonuz da tavana vurdu!"

dusman = Dusman()

```

Bu kodlar hakkında son bir noktaya daha değinelim. Hatırlarsanız oyuna başlarken oluşturulan niteliklerde değişiklik yapabiliyorduk. Mesela yukarıda “Dusman” sınıfı için “ego” adlı yeni bir nitelik tanımlamıştık. Bu nitelik sadece “Dusman” tarafından kullanılabiliyordu, Oyuncu tarafından değil. Aynı şekilde, yeni bir nitelik belirlemek yerine, istersek varolan bir niteliği iptal de edebiliriz. Diyelim ki Oyuncu’nun oyuna başlarken “fabrika”ları olsun istiyoruz, ama Dusman’ın oyun başlangıcında fabrikası olsun istemiyoruz. Bunu şöyle yapabiliriz:

```

class Dusman(Oyun):
    def __init__(self):
        Oyun.__init__(self)
        del self.fabrika
        self.ego = 0

```

Gördüğünüz gibi “Dusman” sınıfı için “\_\_init\_\_” fonksiyonunu tanımlarken “fabrika” niteliğini “del” komutuyla siliyoruz. Bu silme işlemi sadece “Dusman” sınıfı için geçerli oluyor. Bu işlem öteki sınıfları etkilemiyor. Bunu şöyle de ifade edebiliriz;

“del komutu yardımıyla fabrika adlı değişkene Dusman adlı bölgeden erişilmesini engelliyoruz.”

Dolayısıyla bu değişiklik sadece o “bölgeyi” etkiliyor. Öteki sınıflar ve daha sonra oluşturulacak yeni sınıflar bu işlemten etkilenmez. Yani aslında “del” komutuyla herhangi bir şeyi sildiğimiz yok! Sadece “erişimi engelliyoruz”.

Küçük bir not: Burada “bölge” olarak bahsettiğimiz şey aslında Python’cada “isim alanı” (namespace) olarak adlandırılıyor.

Şimdi bir örnek de Tkinter ile yapalım. Yukarıda verdiğimiz örneği hatırlıyorsunuz:

```

from Tkinter import *

class Arayuz:
    def __init__(self):
        pencere = Tk()
        dugme = Button(text="tamam",command=self.yaz)
        dugme.pack()

    def yaz(self):
        print "Hadi eyvallah!"

```

```
uygulama = Arayuz()
```

Bu örnek gayet düzgün çalışsa da bu sınıfı daha düzgün ve düzenli bir hale getirmemiz mümkün:

```
#!/usr/bin/env python
#-*-coding:utf-8-*-

from Tkinter import *

class Arayuz(Frame):
    def __init__(self):
        Frame.__init__(self)
        self.pack()
        self.pencerearaclari()

    def pencerearaclari(self):
        self.dugme = Button(self, text="tamam", command=self.yaz)
        self.dugme.pack()

    def yaz(self):
        print "Hadi eyvallah!"

uygulama = Arayuz()
uygulama.mainloop()
```

Burada dikkat ederseniz, Tkinter’in “Frame” adlı sınıfını miras aldık. Buradan anlayacağımız gibi, “miras alma” (inheritance) özelliğini kullanmak için miras alacağımız sınıfın o anda kullandığımız modül içinde olması şart değil. Burada olduğu gibi, başka modüllerin içindeki sınıfları da miras alabiliyoruz. Yukarıdaki kodları dikkatlice inceleyin. Başta biraz karışık gibi görünse de aslında daha önce verdiğimiz basit örneklerden hiç bir farkı yoktur.

## 20.9 Eski ve Yeni Sınıflar

Şimdiye kadar verdiğimiz sınıf örneklerinde önemli bir konudan hiç bahsetmedik. Python’da iki tip sınıf vardır: Eski tip sınıflar ve yeni tip sınıflar. Ancak korkmanızı gerektirecek kadar fark yoktur bu iki sınıf tipi arasında. Ayrıca hangi sınıf tipini kullanırsanız kullanın sorun yaşamazsınız. Ama tabii ki kendimizi yeni tipe alıştırmakta fayda var, çünkü muhtemelen Python’un sonraki sürümlerinden birinde (büyük ihtimalle Python 3.0’da) eski tip sınıflar kullanımdan kaldırılacaktır.

Eski tip sınıflar ile yeni tip sınıflar arasındaki en büyük fark şudur:

Eski tip sınıflar şöyle tanımlanır:

```
class Deneme:
```

Yeni tip sınıflar ise şöyle tanımlanır:

```
class Deneme(object)
```

Gördüğümüz gibi, eski tip sınıflarda başka bir sınıfı miras alma zorunluluğu yoktur. O yüzden sınıfları istersek parantezsiz olarak tanımlayabiliyoruz. Yeni tip sınıflarda ise her sınıf mutlaka başka bir sınıfı miras almalıdır. Eğer kodlarınız içinde gerçekten miras almanız gereken başka bir sınıf yoksa, öntanımlı olarak “object” adlı sınıfı miras almanız gerekiyor. Dolayısıyla politikamız şu olacak:

“Ya bir sınıfı miras al, ya da miras alman gereken herhangi bir sınıf yoksa, “object” adlı sınıfı miras al...”

Dediğimiz gibi, eski ve yeni sınıflar arasındaki en temel fark budur.

Aslında daha en başta hiç eski tip sınıfları anlatmadan doğrudan yeni tip sınıfları anlatmakla işe başlayabilirdik. Ama bu pek doğru bir yöntem olmazdı. Çünkü her ne kadar eski tip sınıflar sonraki bir Python sürümünde tedavülden kaldırılacaksa da, etrafta eski sınıflarla yazılmış bolca kod göreceksiniz. Dolayısıyla sadece yeni tip sınıfları öğrenmek mevcut tabloyu eksik algılamak olacaktır...

Yukarıda hatırlarsanız “pass” ifadesini kullanmıştık. Sınıfların yapısı gereği bir kod bloğu belirtmemiz gerektiğinde, ama o anda yazacak bir şeyimiz olmadığında sırf bir “yer tutucu” vazifesi görsün diye o pass ifadesini kullanmıştık. Yine bu pass ifadesini kullanarak başka bir şey daha yapabiliriz. Şu örneğe bakalım:

```
class BosSinif(object):  
    pass
```

Böylece içi boş da olsa kurallara uygun bir sınıf tanımlamış olduk. Ayrıca dikkat ederseniz, sınıfımızı tanımlarken “yeni sınıf” yapısını kullandık. Özel olarak miras alacağımız bir sınıf olmadığı için doğrudan “object” adlı sınıfı miras aldık. Yine dikkat ederseniz sınıfımız için bir “örnek” (instance) de belirtmedik. Hem sınıfın içeriğini doldurma işini, hem de örnek belirleme işini komut satırından halledeceğiz. Önce sınıfımızı örnekliyoruz:

```
sinifimiz = BosSinif()
```

Gördüğümüz gibi BosSinif() şeklinde, parametresiz olarak örnekliyoruz sınıfımızı. Zaten parantez içinde bir parametre belirtirseniz hata mesajı alırsınız...

Şimdi boş olan sınıfımıza “nitelikler” ekliyoruz:

```
>>> sinifimiz.sayi1 = 45  
>>> sinifimiz.sayi2 = 55  
>>> sinifimiz.sonuc = sinifimiz.sayi1 * sinifimiz.sayi2  
>>> sinifimiz.sonuc
```

```
2475
```

İstersek sınıfımızın son halini, Python sınıflarının \_\_dict\_\_ metodu yardımıyla görebiliriz:

```
sinifimiz.__dict__  
  
{'sayi2': 55, 'sayi1': 45, 'sonuc': 2475}
```

Gördüğümüz gibi sınıfın içeriği aslında bir sözlükten ibaret... Dolayısıyla sözlüklere ait şu işlemler sınıfımız için de geçerlidir:

```
sinifimiz.__dict__.keys()  
  
['sayi2', 'sayi1', 'sonuc']  
  
sinifimiz.__dict__.values()  
  
[55, 45, 2475]
```

Buradan öğrendiğimiz başka bir şey de, sınıfların içeriğinin dinamik olarak değiştirilebileceğidir. Yani bir sınıfı her şeyiyle tanımladıktan sonra, istersek o sınıfın niteliklerini etkileşimli olarak değiştirebiliyoruz.

## 20.10 Sonuç

Böylece “Nesne Tabanlı Programlama” konusunun sonuna gelmiş oluyoruz. Aslında daha doğru bir ifadeyle, Nesne Tabanlı Programlama’ya hızlı bir giriş yapmış oluyoruz. Çünkü NTP şu birkaç sayfada anlatılanlardan ibaret değildir. Bu yazımızda bizim yapmaya çalıştığımız şey, okuyucuya NTP hakkında bir fikir vermektir. Eğer okuyucu bu yazı sayesinde NTP hakkında hiç değilse birazcık fikir sahibi olmuşsa kendimizi başarılı sayacağız. Bu yazıdaki amaç NTP gibi çetrefilli bir konuyu okuyucunun gözünde bir nebze de olsa sevimli kılabilmek, konuyu kolay hazmedilir bir hale getirmektir. Okuyucu bu yazıdan sonra NTP’ye ilişkin başka kaynakları daha bir kendine güvenle inceleme imkanına kavuşacak ve okuduğunu daha kolay anlayacaktır.

---

# Sqlite ile Veritabanı Programlama

---

## 21.1 Giriş

Bu bölümde, Python'daki ileri düzey konulardan biri olan veritabanı programlamayı (*database programming*) inceleyeceğiz. Peki nedir bu “veritabanı” denen şey?

Esasında veritabanı, hiçbirimizin yabancı olduğu bir kavram değil. Biz bu kelimeyi, teknik anlamının dışında, günlük hayatta da sıkça kullanıyoruz. Veritabanı, herkesin bildiği ve kullanıldığı anlamıyla, içinde veri barındıran bir “şey”dir. Günlük kullanımda, hakikaten, içinde veri barındıran her şeye veritabanı dendiğini duyarsınız.

Veritabanı kelimesinin günlük kullanımdaki anlamı dışında bir de teknik anlamı vardır. Bizi esas ilgilendiren de zaten terimin teknik anlamıdır. Mesela Vikipedi’de veritabanı şöyle tanımlanıyor:

*Bilgisayar terminolojisinde, sistematik erişim imkânı olan, yönetilebilir, güncellenebilir, taşınabilir, birbirleri arasında tanımlı ilişkiler bulunabilen bilgiler kümesidir. Bir başka tanımı da, bir bilgisayarda sistematik şekilde saklanmış, programlarca işlenebilecek veri yığınıdır.*

Yukarıdaki tanım, veritabanının ne demek olduğunu gayet iyi ifade ediyor. Ama esasında bizim veritabanı tanımı üzerinde fazlaca durmamıza gerek yok. Biz her zaman olduğu gibi işin teknik boyutuyla değil, taktik boyutuyla ilgilenmeyi tercih edeceğiz. O halde yavaş yavaş işe koyulmaya başlayalım.

Python’la veritabanı programlama işlemleri için pek çok alternatifimiz var. Python’la hangi veritabanı sistemlerini kullanabileceğinizi görmek için <http://wiki.python.org/moin/DatabaseInterfaces> adresindeki listeyi inceleyebilirsiniz. Biz bunlar içinde, sadeliği, basitliği ve kullanım kolaylığı nedeniyle **Sqlite** adlı veritabanı yönetim sistemini kullanacağız.

## 21.2 Neden Sqlite?

Dediğimiz gibi, Python’da veritabanı işlemleri için kullanabileceğiniz pek çok alternatif bulunur. Ama biz bütün bu alternatifler içinde Sqlite’ı tercih edeceğiz. Peki neden Sqlite?

Sqlite’in öteki sistemlere göre pek çok avantajı bulunur. Gelin isterseniz Sqlite’in bazı avantajlarına şöyle bir göz gezdirelim:

- Her şeyden önce Sqlite Python’un 2.5 sürümlerinden bu yana bu dilin bir parçasıdır. Dolayısıyla eğer kullandığınız Python sürümü 2.5 veya üstü ise Sqlite’ı Python’daki herhangi bir modül gibi içe aktarabilir ve kullanmaya başlayabilirsiniz.
- Sqlite herhangi bir yazılım veya sunucu kurulumu gerektirmez. Bu sayede, bu modülü kullanabilmek için sunucu konfigürasyonu yapmaya da gerek yoktur. Bazı veritabanlarını kullanabilmek için arka planda bir veritabanı sunucusu çalıştırıyor olmanız gerekir. Sqlite’ta ise böyle bir şey yapmazsınız.
- Sqlite, öteki pek çok veritabanı alternatifine göre basittir. Bu yüzden Sqlite’ı çok kısa bir sürede kavrayıp kullanmaya başlayabilirsiniz.
- Sqlite özgür bir yazılımdır. Bu yazılımın baştan aşağı bütün kodları kamuya açıktır. Dolayısıyla Sqlite kodlarının her zerresini istediğiniz gibi kullanabilir, değişikliğe uğratabilir, satabilir ve ticari veya ticari olmayan uygulamalarınızda gönül rahatlığıyla kullanabilirsiniz.
- Sqlite’in “sade” ve “basit” olması sizi yanıltmasın. Bu özelliklerine bakarak, Sqlite’in yeteneksiz bir veritabanı sistemi olduğunu düşünmeyin. Bugün Sqlite’ı aktif olarak kullanan pek çok büyük ve tanınmış şirket bulunur. Mesela, Adobe, Apple, Mozilla/Firefox, Google, Symbian ve Sun bu şirketlerden bazılarıdır. Hatta GNOME masaüstü ortamının sevilen müzik ve video çalarlarından Banshee’de de veritabanı olarak Sqlite kullanıldığını söyleyelim.

Yukarıdaki sebeplerden ötürü, veritabanı konusunu Sqlite üzerinden anlatacağız. O halde hemen yola koyulalım.

## 21.3 Sqlite’in Yapısı

Bu bölümün en başında verdiğimiz veritabanı tanımından da anlaşılacağı gibi, veritabanları, verileri sonradan kullanılmak üzere içinde tutan bir sistemdir. Bütün **ilişkisel veritabanlarında** olduğu gibi, Sqlite da bu verileri bir tablo yapısı içinde tutar. Yani aslında bir Sqlite veritabanı içindeki veriler şöyle bir yapıya sahiptir:

Sütun 1	Sütun 2	Sütun 3	Sütun 4	Sütun 5
Değer 1/1	Değer 2/1	Değer 3/1	Değer 4/1	Değer 5/1
Değer 1/2	Değer 2/2	Değer 3/2	Değer 4/2	Değer 5/2
Değer 1/3	Değer 2/3	Değer 3/3	Değer 4/3	Değer 5/3
Değer 1/4	Değer 2/4	Değer 3/4	Değer 4/4	Değer 5/4

Sqlite içinde oluşturulan yukarıdakine benzer her tablonun bir de ismi vardır. Daha doğrusu, Sqlite ile bir tablo oluştururken, bu tabloya bir de ad vermemiz gerekir. Mesela yukarıdaki tabloya “Değerler” adını verdiğimiz varsayabilirsiniz...

Sqlite ile çalışırken veriler üzerinde yapacağımız işlemleri, yukarıdaki tablonun adını ve bu tablodaki sütunları kullanarak gerçekleştireceğiz. Bu yüzden Sqlite’in yapısını anlamak büyük önem taşır. Gördüğünüz gibi, bu veritabanı sisteminin yapısını anlamak da öyle zor bir iş değildir.



## 21.4 Veritabanıyla Bağlantı Kurmak

Bu bölümde `sqlite` modülünü kullanarak bir veritabanına nasıl bağlanacağımızı, elimizde herhangi bir veritabanı yoksa veritabanını nasıl oluşturacağımızı inceleyeceğiz.

Dikkat ederseniz burada bir `sqlite` modülünden söz ettik. Dolayısıyla, tahmin edebileceğiniz gibi, bu modülü kullanabilmek için öncelikle modülü içe aktarmamız gerekiyor. Bu bölümün başında da söylediğimiz gibi, `Sqlite`, Python'un 2.5 sürümünden bu yana Python'un bir parçasıdır:

```
>>> import sqlite3
```

Python'da `Sqlite` veritabanı sistemine ait modül "`sqlite3`" adını taşır. Bu yüzden, bu modülü içe aktarmak için `import sqlite3` ifadesini kullanmamız gerekiyor. Eğer bu isim size çok uzun geliyorsa veya modül adında sayıların ve harflerin birlikte bulunması nedeniyle hem sayı hem de harf girmeyi bir angarya olarak görüyorsanız elbette `sqlite3` modülünü farklı bir adla da içe aktarabileceğinizi biliyorsunuz. Mesela:

```
>>> import sqlite3 as sql
```

Veya:

```
>>> import sqlite3 as lite
```

Böylece `sqlite3` modülünü "`sql`" veya "`lite`" adıyla içe aktarmış olduk. Ancak ben konuyu anlatırken, okur açısından kafa karışıklığına sebep olmamak için, modülü `import sqlite3` şeklinde içe aktarmışız gibi davranacağım.

Gelelim bu modül yardımıyla nasıl veritabanı oluşturulacağına... Bunun için `sqlite3` modülünün `connect()` adlı fonksiyonundan yararlanacağız. Bu fonksiyonu şu şekilde kullanıyoruz:

```
>>> sqlite3.connect("veritabanı_adi")
```

`connect()` metoduna verdiğimiz "`veritabanı_adi`" adlı argüman kullanacağımız veritabanının adıdır. Eğer belirtilen isimde bir veritabanı sistemde bulunmuyorsa o adla yeni bir veritabanı oluşturulacaktır. Mesela:

```
>>> vt = sqlite3.connect("deneme.db")
```

Eğer bu komutu verdiğiniz dizin içinde `deneme.db` adlı bir veritabanı yoksa, bu ada sahip bir veritabanı oluşturulur. Eğer zaten bu adla bir veritabanı dosyanız varsa, `sqlite3` bu veritabanına bağlanacaktır.

Elbette isterseniz `connect()` metoduna argüman olarak tam dosya yolu da verebilirsiniz:

```
>>> import sqlite3 as sq
```

```
>>> v = sq.connect("/home/istihza/test.db") #GNU/Linux
```

```
>>> v = sq.connect("c:/documents and settings/fozgul/desktop/test.db") #Windows
```

Bu komut yardımıyla sabit disk üzerinde bir dosya oluşturmuş veya varolan bir dosyaya bağlanmış oluyoruz. Ancak isterseniz `sqlite3` ile geçici bir veritabanı da oluşturabilirsiniz:

```
>>> vt = sqlite3.connect(":memory:")
```

Oluşturduğunuz bu geçici veritabanı sabit disk üzerinde değil RAM (bellek) üzerinde çalışır. Veritabanını kapattığınız anda da bu geçici veritabanı silinir. Eğer arzu ederseniz, RAM üzerinde

değil, disk üzerinde de geçici veritabanları oluşturabilirsiniz. Bunun için de şöyle bir komut kullanıyoruz:

```
>>> vt = sqlite3.connect("")
```

Gördüğünüz gibi, disk üzerinde geçici bir veritabanı oluşturmak için boş bir karakter dizisi kullandık. Tıpkı `:memory:` kullanımında olduğu gibi, boş karakter dizisiyle oluşturulan geçici veritabanları da veritabanı bağlantısının kesilmesiyle birlikte ortadan kalkacaktır.

Geçici veritabanı oluşturmak, özellikle çeşitli testler veya denemeler yaptığınız durumlarda işinize yarar. Sonradan nasıl olsa sileceğiniz, sırf test amaçlı tuttuğunuz bir veritabanını disk üzerinde oluşturmak yerine RAM üzerinde oluşturmayı tercih edebilirsiniz. Ayrıca, geçici veritabanları sayesinde, yazdığınız bir kodu test ederken bir hatayla karşılaşsanız sorunun veritabanı içinde varolan verilerden değil, yazdığınız koddan kaynaklandığından da emin olabilirsiniz. Çünkü, dediğimiz gibi, programın her yeniden çalışışında veritabanı baştan oluşturulacaktır. Her şeyden öte, bellek üzerinde yapılan işlemler sabit disk üzerinde yapılan işlemlere göre daha hızlıdır...

Dikkatinizi çekmek istediğim bir nokta da şudur: Gördüğünüz gibi Sqlite, veritabanını o anda içinde bulunduğunuz dizin içinde oluşturuyor. Mesela MySQL kullanıyor olsaydınız, oluşturulan veritabanlarının önceden tanımlanmış bir dizin içine atıldığını görecektiniz. Örneğin GNU/Linux sistemlerinde, MySQL veritabanları `/var/lib/mysql` gibi bir dizinin içinde tutulur...

Böylece Sqlite ile nasıl veritabanı bağlantısı kuracağımızı ve nasıl yeni bir veritabanı oluşturacağımızı öğrenmiş olduk. Veritabanı oluşturduktan sonra, veritabanı üzerinde işlem yapabilmek için ilk adım olarak bir imleç oluşturmamız gerekir. İmleç oluşturmak için `cursor()` metodundan yararlanacağız:

```
>>> im = vt.cursor()
```

İmleci oluşturduktan sonra artık önümüz iyice açılıyor. Böylece, yukarıda oluşturduğumuz imlec nesnesinin `execute()` metodunu kullanarak SQL komutlarını çalıştırabileceğiz.

## 21.5 Veri Girişi

Önceki bölümün sonunda söylediğimiz gibi, bir imleç nesnesi oluşturduktan sonra bunun `execute()` metodunu kullanarak SQL komutlarını işletebiliyoruz.

Dilerseniz şimdi basit bir örnek yaparak neyin ne olduğunu anlamaya çalışalım:

```
>>> im.execute("CREATE TABLE adres_defteri (isim, soyisim)")
```

Hatırlarsanız, Sqlite veritabanı sisteminin tablo benzeri bir yapıya sahip olduğunu ve bu sistemdeki her tablonun da bir isminin bulunduğunu söylemiştik. İşte burada yaptığımız şey, “adres\_defteri” adlı bir tablo oluşturup, bu tabloya “isim” ve “soyisim” adlı iki sütun eklemekten ibarettir. Yani aslında şöyle bir şey oluşturmuş oluyoruz:

isim	soyisim

Ayrıca oluşturduğumuz bu tablonun adının da “adres\_defteri” olduğunu unutmuyoruz...

Bu işlemleri nasıl yaptığımıza dikkat edin. Burada `CREATE TABLE adres_defteri (isim, soyisim)` tek bir karakter dizisidir. Bu karakter dizisindeki `CREATE TABLE` kısmı bir SQL komutu olup, bu komut bir tablo oluşturulmasını sağlar.

Burada `CREATE TABLE` ifadesini büyük harflerle yazdık. Ancak bu ifadeyi siz isterseniz küçük harflerle de yazabilirsiniz. Benim burada büyük harf kullanmaktaki amacım SQL komutlarının,

“adres\_defteri”, “isim” ve “soyisim” gibi değişkenlerden görsel olarak ayırt edilebilmesini sağlamak. Yani CREATE TABLE ifadesinin mesela “adres\_defteri” değişkeninden kolayca ayırt edilebilmesini istediğim için burada CREATE TABLE ifadesini büyük harflerle yazdım.

Karakter dizisinin devamında (isim, soyisim) ifadesini görüyoruz. Tahmin edebileceğiniz gibi, bunlar tablodaki sütun başlıklarının adını gösteriyor. Buna göre, oluşturduğumuz tabloda “isim” ve “soyisim” adlı iki farklı sütun başlığı olacak.

Yukarıda execute() metodunu kullanarak, veritabanı içinde adres\_defteri adlı bir tablo oluşturduk. Ardından da bu tablo içine isim ve soyisim adlı iki sütun başlığı yerleştirdik. Bu işlemin ne kadar kolay olduğunu görüyorsunuz. Şimdi yine buna benzer bir komut yardımıyla, yukarıda oluşturduğumuz sütun başlıklarının altını dolduracağız:

```
>>> im.execute("INSERT INTO adres_defteri VALUES ('Fırat', 'Özgül')")
```

Burada CREATE TABLE komutundan sonra INSERT INTO adlı yeni bir SQL komutu daha öğreniyoruz. CREATE TABLE ifadesi Türkçe’de “TABLO OLUŞTUR” anlamına geliyor. INSERT INTO ise “... İÇİNE YERLEŞTİR” anlamına gelir. Yukarıdaki karakter dizisi içinde görünen VALUES ise “DEĞERLER” demektir. Yani aslında yukarıdaki karakter dizisi şu anlama gelir: “adres\_defteri İÇİNE ‘Fırat’ ve ‘Özgül’ DEĞERLERİNİ YERLEŞTİR. Yani şöyle bir tablo oluştur”:

isim	soyisim
Fırat	Özgül

Buraya kadar gayet güzel gidiyoruz. İsterseniz şimdi derin bir nefes alıp, şu ana kadar yaptığımız şeyleri bir gözden geçirelim:

- Öncelikle sqlite3 modülünü içe aktardık. Bu modülün nimetlerinden yararlanabilmek için bunu yapmamız gerekiyordu. “sqlite3” kelimesini her defasında yazmak bize angarya gibi gelebileceği için bu modülü farklı bir adla içe aktarmayı tercih edebiliriz. Mesela import sqlite3 as sql veya import sqlite3 as lite gibi...
- sqlite3 modülünü içe aktardıktan sonra bir veritabanına bağlanmamız veya elimizde bir veritabanı yoksa yeni bir veritabanı oluşturmamız gerekiyor. Bunun için connect() adlı bir fonksiyondan yararlanıyoruz. Bu fonksiyonu, sqlite3.connect("veritabanı\_adı") şeklinde kullanıyoruz. Eğer içinde bulunduğumuz dizinde, “veritabanı\_adı” adlı bir veritabanı varsa Sqlite bu veritabanına bağlanır. Eğer bu adda bir veritabanı yoksa, çalışma dizini altında bu ada sahip yeni bir veritabanı oluşturulur. Özellikle deneme amaçlı işlemler yapmamız gerektiğinde, sabit disk üzerinde bir veritabanı oluşturmak yerine RAM üstünde geçici bir veritabanı ile çalışmayı da tercih edebiliriz. Bunun için yukarıdaki komutu şöyle yazıyoruz: sqlite3.connect(":memory:"). Bu komutla RAM üzerinde oluşturduğumuz veritabanı, bağlantı kesildiği anda ortadan kalkacaktır.
- Veritabanımızı oluşturduktan veya varolan bir veritabanına bağlandıktan sonra yapmamız gereken şey bir imleç oluşturmak olacaktır. Daha sonra bu imlece ait metotlardan yararlanarak önemli işler yapabileceğiz... Sqlite’ta bir imleç oluşturabilmek için db.cursor() gibi bir komut kullanıyoruz. Tabii ben burada oluşturduğunuz veritabanına “db” adını verdiğiniz varsayıyorum...
- İmlecimizi de oluşturduktan sonra önümüz iyice açılmış oldu. Şimdi dir(im) gibi bir komut kullanarak imlecin metotlarının ne olduğunu inceleyebilirsiniz. Tabii ben burada imlece “im” adını verdiğinizi varsaydım... Gördüğünüz gibi, listede execute() adlı bir metot da var. Artık imlecin bu execute() metodunu kullanarak SQL komutlarını işletebiliriz.
- Yukarıda iki adet SQL komutu öğrendik. Bunlardan ilki CREATE TABLE. Bu komut veritabanı içinde bir tablo oluşturmamızı sağlıyor. İkincisi ise INSERT INTO ... VALUES .... Bu komut da, oluşturduğumuz tabloya içerik eklememizi sağlıyor. Bunları şu şekilde kullandığınızı hatırlıyorsunuz:

```
>>> im.execute("CREATE TABLE adres_defteri (isim, soyisim)")
>>> im.execute("INSERT INTO adres_defteri VALUES ('Fırat', 'Özgül')")
```

Burada bir şey dikkatinizi çekmiş olmalı. SQL komutlarını yazmaya başlarken çift tırnakla başladık. Dolayısıyla karakter dizisini yazarken iç taraftaki *Fırat* ve *Özgül* değerlerini yazmak için tek tırnak kullanmamız gerekti. Karakter dizileri içindeki manevra alanınızı genişletmek için, SQL komutlarını üç tırnak içinde yazmayı da tercih edebilirsiniz. Böylece karakter dizisi içindeki tek ve çift tırnakları daha rahat bir şekilde kullanabilirsiniz. Yani:

```
>>> im.execute("""CREATE TABLE adres_defteri (isim, soyisim)""")
>>> im.execute("""INSERT INTO adres_defteri VALUES ("Fırat", "Özgül")""")
```

Ayrıca üç tırnak kullanmanız sayesinde, uzun satırları gerektiğinde bölerek çok daha okunaklı kodlar da yazabileceğinizi biliyorsunuz.

## 21.6 Veri İşleme - commit() Metodu

Bir önceki bölümde bir Sqlite veritabanına nasıl veri gireceğimizi öğrendik. Ama aslında iş sadece veri girmeyele bitmiyor. Verileri veritabanına “işleyebilmek” için bir adım daha atmamız gerekiyor. Mesela şu örneğe bir bakalım:

```
# -*- coding: utf-8 -*-

import sqlite3

vt = sqlite3.connect(":memory:")

im = vt.cursor()
im.execute("""CREATE TABLE personel (isim, soyisim, şehir, eposta)""")

im.execute("""INSERT INTO personel VALUES
("Orçun", "Kunek", "Adana", "okunek@gmail.com")""")
```

Burada öncelikle RAM üzerinde geçici bir veritabanı oluşturduk. Zaten gerçek bir uygulama yazmadığımız, henüz test aşamasında olduğumuz için en iyi yaklaşım geçici bir veritabanı oluşturmak olacaktır.

Ardından, `vt.cursor()` komutuyla imlecimizi de oluşturduktan sonra, SQL komutlarımızı çalıştırıyoruz. Önce isim, soyisim, şehir ve eposta adlı sütunlardan oluşan, “personel” adlı bir tablo oluşturduk. Daha sonra “personel” tablosunun içine “Orçun”, “Kunek”, “Adana” ve “okunek@gmail.com” değerlerini yerleştirdik.

Ancak her ne kadar veritabanına veri işlemiş gibi görünsek de aslında henüz işlenmiş bir şey yoktur. Biz henüz sadece verileri girdik. Ama verileri veritabanına işlemedik. Bu girdiğimiz verileri veritabanına işleyebilmek için `commit()` adlı bir metottan yararlanacağız:

```
>>> vt.commit()
```

Gördüğünüz gibi, `commit()` imlecin değil, bağlantı nesnesinin (yani burada `vt` değişkeninin) bir metodudur. Şimdi bu satırı da betiğimize ekleyelim:

```
# -*- coding: utf-8 -*-

import sqlite3
```

```
vt = sqlite3.connect(":memory:")

im = vt.cursor()
im.execute("""CREATE TABLE personel (isim, soyisim, şehir, eposta)""")

im.execute("""INSERT INTO personel VALUES
("Orçun", "Kunek", "Adana", "okunek@gmail.com")""")

vt.commit()
```

Bu son satırı da ekledikten sonra Sqlite veritabanı içinde şöyle bir tablo oluşturmuş olduk:

isim	soyisim	şehir	eposta
Orçun	Kunek	Adana	okunek@gmail.com

## 21.7 Veritabanından Veri Almak

Yukarıda, bir veritabanına nasıl veri gireceğimizi ve işleyeceğimizi gördük. İşin asıl önemli kısmı, bu verileri daha sonra veritabanından geri alabilmektir. Şimdi bu işlemi nasıl yapacağımıza bakacağız.

Veritabanından herhangi bir veri alabilmek için `SELECT...FROM...` adlı bir SQL komutundan yararlanmamız gerekiyor.

Dilerseniz önce bir tablo oluşturalım:

```
# -*- coding: utf-8 -*-

import sqlite3

vt = sqlite3.connect(":memory:")

im = vt.cursor()

im.execute("""CREATE TABLE faturalar
(fatura, miktar, ilk_odeme_tarihi, son_odeme_tarihi)""")
```

Şimdi bu tabloya bazı veriler ekleyelim:

```
im.execute("""INSERT INTO faturalar VALUES
("Elektrik", 45, "23 Ocak 2010", "30 Ocak 2010")""")
```

Verileri veritabanına işleyelim:

```
vt.commit()
```

Yukarıdaki kodlar bize şöyle bir tablo verdi:

fatura	miktar	ilk_odeme_tarihi	son_odeme_tarihi
Elektrik	45	23 Ocak 2010	30 Ocak 2010

Buraya kadar olan kısmı zaten biliyoruz. Bilmediğimiz ise bu veritabanından nasıl veri alacağımız. Onu da şöyle yapıyoruz:

```
im.execute("""SELECT * FROM faturalar""")
```

Burada özel bir SQL komutu olan `SELECT...FROM...`'dan faydalandık. Burada joker karakterlerden biri olan `"*"` işaretini kullandığımıza dikkat edin. `SELECT * FROM faturalar` ifadesi şu anlama gelir: *"faturalar adlı tablodaki bütün öğeleri seç!"*

Burada *"SELECT"* kelimesi *"SEÇMEK"* demektir. *"FROM"* ise *"...DEN/...DAN"* anlamı verir. Yani *"SELECT FROM faturalar"* dediğimizde *"faturalardan seç"* demiş oluyoruz... Burada kullandığımız `"*"` işareti de *"her şey"* anlamına geldiği için, *"SELECT \* FROM faturalar"* ifadesi *"faturalardan her şeyi seç"* gibi bir anlama gelmiş oluyor.

Betiğimizi yazmaya devam edelim:

```
veriler = im.fetchall()
```

Burada da ilk defa gördüğümüz bir metot var: `fetchall()`. Gördüğünüz gibi, `fetchall()` imlecin bir metodudur. Yukarıda gördüğümüz `SELECT...FROM...` komutu bir tablodaki verileri seçiyordu. `fetchall()` metodu ise seçilen bütün verileri alma işlevi görür. Yukarıda biz `fetchall()` metoduyla aldığımız bütün verileri `veriler` adlı bir değişkene atadık.

Artık bu verileri rahatlıkla yazdırabiliriz:

```
print veriler
```

Dilerseniz betiğimizi topluca görelim:

```
# -*- coding: utf-8 -*-

import sqlite3

vt = sqlite3.connect(":memory:")

im = vt.cursor()
im.execute("""CREATE TABLE faturalar
(fatura, miktar, ilk_odeme_tarihi, son_odeme_tarihi)""")

im.execute("""INSERT INTO faturalar VALUES
('Elektrik', 45, '23 Ocak 2010', '30 Ocak 2010')""")

vt.commit()

im.execute("""SELECT * FROM faturalar""")

veriler = im.fetchall()

print veriler
```

Bu betiği çalıştırdığımızda şöyle bir çıktı alırız:

```
[(u'Elektrik', 45, u'23 Ocak 2010', u'30 Ocak 2010')]
```

Gördüğünüz gibi, veriler bir liste içinde demet halinde yer alıyor. Ama tabii siz bu verileri istediğiniz gibi biçimlendirecek kadar Python bilgisine sahipsiniz.

Bu arada, tablo oluştururken sütun adlarında boşluk (ve Türkçe karakter) kullanmak iyi bir fikir değildir. Mesela ilk ödeme tarihi yerine `ilk_odeme_tarihi` ifadesini tercih edin. Eğer kelimeler arasında mutlaka boşluk bırakmak isterseniz bütün kelimeleri tırnak içine alın. Mesela: *"ilk ödeme tarihi"* veya *"ilk ödeme tarihi"*.

## 21.8 Veritabanı Güvenliği - SQL Injection

Python'da veritabanları ve Sqlite konusunda daha fazla ilerlemeden önce çok önemli bir konudan bahsetmemiz gerekiyor. Tahmin edebileceğiniz gibi, veritabanı denen şey oldukça hassas bir konudur. Bilgiyi bir araya toplayan bu sistem, içerdeki bilgilerin değerine ve önemine de bağlı olarak üçüncü şahısların ağzını sulandırabilir. Ancak depoladığınız verilerin ne kadar değerli ve önemli olduğundan bağımsız olarak veritabanı güvenliğini sağlamak, siz programcıların asli görevidir.

Peki veritabanı yönetim sistemleri acaba hangi tehditlerle karşı karşıya?

SQL komutlarını işleten bütün veritabanları için günümüzdeki en büyük tehditlerden birisi hiç kuşkusuz kötü niyetli kişilerin SQL'e sızma (*SQL injection*) girişimleridir.

Şimdi şöyle bir şey düşünün: Diyelim ki siz bir alışveriş karşılığı birine 100.000 TL'lik bir çek verdiniz. Ancak çeki verdiğiniz kişi bu çek üzerindeki miktarı tahrif ederek artırdı ve banka da tahrif edilerek artırılan bu miktarı çeki getiren kişiye (hamiline) ödedi. Böyle bir durumda epey başınız ağrıyacaktır.

İşte böyle tatsız bir durumla karşılaşmamak için, çek veren kişi çekin üzerindeki miktarı hem rakamla hem de yazıyla belirtmeye özen gösterir. Ayrıca rakam ve yazılara ekleme yapılmasını da engellemek için rakam ve yazıların sağına soluna "#" gibi işaretler de koyar. Böylece çeki alan kişinin, kendisine izin verilenden daha fazla bir miktarı elde etmesini engellemeye çalışır.

Yukarıdakine benzer bir şey veritabanı uygulamalarında da karşımıza çıkabilir. Şimdi şu örneğe bakalım:

```
# -*- coding: utf-8 -*-

import sqlite3

#Deneme amaçlı bir çalışma yaptığımız için veritabanımızı
#bellek üzerinde oluşturunuz.
db = sqlite3.connect(":memory:")

#Veritabanı üzerinde istediğimiz işlemleri yapabilmek
#için bir imleç oluşturmamız gerekiyor.
im = db.cursor()

#imlecin execute() metodunu kullanarak, veritabanı içinde
#"kullanıcılar" adlı bir tablo oluşturunuz. Bu tabloda
#kullanıcı_adi ve parola olmak üzere iki farklı sütun var.
im.execute("""CREATE TABLE kullanıcılar (kullanıcı_adi, parola)""")

#Yukarıda oluşturduğumuz tabloya yerleştireceğimiz verileri
#hazırlıyoruz. Verilerin liste içinde birer demet olarak
#nasıl gösterildiğine özellikle dikkat ediyoruz.
veriler = [
    ("ahmet123", "12345678"),
    ("mehmet321", "87654321"),
    ("selin456", "123123123")
]

#veriler adlı liste içindeki bütün verileri kullanıcılar adlı
#tabloya yerleştiriyoruz. Burada tek öğeli bir demet
#tanımladımıza dikkat edin (i,)
for i in veriler:
    im.execute("""INSERT INTO kullanıcılar VALUES %s""" % (i,))
```

```

#Yaptığımız değişikliklerin tabloya işlenebilmesi için
#commit() metodunu kullanıyoruz.
db.commit()

#Kullanıcıdan kullanıcı adı ve parola bilgilerini alıyoruz...
kull = raw_input("Kullanıcı adınız: ")
paro = raw_input("Parolanız: ")

#Burada yine bir SQL komutu işletiyoruz. Bu komut, kullanıcılar
#adlı tabloda yer alan kullanıcı_adi ve parola adlı sütunlardaki
#bilgileri seçiyor.
im.execute("""SELECT * FROM kullanıcılar WHERE
kullanıcı_adi = '%s' AND parola = '%s'"""%(kull, paro))

#Hatırlarsanız daha önce fetchall() adlı bir metottan
#söz etmiştik. İşte bu fetchone() metodu da ona benzer.
#fetchall() bütün verileri alıyordu, fetchone() ise
#verileri tek tek alır.
data = im.fetchone()

#Eğer data adlı değişken False değilse, yani bu
#değişkenin içinde bir değer varsa kullanıcı adı
#ve parola doğru demektir. Kullanıcıyı içeri alıyoruz.
if data:
    print u"Programa hoşgeldin %s!" % data[0]

#Aksi halde kullanıcıya olumsuz bir mesaj veriyoruz.
else:
    print u"Parola veya kullanıcı adı yanlış!"

```

Bu örnekte henüz bilmediğimiz bazı kısımlar var. Ama siz şimdilik bunları kafanıza takmayın. Nasıl olsa bu kodlarda görünen her şeyi biraz sonra tek tek öğreneceğiz. Siz şimdilik sadece işin özüne odaklanın.

Yukarıdaki kodları çalıştırdığınızda, eğer kullanıcı adı ve parolayı doğru girerseniz *Programa hoşgeldin* çıktısını göreceksiniz. Eğer kullanıcı adınız veya parolanız yanlışsa bununla ilgili bir uyarı alacaksınız.

Her şey iyi hoş, ama bu kodlarda çok ciddi bir problem var.

Dediğimiz gibi, bu kodlar çalışırken eğer kullanıcı, veritabanında varolan bir kullanıcı adı ve parola yazarsa sisteme kabul edilecektir. Eğer doğru kullanıcı adı ve parola girilmezse sistem kullanıcıya giriş izni vermeyecektir. Ama acaba gerçekten öyle mi?

Şimdi yukarıdaki programı tekrar çalıştırın. Kullanıcı adı ve parola sorulduğunda da her ikisi için şunu yazın:

```
x' OR '1' = '1'
```

O da ne! Program sizi içeri aldı... Hem de kullanıcı adı ve parola doğru olmadığı halde... Hatta şu kodu sadece kullanıcı adı kısmına girip parola kısmını boş bırakmanız da sisteme giriş hakkı elde etmenize yetecektir.:

```
x' OR '1' = '1' --
```

İşte yukarıda gösterdiğimiz bu işleme "SQL'e sızma" (SQL injection) adı verilir. Kullanıcı, tıpkı en başta verdiğimiz tahrif edilmiş çek örneğinde olduğu gibi, sistemin zaaflarından yararlanarak, elde etmeye hakkı olandan daha fazlasına erişim hakkı elde ediyor.



Burada en basit şekliyle bool işleçlerinden biri olan or'dan yararlanıyoruz. or'un nasıl işlediğini gayet iyi biliyorsunuz, ama ben yine de birkaç örnekle or'un ne olduğunu ve ne yaptığını size hatırlatayım. Şu örneklerle bakın:

```
>>> a = 21
>>> a == 22
False
>>> b = 13
>>> b == 13
True
>>> if a == 22 and b == 13:
...     print "Merhaba!"
...
>>> if a == 22 or b == 13:
...     print "Merhaba!"
...
Merhaba!
```

Örneklerden de gördüğümüz gibi, and işlecinin True sonucunu verebilmesi için her iki önermenin de doğru olması gerekir. O yüzden a == 22 and b == 13 gibi bir ifade False değeri veriyor. Ancak or işlecinin True sonucu verebilmesi için iki önermeden sadece birinin doğru olması yeterlidir. Bu yüzden, sadece b == 13 kısmı True olduğu halde a == 22 or b == 13 ifadesi True sonucu veriyor... İşte biz de yukarıdaki SQL'e sızma girişiminde or'un bu özelliğinden faydalanıyoruz.

Dilerseniz neler olup bittiğini daha iyi anlayabilmek için, sızdırılan kodu doğrudan ilgili satıra uygulayalım:

```
im.execute("""SELECT * FROM kullanicilar WHERE
kullanici_adi = 'x' OR '1' = '1' AND parola = 'x' OR '1' = '1'""")
```

Sanırım bu şekilde neler olup bittiği daha net görülüyor. Durumu biraz daha netleştirmek için Python'u yardıma çağırabiliriz:

```
>>> kullanıcı_adi = 'ahmet123'
>>> parola = '12345678'
>>> kullanıcı_adi == 'x'
False
>>> '1' == '1'
True
>>> kullanıcı_adi == 'x' or '1' == '1'
True
>>> parola == 'x'
```

```
False
```

```
>>> (kullanici_adi == 'x' or '1' == '1') and (parola == 'x' or '1' == '1')
```

```
True
```

'1' == '1' ifadesi her zaman True değeri verecektir. Dolayısıyla kullanıcı adının ve parolanın doğru olup olmaması hiçbir önem taşımaz. Yani her zaman True değerini vereceği kesin olan ifadeler yardımıyla yukarıdaki gibi bir sızma girişiminde bulunabilirsiniz.

Yukarıda yaptığımız şey, '%s' ile gösterilen yerlere kötü niyetli bir SQL komutu sızdırmaktan ibarettir. Burada zaten başlangıç ve bitiş tırnakları olduğu için sızdırılan kodda başlangıç ve bitiş tırnaklarını yazmıyoruz. O yüzden sızdırılan kod şöyle görünüyor:

```
x' OR '1' = '1
```

Gördüğünüz gibi, x'in başındaki ve 1'in sonundaki tırnak işaretleri koymuyoruz.

Peki yukarıda verdiğimiz şu kod nasıl çalışıyor:

```
x' OR '1' = '1' --
```

Python'da yazdığımız kodlara yorum eklemek için "#" işaretinden yararlandığımızı biliyorsunuz. İşte SQL kodlarına yorum eklemek için de "--" işaretlerinden yararlanılır. Şimdi derseniz yukarıdaki kodu doğrudan ilgili satıra uygulayalım ve ne olduğunu görelim:

```
im.execute("""SELECT * FROM kullanicilar WHERE  
kullanici_adi = 'x' OR '1'='1' --AND parola = '%s'""")
```

Burada yazdığımız "--" işareti AND parola = '%s' kısmının sistem tarafından yorum olarak algılanmasını sağlıyor. Bu yüzden kodların bu kısmı işletilmiyor. Dolayısıyla da sisteme giriş yapabilmek için sadece kullanıcı adını girmemiz yeterli oluyor!.. Burada ayrıca kodlarımızın çalışması için 1'in sonuna bir adet tırnak yerleştirerek kodu kapattığımızı dikkat edin. Çünkü normal bitiş tırnağı yorum tarafında kaldı...

Dikkat ederseniz SQL'e sızdığımızda "ahmet123" adlı kullanıcının hesabını ele geçirmiş olduk. Peki neden ötekiler değil de "ahmet123"? Bunun sebebi, "ahmet123" hesabının tablonun en başında yer alması. Eğer tablonun başında "admin" diye bir hesap olmuş olsaydı, veritabanına azami düzeyde zarar verme imkanına kavuşacaktınız...

Peki SQL'e sızma girişimlerini nasıl önleyeceğiz? Bu girişime karşı alabileceğiniz başlıca önlem "%s" işaretlerini kullanmaktan kaçınmak olacaktır. Bu işaret yerine "?" işaretini kullanacaksınız. Yani yukarıdaki programı şöyle yazacağız:

```
# -*- coding: utf-8 -*-
```

```
import sqlite3
```

```
db = sqlite3.connect(":memory:")
```

```
im = db.cursor()
```

```
im.execute("""CREATE TABLE kullanicilar (kullanici_adi, parola)""")
```

```
veriler = [  
    ("ahmet123", "12345678"),  
    ("mehmet321", "87654321"),  
    ("selin456", "123123123")  
]
```

```

for i in veriler:
    im.execute("""INSERT INTO kullanicilar VALUES (?, ?)""", i)

db.commit()

kull = raw_input("Kullanıcı adınız: ")
paro = raw_input("Parolanız: ")

im.execute("""SELECT * FROM kullanicilar WHERE
kullanici_adi = ? AND parola = ?""", (kull, paro))

data = im.fetchone()

if data:
    print u"Programa hoşgeldin %s!" % data[0]
else:
    print u"Parola veya kullanıcı adı yanlış!"

```

Dediğimiz gibi, SQL’e sızma girişimlerine karşı alabileceğiniz başlıca önlem “%s” işaretleri yerine “?” işaretini kullanmak olmalıdır. Bunun dışında, SQL komutlarını işletmeden önce bazı süzgeçler uygulamak da güvenlik açısından işinize yarayabilir. Örneğin kullanıcıdan alınacak verileri alfanümerik karakterlerle [<http://www.istihza.com/blog/alfanumerik-ne-demek.html/>] sınırlayabilirsiniz:

```

if kull.isalnum() and paro.isalnum():
    im.execute("""SELECT * FROM kullanicilar WHERE
kullanici_adi = '%s' AND parola = '%s'"""%(kull, paro))

```

Böylece kullanıcının bazı “tehlikeli” karakterleri girmesini engelleyebilir, onları sadece harf ve sayı girmeye zorlayabilirsiniz.

Her halükarda unutmamamız gereken şey, güvenliğin çok boyutlu bir kavram olduğudur. Birkaç önlemle pek çok güvenlik açığını engelleyebilirsiniz, ancak bütün güvenlik açıklarını bir çırpıda yamamak pek mümkün değildir. Bir programcı olarak sizin göreviniz, yazdığınız programları güvenlik açıklarına karşı sürekli taramak ve herhangi bir açık ortaya çıktığında da bunu derhal kapatmaya çalışmaktır. **MODÜL DİZİNİ**

# sys Modülü

Python'daki en önemli modüllerden biri *os* ise, öbürü de *sys* adlı modüldür. Bu modüller o kadar önemlidir ki, Python'daki hiç bir modülü bilmeseniz de bu iki modülü bilmeniz gerekir. *os* modülünü, "Modüller" konusunu işlerken anlatmıştık. Şimdi de *sys* modülünü inceleyeceğiz.

*sys* modülü, Python sistemine ilişkin fonksiyonlar ve nitelikler barındırır. Bu modülü kullanarak, elinizdeki Python sürümünü yönetebilirsiniz. Bunun tam olarak ne demek olduğunu biraz sonra gayet net bir şekilde anlayacaksınız.

"Modüller" konusunu anlatırken, modüllerin aslında *.py* uzantılı birer Python programı olduğunu söylemiştik. Mesela daha önce incelediğimiz *os* modülü, bilgisayarımızda bulunan *os.py* adlı bir dosyaydı. Python'daki hemen hemen bütün modüller böyledir. Yani hemen hemen bütün Python modülleri birer *.py* dosyası olarak sistemimizde bulunur. Ancak bu durumun bazı istisnaları da vardır. Örneğin *sys* modülü bu istisnalardan biridir. Bu modül, öteki pek çok modülün aksine Python programlama dili ile değil C programlama dili ile yazılmıştır. Dolayısıyla bu modülün Python kodları yoktur. Bu modülün C kodlarını görmek için <http://svn.python.org/projects/python/branches/release26-maint/Python/sysmodule.c> adresini ziyaret edebilirsiniz.

## 22.1 sys Modülünün İçeriği

Her zaman olduğu gibi, *sys* modülünün içinde ne olduğunu görmek için *dir()* fonksiyonundan yararlanabilirsiniz:

```
>>> dir(sys)
```

```
['__displayhook__', '__doc__', '__egginsert', '__excepthook__',
'__name__', '__package__', '__plen', '__stderr__', '__stdin__',
'__stdout__', '_clear_type_cache', '_current_frames', '_getframe',
'api_version', 'argv', 'builtin_module_names', 'byteorder',
'call_tracing', 'callstats', 'copyright', 'displayhook',
'dont_write_bytecode', 'exc_clear', 'exc_info', 'exc_type',
'excepthook', 'exec_prefix', 'executable', 'exit', 'flags',
'float_info', 'getcheckinterval', 'getdefaultencoding',
'getdlopenflags', 'getfilesystemencoding', 'getprofile',
'getrecursionlimit', 'getrefcount', 'getsizeof', 'gettrace',
'hexversion', 'maxint', 'maxsize', 'maxunicode', 'meta_path',
'modules', 'path', 'path_hooks', 'path_importer_cache', 'platform',
```

```
'prefix', 'ps1', 'ps2', 'py3kwarning', 'pydebug', 'setcheckinterval',  
'setdlopenflags', 'setprofile', 'setrecursionlimit', 'settrace',  
'stderr', 'stdin', 'stdout', 'subversion', 'version',  
'version_info', 'warnoptions']
```

Gördüğünüz gibi sys modülü içinde epey fonksiyon ve nitelik var. Biz bu fonksiyon ve nitelikler arasından en önemlilerini incelemeye çalışacağız. Bu bölümde şu fonksiyon ve nitelikleri inceleyeceğiz:

1. argv
2. exit
3. getdefaultencoding
4. path
5. platform
6. stdout
7. version\_info

O halde ilk niteliğimizle başlayalım...

## 22.2 argv Niteliği

Bu nitelik sys modülünün en bilinen ve en sık kullanılan niteliğidir. Dolayısıyla bu niteliği öğrenirken bu niteliğe özel önem ve ilgi göstermenizi öneririm.

Etrafta şöyle programları sıkça görmüş olmalısınız: Komut satırında programın adı ile birlikte birtakım seçenekler de belirtirsiniz ve ilgili program o belirttiğiniz seçeneklere göre programı işletir. Mesela elimizde “muzikcalar” adlı bir program olduğunu varsayalım. Amacı müzik çalmak olan program mesela şöyle kullanılıyor olabilir:

```
muzikcalar sarki_adi.mp3
```

Bu komutla “sarki\_adi.mp3” adlı şarkıyı çalabiliriz... Böyle bir programı Python’da yazmak için argv niteliğinden yararlanabiliriz. Şimdi şu kodlara bakın:

```
#!/usr/bin/env python  
# -*- coding: utf-8 -*-  
  
import sys  
import os  
  
def sarkiyi_cal():  
    if os.name == "nt":  
        os.startfile(sys.argv[1])  
  
    elif os.name == "posix":  
        os.system("xdg-open %s" %sys.argv[1])  
  
if len(sys.argv) < 2:  
    print "Çalınacak şarkı adını belirtmediniz!"  
  
elif len(sys.argv) > 2:  
    print "Birden fazla dosya adı belirttiniz!"
```

```
else:
    sarkiyi_cal()
```

Gelin isterseniz bu kodları açıklamaya başlamadan önce, argv niteliğinin tam olarak nasıl bir işe yaradığını anlamak için şöyle bir şey yazalım:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import sys

print sys.argv
```

Şimdi bu programı *argv\_testi.py* adıyla kaydedin ve programı `python argv_testi.py` komutuyla çalıştırın. Bu programı çalıştırdığınızda şöyle bir çıktı alacaksınız:

```
istihza@istihza:~/Desktop$ python argv_testi.py

['argv_testi.py']
```

Gördüğünüz gibi `sys` modülünün `argv` niteliği aslında basit bir listeden ibarettir. Bu listenin ilk ögesi de yazdığımız programın adıdır (*argv\_testi.py*). `argv` niteliği bir liste olduğu için listelerle yapabildiğiniz her şeyi bu nitelikte de yapabilirsiniz.

Şimdi aynı programı şu şekilde çalıştırın:

```
istihza@istihza:~/Desktop$ python argv_testi.py falanca filanca
```

Bu defa şöyle bir çıktı alacağız:

```
['argv_testi.py', 'falanca', 'filanca']
```

Gördüğünüz gibi, yazdığınız programa eklediğiniz her seçenek `argv` listesine sırayla ekleniyor. `argv` listesinin ilk ögesi her zaman programınızın adıdır. Dolayısıyla eğer `argv` listesinin uzunluğu 2'den azsa, programınız herhangi bir seçenek belirtilmeden, sadece ismi belirtilerek çalıştırılmış demektir. Bu bilgiyi kullanarak şöyle bir program yazabilirsiniz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import sys

if len(sys.argv) < 2:
    print "Herhangi bir seçenek belirtmediniz!"

elif len(sys.argv) >= 2:
    print "Programı şu seçeneklerle birlikte çalıştırdınız: "
    for i in sys.argv[1:]:
        print i
```

Dediğimiz gibi, eğer `argv` niteliğinin gösterdiği listenin uzunluğu 2'den azsa kullanıcı hiç bir seçenek belirtmemiş demektir (`if len(sys.argv) < 2:`).

Ama eğer `argv` niteliğinin gösterdiği listenin uzunluğu 2 veya daha fazla ise kullanıcımız programı çalıştırırken bazı seçenekler belirtmiş demektir (`elif len(sys.argv) >= 2:`). Bu seçenekleri `argv` listesinin ilk ögesi dışındaki bütün öğelerini ekrana yazdırarak görebilirsiniz:

```
for i in sys.argv[1:]:  
    print i
```

İlk öğeyi ekrana yazdırmaya gerek yok. Çünkü bildiğiniz gibi, ilk öğe her zaman programımızın adını gösteriyor...

Bütün bu açıklamalardan sonra, ilk başta yazdığımız programı anlamak çok daha kolay bir hal almış olmalı. Programımızı tekrar görelim:

```
#!/usr/bin/env python  
# -*- coding: utf-8 -*-  
  
import sys  
import os  
  
def sarkiyi_cal():  
    if os.name == "nt":  
        os.startfile(sys.argv[1])  
  
        elif os.name == "posix":  
            os.system("xdg-open %s" %sys.argv[1])  
  
if len(sys.argv) < 2:  
    print "Çalınacak şarkı adını belirtmediniz!"  
  
elif len(sys.argv) > 2:  
    print "Birden fazla dosya adı belirttiniz!"  
  
else:  
    sarkiyi_cal()
```

Burada öncelikle `sys` ve `os` modüllerini içe aktarıyoruz. Çünkü programımız içinde bu iki modüle ait nitelik ve fonksiyonlardan yararlanacağız.

Ardından `sarkiyi_cal()` adlı bir fonksiyon yazdık. Eğer kullanılan işletim sistemi Windows ise `argv` listesinin birinci öğesini, yani kullanıcının program adından sonra yazdığı ilk seçeneği, `os` modülünün `startfile()` fonksiyonunu kullanarak çalıştırıyoruz. Eğer kullanılan işletim sistemi GNU/Linux ise, `os` modülünün `system()` fonksiyonunu kullanarak yine `argv` listesinin ilk öğesini çalıştırıyoruz.

Elbette kullanıcının program adından sonra hiç bir seçenek belirtmeme veya 2'den fazla seçenek belirtme ihtimali de var. İşte bu ihtimallere karşı şu kodları yazıyoruz:

```
if len(sys.argv) < 2:  
    print "Çalınacak şarkı adını belirtmediniz!"  
  
elif len(sys.argv) > 2:  
    print "Birden fazla dosya adı belirttiniz!"
```

Eğer kullanıcımız program adının yanına sadece 1 adet seçenek eklerse, o zaman `sarkiyi_cal()` adlı fonksiyonu devreye sokuyoruz. Peki size bir soru: Bu haliyle programımız müzik dosyası veya başka bir tür dosya diye bir ayırmda bulunmaksızın bütün dosyaları, sistemde o türdeki dosyayı açan öntanımlı uygulamayla çalıştıracaktır. Acaba biz bu programda kullanıcının girdiği dosya adlarının müzik dosyası olup olmadığını nasıl denetler ve programımızı sadece müzik dosyalarını çalıştıracak hale getirebiliriz?

`argv` niteliği başka pek çok faydalı iş için kullanılabilecek harika bir araçtır. Mesela şu programa bir bakın:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import sys

surum = "0.2"

yardim = """
FALANCA programının %s sürümünü kullanıyorsunuz.
Bu programı kullanabilmek için sisteminizde Python
dışında falanca adlı modülün de kurulu olması gerekiyor.
Programı komut satırından python falanca.py komutuyla
çalıştırabilirsiniz. İyi eğlenceler!""" % surum

if len(sys.argv) < 2:
    print "... "

elif sys.argv[1] == "-v":
    print surum

elif sys.argv[1] == "-h":
    print yardim

else:
    print "Böyle bir seçenek yok!"
```

Gördüğünüz gibi bu programda kullanıcılarımıza programın sürümünü ve yardım dosyasını görüntüleme imkanı veriyoruz. Eğer kullanıcımız şu komutu verirse kullandığı programın sürümünü öğrenebilir:

```
python falanca.py -v
```

Ya da eğer şu komutu verirse programın yardım dosyasına ulaşabilir:

```
python falanca.py -h
```

Eğer kullanıcı “-v” ve “-h” dışında başka bir şey yazarsa programımız kendisini böyle bir seçenek olmadığı konusunda uyaracaktır.

## 22.3 exit() Fonksiyonu

sys modülünün içindeki fonksiyonlardan biri de `exit()` fonksiyonudur. Aslında siz bu fonksiyonu daha önce de görmüştünüz. Hatırlarsanız ilk derslerimizin birinde bu fonksiyonun Python’un etkileşimli kabuğundan çıkmak için kullanıldığını söylemiştik. Mesela yukarıda yazdığımız programda bu fonksiyonu kullanmak istersek şöyle bir şey yazabiliriz:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import sys

surum = "0.2"

yardim = """
FALANCA programının %s sürümünü kullanıyorsunuz.
Bu programı kullanabilmek için sisteminizde Python
```



```
dışında filanca adlı modülün de kurulu olması gerekiyor.  
Programı komut satırından python falanca.py komutuyla  
çalıştırabilirsiniz. İyi eğlenceler!""" % surum
```

```
if len(sys.argv) < 2:  
    sys.exit()  
  
elif sys.argv[1] == "-v":  
    print surum  
  
elif sys.argv[1] == "-h":  
    print yardım  
  
else:  
    print "Böyle bir seçenek yok!"
```

Böylece eğer argv listesinin uzunluğu 2’den azsa programımız kapanacaktır.

Bu fonksiyonun çok fazla bir özelliği yoktur. Gördüğümüz gibi *exit()* fonksiyonunu programdan çıkarmak için kullanıyoruz.

## 22.4 getdefaultencoding() Fonksiyonu

Bildiğiniz gibi, yazdığımız bir programda Türkçe karakterler kullanabilmemiz için programımızın en başına şuna benzer bir satır eklememiz gerekiyor:

```
# -*- coding: utf-8 -*-
```

Böylece Python’a, program içinde kullanacağımız kodlama biçiminin “utf-8” olacağını bildirmiş oluyoruz. Eğer bu satırı yazmazsak Python otomatik olarak “ascii” adlı bir kodlama biçimini kullanacaktır.

---

**Not:** “utf-8”, “ascii” ve unicode konularında ayrıntılı bilgi edinmek için “ascii, unicode ve Python” başlıklı konuyu inceleyebilirsiniz.

---

Python’un öntanımlı kodlama biçiminin ne olduğunu öğrenmek için *getdefaultencoding()* adlı fonksiyondan yararlanabilirsiniz:

```
>>> sys.getdefaultencoding()  
  
'ascii'
```

Gördüğümüz gibi Python öntanımlı olarak ‘ascii’ adlı kodlama biçimini kullanıyor. Python’da yaşadığımız pek çok Türkçe karakter probleminin nedeni budur.

---

**Not:** Python’un 3.x sürümlerinde öntanımlı kodlama biçimi “utf-8” olarak değiştirildi. Bu yüzden Python’un 3.x sürümlerinde Türkçe karakter problemleri çok daha az olacaktır.

---

## 22.5 path Niteliği

Modüller konusu içinde *os* modülünü anlatırken *sys* modülünün *path* adlı niteliğinden söz etmiştik. Orada da söylediğimiz gibi, Python içe aktarmaya çalıştığımız bir modülü ararken *path*

niteliğinin gösterdiği dizinlerin içine bakar.

Bu niteliği şöyle kullanıyoruz:

```
>>> import sys
>>> sys.path
```

Bu komutun çıktısının, komutu verdiğiniz işletim sistemine göre farklılık göstereceğini biliyorsunuz.

Tıpkı `argv` niteliği gibi, `path` niteliği de bir listedir. Dolayısıyla listeler üzerinde yaptığınız her şeyi `path` niteliği üzerinde de yapabilirsiniz. Mesela listelerin `insert()` veya `append()` metotlarını kullanarak `path` listesine yeni öğeler ekleyebilirsiniz:

```
>>> sys.path.append("/herhangi/bir/dizin")
```

Bu komut, eklediğiniz dizini listenin en sonuna ilâştirecektir. Eğer dizininizi listenin en başına eklemek isterseniz şu komutu kullanabilirsiniz:

```
>>> sys.path.insert(0, "/herhangi/bir/dizin")
```

Python içe aktaracağınız bir modülü `path` listesinde ararken listenin başından sonuna doğru ilerler ve modülü bulduğu anda arama işlemini durdurur. Dolayısıyla, mesela bir modül iki farklı dizin içinde yer alıyorsa, Python listede soldan sağa doğru bulduğu ilk modülü dikkate alacaktır. O yüzden yazdığınız programlarda bazen bir dizini bu listenin en başına eklemeniz gerekebilir.

## 22.6 platform Niteliği

Hatırlarsanız kullanılan işletim sistemini tespit etmek için `os` modülünün `name` adlı niteliğini kullanıyorduk. Aynen buna benzer bir şekilde, kullanılan işletim sistemini tespit etmek için `sys` modülünün `platform` adlı niteliğinden de yararlanabiliriz:

```
>>> sys.platform

'linux2'
```

Ben bu komutu GNU/Linux üzerinde verdiğim için aldığım çıktı `linux2` oldu. Eğer aynı komutu Windows üzerinde vererseniz şöyle bir çıktı alırsınız:

```
>>> sys.platform

'win32'
```

Bu komut Mac Os X'te ise şöyle bir çıktı verir:

```
>>> sys.platform

'darwin'
```

Bu nitelik ile ilgili şöyle bir uyarı yapalım: Windows mimariniz 32 bit de olsa 64 bit de olsa bu nitelik her zaman `win32` çıktısı verecektir. Dolayısıyla bu niteliği kullanarak 32/64 bit ayrımı yapamazsınız. Kullanılan işletim sisteminin 32 bit mi yoksa 64 bit mi olduğunu tespit edebilmek için `platform` adlı başka bir modülden yararlanabilirsiniz:

```
>>> import platform
>>> platform.architecture()
('32bit', 'ELF')
```

Ben bu komutu 32 bitlik bir GNU/Linux işletim sisteminde verdiğim için çıktı yukarıdaki gibi oldu. Eğer sistemim 64 bit olsaydı şöyle bir çıktı alacaktım:

```
>>> platform.architecture()
('64bit', 'ELF')
```

Bu komut 32 bitlik Windows işletim sistemlerin şu çıktıyı verir:

```
>>> platform.architecture()
('32bit', 'WindowsPE')
```

64 bitlik Windows işletim sisteminde ise şu çıktıyı alacaksınız:

```
>>> platform.architecture()
('64bit', 'WindowsPE')
```

Gördüğünüz gibi bu çıktılar birer demettir. Dolayısıyla demetlere nasıl davranıyorsanız bu çıktıların da öyle davranabilirsiniz.

## 22.7 stdout Niteliği

Bildiğiniz gibi Python'da ekrana herhangi bir şey yazdırabilmek için *print* deyiminden yararlanıyoruz:

```
>>> print "merhaba!"
merhaba!
```

Python *print* deyimlerinden sonra otomatik olarak yeni satıra geçer. Yani:

```
>>> for i in range(10):
...     print "%5s. satır" %(int(i)+1)
...
1. satır
2. satır
3. satır
4. satır
5. satır
6. satır
7. satır
8. satır
9. satır
10. satır
```

Gördüğünüz gibi, her satırdan sonra bir alt satıra geçiliyor. *print* deyiminin bir sonraki satıra geçmesini engellemek için virgülden yararlanabiliriz:

```
>>> for i in range(10):
...     print i,
...
0 1 2 3 4 5 6 7 8 9
```

Burada çıktıya dikkat edin. Sayılar alt alta değil, yan yana diziliyor. Bunu sağlayan şey `print i` satırından sonra getirdiğimiz bir adet virgül işaretidir... Ancak burada her sayının arasında birer adet de boşluk karakteri var. İstediğiniz şey bu olabilir. Ama olmaya da bilir... Eğer hem öğeleri yan yana dizmek hem de öğeler arasında boşluk bırakmamak istiyorsanız `sys` modülünün `stdout` niteliğini kullanabilirsiniz. Daha doğrusu bu iş için `sys` modülünün `stdout` niteliğinin `write()` adlı metodunu kullanacağız. Hemen bir örnek verelim:

```
>>> sys.stdout.write("Merhaba!")
```

```
Merhaba!>>>
```

Gördüğünüz gibi, `sys.stdout.write()` *print* niteliğiyle aynı işi görüyor. Tek farkları, *print* niteliğinin işini bitirdikten sonra yeni satıra geçerken `sys.stdout.write()` metodunun bunu yapmaması. Bu durumu daha net bir şekilde görebilmek için şöyle bir örnek verebiliriz:

```
>>> for i in range(10):
...     sys.stdout.write(str(i))
...
0123456789>>>
```

Gördüğünüz gibi, `sys.stdout.write()` metodu bütün sayıları yan yana dizdi ve satır sonunda da yeni satıra geçmedi. Bu kodlarda dikkat etmemiz gereken bir başka şey de `sys.stdout.write()` metoduna verdiğimiz argümanın bir karakter dizisi olması zorunludur. `sys.stdout.write()` metodu sayıları ekrana basamaz:

```
>>> for i in range(10):
...     sys.stdout.write(i)
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: argument 1 must be string or
read-only character buffer, not int
```

Böyle bir hata almamak için, *for* döngüsü içinde "i" değişkenini `str()` fonksiyonunu kullanarak karakter dizisine dönüştürüyoruz.

`sys.stdout.write()` komutunu kullanabileceğiniz faydalı bir örnek şöyle olabilir:

```
>>> import sys, time

>>> for i in range(20):
...     time.sleep(1)
...     sys.stdout.write("%5\r" %i)
...     sys.stdout.flush()
```

Bu kodları çalıştırdığınızda 0'dan 20'ye kadar olan sayılar birer saniye aralıklarla tek bir satıra yazdırılacaktır. Ayrıca bu sayılar yan yana değil, birbirlerinin yerine geçerek ekrana basılacaktır... Peki bu kodlar nasıl çalışıyor? Hemen açıklayalım:

1. İlk satırda `sys` ve `time` adlı iki modülü içe aktardık. Bunlardan `sys` modülünü biliyoruz. `time` modülünü ise henüz öğrenmedik. Ama bu modülü de zamanı geldiğinde inceleyeceğiz. Şimdilik biz bu `time` adlı modülün, Python'da tarih ve zamanla ilgili işlemler yapmamızı sağlayan bir modül olduğunu bilelim yeter.

2. Sonraki satırda bir *for* döngüsü kurduk. Bu döngüde 0'dan 20'ye kadar olan sayıları ekrana basacağız.

3. *for* döngüsünün ilk satırında `time.sleep(1)` gibi bir kod görüyoruz. Burada *time* modülünün *sleep()* adlı fonksiyonunu kullandık. Bu fonksiyon, yapılan işlemlere belli sürelerde zaman aralıkları koymamızı sağlar. Biz burada bu süre aralığını 1 saniye olarak belirledik. Bizim kodlarımızda 0'dan 20'ye kadar olan sayılar 1'er saniye aralıklarla ekrana basılacak.

4. Sonraki satırda `sys.stdout.write("%s\r" %i)` komutunu görüyoruz. Bu kod 0'dan 20'ye kadar olan sayıların alt alta değil, yan yana yazılmasını sağlıyor. Burada bir de `"\r"` ifadesini görüyoruz. Aslında bu bir kaçış dizisidir. Görevi ise bir karakter dizisinin 0. konumuna dönülmesini sağlamaktır. Bu kaçış dizisinin nasıl çalıştığını anlamak için şöyle bir örnek verebiliriz:

```
>>> print "merhaba\ristihza"

istihza
```

Burada olan şey şu: "merhaba" karakter dizisinin hemen ardından `"\r"` kaçış dizisinin etkisiyle 0. konuma, yani karakter dizisinin en başına dönülüyor ve "istihza" karakter dizisi "merhaba" karakter dizisinin üzerine yazılıyor. "istihza" karakter dizisi ile "merhaba" karakter dizisi aynı sayıda karakterden oluştuğu için "merhaba" karakter dizisi siliniyor, onun yerine sadece "istihza" karakter dizisi çıktıda görünüyor. Yukarıdaki `sys.stdout.write("%s\r" %i)` komutunda da `"\r"` kaçış dizisi sayılar ekrana döküldükçe en başa dönülmesini, böylece bir sonraki sayının bir önceki sayıyı silmesini sağlıyor.

5. Son satırdaki `sys.stdout.flush()` ise sayıların ekranda hemen görünebilmesini sağlıyor. Dilerseniz bu satırı kaldırıp programı öyle çalıştırmayı deneyerek bu satırın tam olarak ne işe yaradığını görebilirsiniz. (Bazı işletim sistemlerinde bu satır gereksizdir. Ancak yazdığınız programın birden fazla işletim sisteminde düzgün çalışmasını istiyorsanız bu satırı eklemeniz gerekir.)

Aslında yukarıda yazdığımız kodları şöyle de yazabilirdik:

```
>>> import sys, time

>>> for i in range(20):
...     time.sleep(1)
...     print "%s\r" %i,
...     sys.stdout.flush()
```

Ancak `sys.stdout` niteliği size aynı işi daha doğal bir şekilde yapma imkanı verir. Ayrıca `sys.stdout`'un başka mezzetleri de vardır.

Normal şartlar altında *print* deyimi yardımıyla yazdırdığımız şeyler doğrudan ekranda görünür. Yani:

```
>>> print "bir şey"
```

Bu komutu verdiğimizde *bir şey* çıktısı hemen ekranda görünecektir. Çünkü *print* komutu, çıktıları doğrudan ekrana verir. Ama eğer istersek biz *print* çıktılarını ekran dışında başka yerlere de yönlendirebiliriz. Mesela şu kodlara bir bakın:

```
>>> import sys

>>> f = open("log.txt", "w")

>>> sys.stdout = f
```

```
>>> print "Yeni bir mesajınız var!"
>>> f.close()
```

Burada şu işlemleri yaptık:

1. Öncelikle `sys` modülünü içe aktardık. Bunu neden yaptığımızı artık adınız gibi biliyorsunuz...
2. Daha sonra, mevcut çalışma dizini içinde `log.txt` adlı bir dosya oluşturduk ve bu dosyayı yazma kipinde ("w") açtık.
3. Sonraki satırda `f` adını verdiğimiz dosyamızı `sys.stdout` niteliğine atadık. "stdout" kelimesinin açılımı "standard output"tur. Bu ifade "standart çıktı konumu" anlamına gelir. Dolayısıyla `sys.stdout` niteliği Python'da `print` komutunun nereye çıktı vereceğini belirler. Normal şartlar altında, bildiğiniz gibi, `print` komutu doğrudan ekrana çıktı verir. Yani Python'un öntanımlı "standart çıktı konumu" bilgisayar ekranıdır. İşte biz bu `sys.stdout` niteliğinin gösterdiği değeri başka bir konuma atayarak standart çıktı konumunu değiştirebiliriz. Biz yukarıdaki örnekte standart çıktı konumu olarak `f` adını verdiğimiz dosyayı gösterdik.
4. Gördüğünüz gibi, bir önceki adımda yaptığımız işlemlerden ötürü artık `print` komutu ekrana çıktı vermiyor. Bu komutla yazdırdığımız şeyler doğruca `log.txt` dosyasına yazdırılıyor.
5. Son olarak, dosyada yaptığımız değişikliklerin etkili olabilmesi için dosyamızı kapatıyoruz.

Şimdi mevcut çalışma dizini altındaki `log.txt` adlı dosyayı açın ve içinde ne yazdığını bakın.

Dosyayı kapattıktan sonra artık `print` komutunu kullanamayız. Dosyayı kapattıktan sonra `print` komutunu kullanmaya çalışırsak Python bize şöyle bir hata mesajı verecektir:

```
>>> print "deneme"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file
```

Python'da kapattığınız bir dosya üzerinde yeni işlem yapamazsınız. Ya dosyayı yeniden açmanız, ya da gerekli bütün işlemleri dosyayı kapatmadan önce yapmanız gerekir...

Yalnız burada şöyle bir soru akla geliyor: Biz yukarıdaki komutlar yardımıyla standart çıktı konumunu değiştirdik. Dolayısıyla artık `print` komutu, çıktılarını ekrana değil dosyaya veriyor... Peki biz `print` komutunu eski haline nasıl döndüreceğiz? Bunun birkaç yolu vardır:

1. Etkileşimli kabuğu kapatıp yeniden açabilirsiniz. Etkileşimli kabuğu kapatıp yeniden açtığınızda her şey eski haline dönecektir.
2. Şu komutu kullanabilirsiniz:

```
>>> sys.stdout = sys.__stdout__
```

`sys.__stdout__` ifadesi, Python'un öntanımlı standart çıktı konumunu tutar. Dolayısıyla `sys.stdout` niteliğinin değerini `sys.__stdout__` yapacak olursanız `print` komutu eski haline dönecektir.

3. Python geliştiricilerinin, bu tür işlemler için tavsiye ettiği yol ise şudur:

```
>>> import sys
>>> orijinal_konum = sys.stdout
```

```
>>> f = open("log.txt", "w")
>>> sys.stdout = f
>>> print "deneme mesajı"
>>> f.close()
>>> sys.stdout = orijinal_konum
>>> print "deneme"

deneme
```

Burada `sys.stdout`'un değerini değiştirmeden önce, özgün çıktı konumunu `orijinal_konum` adlı bir değişkende saklıyoruz. Daha sonra normal bir şekilde çıktı konumu yönlendirme işlemlerini gerçekleştiriyoruz. İşimiz bitip dosyayı kapattıktan sonra da `sys.stdout` değerini eski haline döndürebilmek için bu niteliğin değerini `orijinal_konum` olarak değiştiriyoruz. Böylece `print` komutu eski işlevine yeniden kavuşmuş oluyor.

## 22.8 version\_info Niteliği

`sys` modülü konusunda inceleyeceğimiz son niteliğin adı `version_info`. Oldukça kolay bir nitelik bu. Bu niteliğin görevi kullanılan Python sürümü hakkında bilgi vermektir. Bunu şöyle kullanıyoruz:

```
>>> sys.version_info

(2, 6, 5, 'final', 0)
```

Gördüğünüz gibi, `version_info` niteliği çıktı olarak bir demet veriyor. Bu niteliği kullanarak, yazdığınız programlarda sürüm kontrolü yapabilirsiniz. Mesela:

```
>>> if not sys.version_info[:2] == (2, 6):
...     print ("Bu programı kullanabilmeniz için "
...           "Python'un 2.6 sürümü kurulu olmalı!")
```

Böylelikle `sys` modülünü tamamlamış olduk. En başta da söylediğimiz gibi, bu modül Python'daki en önemli modüllerden biridir. Dolayısıyla bu konu içinde anlatılanları sık sık tekrar etmenizi öneririm.

---

**Not:** Bu yazı Sayın Kürşat Örsel tarafından hazırlanmıştır. Python'daki ileri düzey modüllerden biri olan “`math`” modülü matematikle uğraşanların işlerini bir hayli kolaylaştıracak metotlar ve fonksiyonlar barındırır.

---

# math Modülü

Python'da matematiksel fonksiyonları math modülü ile kullanmaktayız. Şimdi math modülümüzün içeriğine bakalım. Unutmadan modülümüzü çalışmamıza çağıralım:

```
>>> import math
```

Bu komut ile modülümüzü çalışmamıza dahil etmiş olduk. Şimdi içerdiği fonksiyonları aşağıdaki komutu vererek görelim:

```
>>> dir(math)
```

```
[ '__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh',
  'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees',
  'e', 'exp', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum',
  'hypot', 'isinf', 'isnan', 'ldexp', 'log', 'log10', 'log1p', 'modf', 'pi',
  'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

Modülümüzün içeriğini de gördüğümüze göre şimdi kosinüs, sinüs, tanjant, pi, karekök, üslü ifadeler gibi fonksiyonlarla ilgili örneklerle kullanımını anlamaya çalışalım.

## 23.1 Üslü İfadeler Fonksiyonu (pow)

Üslü ifadeler matematikte hep karşımıza çıkmıştır, Python'da bunun için pow fonksiyonu kullanılmaktadır. pow power kelimesinin kısaltması olup Türkçe'de kuvvet, üs anlamlarına denk gelmektedir. Örnek verecek olursak  $2^3$  ifadesinin Python'daki karşılığı:

```
>>> math.pow(2, 3)
```

şeklinde dir. Yukarıdaki kodu yazdığımızda Python bize cevap olarak şunu gösterir:

```
8.0
```

pow ifadesinin kullanımında parentez içerisindeki ilk sayı tabanı, ikinci sayı ise üssü göstermektedir.



## 23.2 Pi Niteliği (pi)

pi sayısı hepimizin okul yıllarından bildiği üzere alan, hacim hesaplamalarında bolca kullanılır ve değeri genellikle 3 ya da 3,14 olarak alınır. Tabii ki esasen pi sayısı bu kadar kısa değildir. Hemen Python'a sorarak öğrenelim bu pi sayısının değerinin kaç olduğunu. Aşağıdaki komutu yazıyoruz:

```
>>> math.pi
```

Dediğimizde Python bize aşağıdaki çıktıyı gösterir:

```
>>> 3.1415926535897931
```

Demek ki gerçek pi sayısı biraz daha uzunmuş. Şimdi küçük bir hacim hesaplaması örneği ile konuyu pekiştirelim. Kürenin hacmini bulalım. Küre hacmi;

$$\text{Küre hacmi} = \frac{4}{3}(\pi \cdot r^3)$$

formülü ile bulunuyordu. Hemen kodlarımızı yazmaya başlayalım:

```
>>> 4.0 / 3.0 * math.pi * math.pow(2, 3)
```

Bu kod ile şunu demiş olduk: *4 ile 3'ü böl, pi sayısı ile çarp, sonra da sonucu 2'nin 3'üncü kuvveti ile çarp.* Python bize cevap olarak şunu gösterdi:

```
33.510321638291124
```

Tabii ki bu işlemleri kwrite programında bir dosya açarak aşağıdaki gibi de yazabiliriz:

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-

from __future__ import division
import math

r = input("Kürenin yarıçapını giriniz:")
hacim = 4 / 3 * math.pi * math.pow(r, 3)
```

Böylece kürenin hacmini veren küçük bir programımız oldu.

## 23.3 Karekök Fonksiyonu (sqrt)

Karekök ile ilgili fonksiyonumuz sqrt. Kullanımı ise aşağıdaki gibidir:

```
>>> math.sqrt(9)
```

Kodunu yazıp enter'e bastığımızda Python bize aşağıdaki sonucu gösterir:

```
3.0
```

## 23.4 Euler Sabiti (e)

Bu nitelik, matematikteki euler sabitini veriyor. Kullanımı ise aşağıdaki gibi:

```
>>> math.e
```

Yukarıdaki kodu yazıp enter'e bastığımızda karşımıza euler sabiti 2.7182818284590451 cevap olarak Python tarafından gösteriliyor. Kullanım ve benzerlik açısından aynı pi sayısı gibi:

```
2.7182818284590451
```

## 23.5 exp Fonksiyonu

exp fonksiyonunun kullanımı şu şekilde:

```
>>> math.exp(x)
```

Bu kodu küçük bir örnekle açıklamak daha kolay olacak.

math.exp(2) kodunu yazdığımızda Python aşağıdaki cevabı verir:

```
7.3890560989306504
```

Bu sayı da nereden çıktı diyorsanız, exp fonksiyonu yukarıda bahsettiğimiz euler sabitinin kuvvetini alan bir fonksiyon.  $\exp(x)$  ifadesindeki  $x$  parametresi bizim kuvvetimizdir. Yani  $\exp(2)$  dediğimizde esasen biz Python'a şunu demiş oluyoruz:  $(2.7182818284590451)^2$  Yani euler sabitinin karesini almış olduk.

## 23.6 Logaritma (log) Fonksiyonu

Logaritma fonksiyonumuzun kullanımı şu şekilde;  $\log(x, y)$ . Burada  $x$  sayısı logaritması alınacak sayı,  $y$  sayısı ise taban sayısını temsil etmektedir. Bir örnek ile pekiştirelim:

```
>>> math.log(2, 2)
```

Python bize aşağıdaki cevabı verir:

```
1.0
```

## 23.7 Logaritma (log10) Fonksiyonu

Bu fonksiyonun yukarıdakinden tek farkı taban sayısının önceden belirlenmiş ve 10 olması. Bu yüzden fonksiyonun kullanımı şöyle;  $\log_{10}(x)$  Burada  $x$  onluk tabana göre logaritması alınacak sayıdır:

```
>>> math.log10(10)
```

Dönen cevap:

```
1.0
```

## 23.8 Degrees Fonksiyonu

Degrees fonksiyonu girilen açıyı radyan'dan dereceye çevirmeye yarar. Kullanımı şu şekilde;

```
>>> math.degrees(x)
```

x burada derece'ye çevrilecek radyandır. Örnek olarak 45 radyanlık bir açı verelim ve derece karşılığını bulalım:

```
>>> math.degrees(45)
```

```
2578.3100780887044
```

## 23.9 Radians Fonksiyonu

Radians fonksiyonu girilen açıyı derece'den radyan'a çevirmeye yarar. Kullanımı şu şekilde;

```
>>> math.radians(x)
```

x burada radyana çevrilecek açıdır. Örnek olarak 45 derecelik bir açı verelim ve radyan karşılığını bulalım:

```
>>> math.radians(45)
```

```
0.78539816339744828
```

Kosinüs, Sinüs, Tanjant fonksiyonlarına girmeden önce belirtmem gereken önemli bir nokta bulunmaktadır. Bu fonksiyonlarda açı olarak Python radyan kullanmaktadır. Bu yüzden aldığımız sonuçlar okulda öğrendiğimiz değerlerden farklı olacaktır. Bunuda radians fonksiyonu ile düzelteceğiz.

## 23.10 Kosinüs Fonksiyonu (cos)

Hemen kosinüs fonksiyonu ile bir örnek yapalım:

```
>>> math.cos(0)
```

Python bunun sonucu olarak bize:

```
1.0
```

cevabını verir.

```
>>> math.cos(45)
```

```
0.52532198881772973
```

Evet yukarıda gördüğümüz üzere bizim beklediğimiz cevap bu değil. Biz 0.7071 gibi bir değer bekliyorduk. Bunu aşağıdaki şekilde düzeltebiliriz:

```
>>> math.cos(math.radians(45))
```

```
0.70710678118654757
```

Şimdi istediğimiz cevabı aldık.

## 23.11 Sinüs Fonksiyonu (sin)

Hemen sinüs fonksiyonu ile bir örnek yapalım:

```
math.sin(0)
```

Python bunun sonucu olarak bize:

```
0.0
```

cevabını verir.

```
>>> math.sin(45)
0.85090352453411844
```

Evet yukarıda gördüğümüz üzere bizim beklediğimiz cevap bu değil. Biz 0.7071 gibi bir değer bekliyorduk. Bunu aşağıdaki şekilde düzeltebiliriz:

```
>>> math.sin(math.radians(45))
0.70710678118654746
```

Şimdi istediğimiz cevabı aldık.

## 23.12 Tanjant (tan) Fonksiyonu

Tanjant fonksiyonu ile bir örnek yapalım:

```
>>> math.tan(0)
```

Python bunun sonucu olarak bize:

```
0.0
```

cevabını verir.

```
>>> math.tan(45)
1.6197751905438615
```

Evet yukarıda gördüğümüz üzere bizim beklediğimiz cevap bu değil. Biz ~1.000(yaklaşık) gibi bir değer bekliyorduk. Bunu aşağıdaki şekilde düzeltebiliriz:

```
>>> math.tan(math.radians(45))
0.99999999999999989
```

Buda yaklaşık olarak 1.000 yapar.

Şimdi istediğimiz cevabı aldık.

Yukarıda verdiğimiz fonksiyonlardan bazılarını kullanarak basit bir fizik sorusu çözelim.

3 newton ve 5 newton büyüklüğünde olan ve aralarındaki açı 60 derece olan iki kuvvetin bileşkesini bulalım. Formülümüz aşağıdaki gibidir.

$$R^2 = F_1^2 + F_2^2 + 2 * F_1 * F_2 * \cos(\alpha)$$

Boş bir kwrite belgesi açarak içine kodlarımızı yazmaya başlayalım:

```
#!/usr/bin/env python
#coding:utf-8 -*-

import math #math modülünü çağırdık.

a = 60 # Açımızı a değişkenine atadık.
f1 = 3 # Birinci kuvveti f1 değişkenine atadık.
f2 = 4 # ikinci kuvveti f2 değişkenine atadık.

#Şimdi formülümüzü yazalım:
R = math.sqrt(math.pow(f1, 2) + math.pow(f2, 2) + 2 * f1 * f2 * math.cos(math.radians(a)))

#...ve ekrana çıktı verelim:
print "Bileşke kuvvet:", R
```

Kodlarımızı çalıştırdığımızda sonuç:

```
6.0827625303
```

Tabii ki daha farklı uygulamalar da yapılabilir.

## 23.13 Faktoriyel (factorial)

Faktoriyel işlemi için Python 2.6 sürümü ile math modülü içerisinde bir fonksiyon oluşturulmuş. Bundan önceki 2.5 sürümünde böyle bir fonksiyon yoktu. Kullanımı ise şu şekilde:

```
math.factorial(x)
```

x sayısı burada bizim faktoriyelini alacağımız sayıyı temsil ediyor. Bir örnek ile inceleyelim. Lise matematik derslerinde faktoriyel şu şekilde hesaplanıyordu.  
 $n! = n \times (n-1) \times (n-2) \times (n-3) \times \dots \times 1$

Modülümüzü içe aktaralım:

```
>>> import math
```

Şimdi;

```
>>> math.factorial(5)
```

dediğimizde Python bize

```
120
```

sonucunu veriyor. Kendimiz hesaplayacak olursak:

```
5!=5x4x3x2x1=120
```

sonucuna ulaşırız. Bu fonksiyon ile Python bizi uzun işlemlerden ve ekstra tanımlamalardan kurtarıyor.

## 23.14 fmod Fonksiyonu

Bu fonksiyon matematikteki modüler aritmetik hesaplamaları için oluşturulmuş. İlk önce kısaca mod işlemleri nasıl yapılıyor ona değinelim.

Ahmet bey ile Fatma hanım; saat 09 da, 7 saat sonra buluşmak üzere anlaşmış ayrılıyorlar. Saat 12 bölmeli olduğu için, Ahmet bey ile Fatma hanım buluştuğunda saat 4' ü gösterir. Bunu modüler aritmetik ile şöyle buluruz:

$$9+7=16$$

$$\begin{array}{r|l} 16 & 12 \\ -12 & \\ \hline & 4 \end{array}$$

$$16 = 4 \pmod{12}$$

Burada 12 sayısı bizim modumuzdur. Yapılan toplama işlemi, tam sayılardaki toplama işleminden farklıdır. Bu tip işlemler Modüler Aritmetik'in konusudur.

Python bu uzun işlemi bizim için fmod fonksiyonu ile tanımlamış, işleri kolaylaştırmış. Birkaç örnek ile fmod fonksiyonun kullanımına bakalım:

```
>>> fmod(x, y)
```

şeklinde bir kullanımı var. Burada x modu alınacak sayıyı, y ise modu temsil eder:

```
>>> math.fmod(639, 5)
```

komutunu verdiğimizde, Python bize;

```
4.0
```

cevabını döndürür.

```
>>> math.fmod(99,3)
```

```
0.0
```

## 23.15 ceil Fonksiyonu

Ceil kelimesinin Türkçe karşılığı tavan çekmek, tavan inşa etmek anlamlarına gelir. Anlaşılacağı üzere biz yukarıya yuvarlama işlemi anlamında kullanacağız. Fonksiyon matematikte kullanılan yuvarlama işlemi ile ilgili fonksiyonlardan biridir. Ceil fonksiyonu verilen ondalıklı sayıdan büyük, en küçük tam sayıyı döndürür. Kullanımı şöyle;

```
>>> math.ceil(x.y)
```

Burada x.y ondalıklı sayıyı temsil etmektedir. Örneğin 7.3 gibi.

```
>>> math.ceil(19.3)
```

```
20.0
```

cevabını döndürür.

## 23.16 floor Fonksiyonu

Diğer yuvarlama işlemi fonksiyonu floor. Floor kelimesi zemin, yer, taban anlamına geliyor. Bizim için taban anlamı daha manalı. Bu fonksiyon ile ondalıklı sayıları en yakın tabana yuvarlama işlemi gerçekleştiriliyor. Hemen bir örnekle inceleyelim.

```
>>> math.floor(3.5)
```

dediğimizde Python bize aşağıdaki sonucu verir.

```
3.0
```

Böyle Python 3.5 sayısını en yakın taban olan 3.0' a yuvarladı.

## 23.17 fabs Fonksiyonu

Bu fonksiyon verilen sayının mutlak değerini verir. Kullanımı ise aşağıdaki gibidir:

```
>>> math.fabs(-3)
```

```
3.0
```

Python bize sonuç olarak 3.0 değerini verdi. Tabi burada amaç matematik dersi anlatmak değil. Bir örnek daha verelim:

```
>>> math.fabs(9)
```

```
9.0
```

## 23.18 frexp Fonksiyonu

Bu fonksiyon verilen değerin mantisini ve karakteristiğini veriyor. Fonksiyonun kullanımı şu şekilde:

```
>>> math.frexp(x)
```

Bu durumda Python bize (m, k) gibi bir sonuç gönderiyor. m mantis, k ise karakteristik.

Bir örnek ile inceleyelim:

```
>>> math.frexp(7)
```

```
(0.875, 3)
```

## 23.19 ldexp Fonksiyonu

Bu fonksiyon kısaca anlatmak gerekirse frexp fonksiyonunun tersi ya da zıttı diyebiliriz. Yukarıda `math.frexp(7)` komutunu kullanarak (0.875, 3) sonucunu elde ediyoruz. Şimdi bu sonuçtan elde ettiğimiz rakamları kullanarak 7 sayısına ldexp fonksiyonunu kullanarak ulaşalım:

```
>>> math.ldexp(0.875, 3)
```

```
7.0
```

Bu şekilde ldexp fonksiyonunu kullanarak değerimize yeniden ulaşabiliyoruz.

## 23.20 modf Fonksiyonu

Bu fonksiyon verilen sayının ondalıklı kısmı ile tam kısmını parçalayarak size geri döndürüyor. Kullanımı modf(x) şeklinde. x burada tam sayı ve ondalıklı sayı olabilir. Hemen bir örnek ile açıklayalım:

```
>>> math.modf(6.5)
```

```
(0.5, 6.0)
```

## 23.21 trunc Fonksiyonu

Bu fonksiyon verilen değerleri tam sayı olarak kırıyor. Kırıyor herhalde uygun kelime diye düşünüyorum ya da kesiyor diyebiliriz. Peki neyi kesiyor? Verilen ondalıklı veya tam sayıların sadece tam kısımlarını bize döndürüyor. Ondalık kısmı kesiyor(kırıyor). Yukarı ya da aşağı yuvarlıyor diyemiyoruz. Bu da - ve + operatörü kullanılmış örnek işlemlerde ortaya çıkıyor. Şimdi örneklerle bakalım:

```
>>> math.trunc(15.4)
```

```
15
```

```
>>> math.trunc(-15.4)
```

```
-15
```

Bu iki örnekten de anlaşılacağı üzere birincisinde aşağıya, ikincisinde (- operatörden dolayı) yukarı yuvarladı olarak algılanabilir. Fakat bu fonksiyonun yaptığı işlem sadece ondalıklı kısmı kesmek. Bu fonksiyon Python 2.6 sürümü ile eklenmiş.

## 23.22 hypot Fonksiyonu

Öklit bağıntısı için oluşturulmuş bir fonksiyondur. (Pisagor teoremi diye geçer.) Gerçi işin Öklit-Pisagor kısmına değinmeyelim, orası matematik uzmanlarının işi. Fonksiyonun kullanımı;

```
>>> math.hypot(x, y)
```

şeklinde. Öklit bağıntısı  $a^2 = b^2 + c^2$  şeklindedir. Hani bir zamanlar lise matematik ve fizik derslerinde üçgenlerin kenar uzunluklarını, vektörlerin birim uzunluklarını bulmak için çokça kullandığımız bir bağıntı. Hatta ezberlerdik 3-4-5 üçgeni, 5-12-13 üçgeni gibi. Neyse bu kadar anı yeter. İşte hypot fonksiyonu işimizi kolaylaştırarak bu bağıntının sonucunu bize döndürüyor. Verilen değerlerin karesini al, topla, karekökünü bul gibi sıkıntılardan kurtarıyor. Hemen yukarıdaki üçgenlerde deneyerek örnekleyelim:



```
>>> math.hypot(3, 4)
```

```
5.0
```

ya da:

```
>>> math.hypot(5, 12)
```

```
13.0
```

## 23.23 Hiperbolik Fonksiyonlar

### 23.23.1 cosh Fonksiyonu

Hiperbolik kosinüs fonksiyonunu döndürür. Kullanımı:

```
>>> math.cosh(x)
```

şeklindedir. Burada x girilen değerdir. Bir örnek ile inceleyelim:

```
>>> math.cosh(0)
```

```
1.0
```

### 23.23.2 sinh Fonksiyonu

Hiperbolik sinüs fonksiyonunu döndürür. Kullanımı:

```
>>> math.sinh(x)
```

şeklindedir. Burada x girilen değerdir. Bir örnek ile inceleyelim:

```
>>> math.sinh(0)
```

```
0.0
```

### 23.23.3 tanh Fonksiyonu

Hiperbolik tanjant fonksiyonunu döndürür. Kullanımı:

```
>>> math.tanh(x)
```

şeklindedir. Burada x girilen değerdir. Bir örnek ile inceleyelim:

```
>>> math.tanh(90)
```

```
1.0
```

### 23.23.4 acosh Fonksiyonu

Hiperbolik kosinüs fonksiyonunun tersini döndürür. Kullanımı:

```
>>> math.acosh(x)
```

şeklindedir. Burada x girilen değerdir. Bir örnek ile inceleyelim:

```
>>> math.acosh(1)
```

```
0.0
```

Bu fonksiyon Python 2.6 sürümü ile eklenmiş.

### 23.23.5 asinh Fonksiyonu

Hiperbolik sinüs fonksiyonunun tersini döndürür. Kullanımı:

```
>>> math.asinh(x)
```

şeklindedir. Burada x girilen değerdir. Bir örnek ile inceleyelim:

```
>>> math.asinh(0)
```

```
0.0
```

Bu fonksiyon Python 2.6 sürümü ile eklenmiş.

### 23.23.6 atanh Fonksiyonu

Hiperbolik tanjant fonksiyonunun tersini döndürür. Kullanımı:

```
>>> math.atanh(x)
```

şeklindedir. Burada x girilen değerdir. Bir örnek ile inceleyelim:

```
>>> math.atanh(0)
```

```
0.0
```

Bu fonksiyon da Python 2.6 sürümü ile eklenmiş.

---

# datetime Modülü

---

Python'da *datetime* adlı bir modül bulunur. Bu modül bize tarih ve zamanla ilgili işlemleri gerçekleştirme imkanı sağlar. Bu modül Python'un standart kütüphanesine dahil olduğundan, yani Python geliştiricilerince yazıldığından, bu modülü kullanmak için herhangi bir harici program kurmamıza gerek yok. Yazdığımız bir programda *datetime* modülüne ihtiyaç duyarsak, modülü normal bir şekilde içe aktarmamız yeterli olacaktır:

```
>>> import datetime
```

Dilerseniz ilk iş olarak bu modülün içeriğinde neler olup olmadığına bir bakalım:

```
>>> dir(datetime)

['MAXYEAR', 'MINYEAR', '__doc__', '__name__', '__package__',
'date', 'datetime', 'datetime_CAPI', 'time', 'timedelta', 'tzinfo']
```

Biz bu derste, yukarıda görünen nitelik ve fonksiyonları isimlerine göre değil, işlevlerine göre inceleyeceğiz.

## 24.1 Bugünün Tarihini Bulmak

*datetime* adlı modülü kullanarak bugünün tarihini bulabiliriz. Bunun için şöyle bir kod yazıyoruz:

```
>>> bugun = datetime.date.today()
```

Burada, *datetime* modülünün *date* adlı sınıfı içinde yer alan *today()* adlı bir fonksiyonu çağırdık. Bu fonksiyon bize bugünün tarihini verir:

```
>>> print bugun
```

```
2010-07-08
```

Bu çıktıda görünen tarih “yıl-ay-gün” şeklindedir. Yani yukarıdaki karakter dizisi 8 Temmuz 2010 tarihini gösteriyor.

Yukarıdaki kodu kullanarak, yıl, ay ve gün bilgilerini tek tek ayıklayabilirsiniz:

```
>>> print "%s yılındayız!" %bugun.year
2010 yılındayız!

>>> print "%s yılının %s. ayındayız!" %(bugun.year, bugun.month)
2010 yılının 7. ayındayız!

>>> print "%s. ayın %s. günündeyiz!" %(bugun.month, bugun.day)
7. ayın 8. günündeyiz!
```

Gördüğünüz gibi `bugun.year` yılı, `bugun.month` ayı, `bugun.day` ise günü gösteriyor.

## 24.2 Bir Tarihin Hangi Güne Geldiğini Bulmak

Bir önceki bölümde `datetime` modülünü kullanarak bugünün tarihiyle ilgili bazı bilgiler edinebildik. Bu bölümde ise bir tarihin haftanın hangi gününe denk geldiğini bulmaya çalışacağız. Şimdi şu kodlara dikkatlice bakalım:

```
>>> bugun = datetime.date.today()
```

Böylece bugünün tarihin aldık. Şimdi bu tarihin hangi güne denk geldiğini bulacağız:

```
>>> print bugun.weekday()
3
```

Muhtemelen çıktının bu şekilde olmasını beklemiyordunuz. Python haftanın günlerini gösterirken sayılardan yararlanır. Buna göre şöyle bir tablo çizebiliriz:

Sayı	Gün
0	Pazartesi
1	Salı
2	Çarşamba
3	Perşembe
4	Cuma
5	Cumartesi
6	Pazar

Buna göre yukarıdaki çıktıda gördüğümüz 3 sayısı Perşembe gününe işaret ediyor. Eğer Python'ın günleri sayıyla göstermesi hoşunuza gitmediyse, sözlüklerden yararlanarak şöyle bir şey yazabilirsiniz:

```
#!/usr/bin/env python
#-*- coding: utf-8

import datetime

hafta = {0: "Pazartesi",
         1: "Salı",
         2: "Çarşamba",
         3: "Perşembe",
         4: "Cuma",
         5: "Cumartesi",
         6: "Pazar"}
```

```
bugun = datetime.date.today()

print "Bugün %s" %(hafta[bugun.weekday()])
```

Yukarıda yaptığımız işlemler hep bugünün tarihi ile ilgiliydi. Ancak tabii istersek farklı tarihler üzerinde de işlem yapabiliriz. Bunun için öncelikle üzerinde işlem yapacağımız tarihi belirtmemiz gerekiyor. Bunu şu şekilde yapıyoruz:

```
>>> tarih = datetime.date(2010, 5, 30)
```

Burada, 30 Mayıs 2010 tarihini belirttik. Bunu nasıl yaptığımıza çok dikkat edin. *datetime.date* adlı sınıfı kullanarak, ilgili tarihi yıl-ay-gün şeklinde parantez içinde yazıyoruz. Böylece şu çıktıyı elde edebiliyoruz:

```
>>> print tarih

2010-05-30
```

Mesela bu tarihin hangi güne denk geldiğini bulmak istersek *weekday()* adlı fonksiyondan yararlanabileceğimizi biliyorsunuz:

```
>>> print tarih.weekday()

6
```

Demek ki 30 Mayıs 2010 tarihi Pazar gününe denk geliyormuş... Tabii isterseniz yukarıda yazdığımız sözlükten yararlanarak daha net bir çıktı alabilirsiniz:

```
#!/usr/bin/env python
#-*- coding: utf-8

import datetime

hafta = {0: "Pazartesi",
         1: "Salı",
         2: "Çarşamba",
         3: "Perşembe",
         4: "Cuma",
         5: "Cumartesi",
         6: "Pazar"}

tarih = datetime.date(2010, 5, 30)

print ("%s tarihi %s gününe denk gelir"
      %(tarih, hafta[tarih.weekday()]))
```

Daha önce öğrendiğimiz *month*, *day* ve *year* nitelikleri burada da geçerlidir:

```
>>> print tarih.month #ay

5

>>> print tarih.day #gün

30

>>> print tarih.year #yıl
```

## 24.3 Tarihleri Biçimlendirmek

Yukarıda öğrendiğimiz bilgilere göre bugünün tarihini elde edebilmek için şöyle bir kod yazabiliyoruz:

```
>>> bugun = datetime.date.today()
```

Bu kod bize bugünün tarihinin ne olduğunu gösterir:

```
>>> print bugun
```

```
2010-07-08
```

Ancak bu çıktı bir karakter dizisi olmadığı için, bu çıktıyı istediğimiz gibi biçimlendirmek mümkün değildir. Yani mesela şöyle bir şey yazamayız:

```
>>> print bugun[:4]
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'datetime.date' object is unsubscriptable
```

Dediğimiz gibi, yukarıdaki çıktı bir karakter dizisi değildir. İsterseniz bu durumu hemen teyit edelim:

```
>>> type(bugun)
```

```
<type 'datetime.date'>
```

Gördüğünüz gibi elimizdeki şey bir “datetime.date” nesnesi... Eğer biz tarih çıktılarını karakter dizisi olarak almak istersek *strftime()* adlı özel bir fonksiyondan yararlanabiliriz. Bunu şöyle kullanıyoruz:

```
>>> bugun = datetime.date.today()
```

```
>>> print bugun.strftime("%c")
```

```
08.07.2010 00:00:00
```

*strftime()* fonksiyonu son derece yararlı bir araçtır. Bu aracı kullanarak tarih bilgilerini istediğimiz şekilde biçimlendirebiliriz. Bu işi yaparken bir takım işaretlerden/karakterlerden yararlanmamız gerekiyor. Biz yukarıdaki örnekte “%c” adlı karakteri kullandık. Dilerseniz bu karakterlerin en önemlilerini listeleyelim:

Karak- ter	Anlamı
%a	hafta gününün kısaltılmış adı
%A	hafta gününün tam adı
%b	ayın kısaltılmış adı
%B	ayın tam adı
%c	tam tarih
%d	tamsayı cinsinden gün
%j	belli bir tarihin, yılın kaçınıcı gününe denk geldiğini gösteren 1-366 arası bir sayı
%m	tamsayı cinsinden ay
%U	belli bir tarihin yılın kaçınıcı haftasına geldiğini gösteren 0-53 arası bir sayı
%y	yılın son iki rakamı
%Y	tamsayı cinsinden yıl

İsterseniz bu tabloya bakarak birkaç örnek de verelim:

```
>>> print "Bugün günlerden %s" %bugun.strftime("%A")
```

```
Bugün günlerden Perşembe
```

```
>>> print bugun.strftime("%a")
```

```
Per
```

Dilerseniz bütün karakter dizisini doğrudan *strftime()* fonksiyonu içine de yazabilirsiniz:

```
>>> print bugun.strftime("Aylardan %B")
```

```
Aylardan Temmuz
```

Dediğimiz gibi, *strftime()* fonksiyonunun dönüş değeri bir karakter dizisidir. Dolayısıyla bu karakter dizisini istediğiniz gibi biçimlendirebilirsiniz:

```
>>> tarih = datetime.date(2005, 4, 3)
```

```
>>> krk = tarih.strftime("%c")
```

```
>>> print krk
```

```
03.04.2005 00:00:00
```

```
>>> print type(krk)
```

```
<type 'str'>
```

```
>>> print krk[:2]
```

```
03
```

*strftime()* fonksiyonunun önemli bir özelliği vardır. Bu fonksiyon sisteminizde tanımlı olan dil yereline (locale) göre işlem yapar. Yani eğer sisteminizin dil yereli İngilizce ise ay isimleri de tabii ki İngilizce olacaktır.

*strftime()* fonksiyonunun genellikle şöyle kullanıldığına tanık olursunuz:

```
>>> bugun = datetime.date.today()
>>> print bugun.strftime("Son güncelleme: %d/%m/%y")
```

## 24.4 Tarihlerle Aritmetik İşlem Yapmak

Eğer yazdığınız bir programda tarihlere dayalı bir işlem yapıyorsanız, mevcut tarihin öncesini ve sonrasını da bilmeniz gerekebilir. Mesela TCMB’den döviz kuru bilgilerini alan bir program yazıyorsunuz diyelim. En basitinden, mesela doların bir gün öncesine göre yükselip yükselmediğini kullanıcıya gösterebilmek için, bugünün tarihini bilmenin yanısıra bir önceki günün tarihini de bilmeniz gerekir. Hatta örneğin eğer dolar kurundaki haftalık değişimi grafiklerle gösteren bir uygulama yazacaksanız mevcut günün 7 gün öncesine kadar giden tarihleri de alabilmeniz gerekir. Gelin isterseniz bu tür işlemleri nasıl yapabileceğimizi gösteren basit bir örnek verelim:

```
>>> import datetime
>>> bugun = datetime.date.today()
```

Böylece bugünün tarihini bulduk. Diyelim ki bugünün tarihi 09.07.2010 Cuma olsun. Siz bu tarihin 1 gün öncesini bulmak istiyorsunuz:

```
>>> fark = datetime.timedelta(1)
>>> dun = bugun - fark
>>> print dun
```

```
2010-07-08
```

Burada *datetime* modülü içinde bulunan *timedelta()* adlı özel bir fonksiyondan yararlandık. Bu fonksiyona verdiğimiz sayı değerli argüman, kaç günlük bir fark üzerinden işlem yapacağımızı gösteriyor. Bizim amacımız 1 günlük fark üzerinden işlem yapmak olduğu için *timedelta()* fonksiyonuna argüman olarak 1 sayısını verdik.

Daha sonra 1 gün öncesinin tarihi bulabilmek için bugünün tarihinden, fark değişkeninin değerini (yani 1’i) çıkarıyoruz. Böylece bugünün 1 gün öncesi olan 8 Temmuz 2010 tarihini elde ediyoruz. Eğer bir önceki günün haftanın hangi gününe denk geldiğini bulmak istiyorsak şöyle bir şey yazabileceğimizi biliyorsunuz:

```
>>> print dun.weekday()
```

```
3
```

Demek ki 8 Temmuz 2010 tarihi Perşembeye denk geliyormuş...

Dilerseniz *strftime()* fonksiyonunu kullanarak doğrudan gün adını da alabilirsiniz:

```
>>> print dun.strftime("%A")
```

```
Perşembe
```

Bu *strftime()* fonksiyonunun oldukça geniş bir kullanım alanı vardır. Mesela [istihza.com](http://istihza.com)’daki sayfaların en sonuna eklenen “Son güncelleme” tarihini oluşturmak için ben de bu fonksiyon kullanılıyor. Bunun için kullanılan karakter dizisi şöyle:

```
bugun.strftime('%d/%m/%Y')
```



# time Modülü

*time* modülü, Python'da saat/zaman ile ilgili işlemler yapmamızı sağlar. Bu modül işlev olarak az çok *datetime()* modülüyle benzerlik gösterir.

Python'daki bu *time* adlı modül, tıpkı öteki modüller gibi içinde birtakım fonksiyonlar barındırır. Dilerseniz bu fonksiyonlar içinde en çok kullanılanı inceleyerek başlayalım konumuza...

## 25.1 sleep() Fonksiyonu

Dediğimiz gibi, bu fonksiyon, *time()* modülü içinde en sık kullanılan fonksiyondur. *sleep()* fonksiyonu, yazdığımız bir programın işleyişini belli bir süre durdurmamızı sağlar. İsterseniz buna basit bir örnek verelim:

```
>>> import time
>>> for i in range(10):
...     time.sleep(1)
...     print i
```

Gördüğümüz gibi, 0'dan 10'a kadar olan sayılar 1'er saniye aralıklarla ekrana dökülüyor. Programın kaç saniye duracağını, *sleep()* fonksiyonuna verdiğimiz bir parametre ile belirliyoruz. Biz yukarıdaki örnekte bu parametreyi "1" olarak belirledik. Yani programımızın 1'er saniye aralıklarla çalışmasını istedik...

Gelin isterseniz yukarıdaki bilgiyi kullanarak grafik arayüze sahip bir sayaç yazalım:

```
from Tkinter import *
import time

pencere = Tk()
pencere.geometry("200x200")

etk = Label(font = "Helvetica 60 bold",
            fg = "steelblue")

etk.pack(pady=50)

a = -1
```

```
while True:
    a += 1
    time.sleep(1)
    pencere.update()
    etk["text"] = a

pencere.mainloop()
```

Yalnız dikkat ettiyseniz bu sayaç programı çalışma sırasında bloke oluyor, yani takılıyor. Program penceresini fare ile sürüklerken bu takılmaları daha iyi görebiliyoruz. Üstelik programı çarpı düğmesinden kapatmaya çalıştığımızda da programda belli bir kasılma göze çarpıyor. Ayrıca program kapanırken hata da veriyor. Bunu engellemek için programınızı çok katmanlı (*multi-threaded*) bir hale getirebilirsiniz:

```
# -*- coding: utf-8 -*-

from Tkinter import *
import time
import threading

class GUI(object):
    def __init__(self):
        self.pencere_olustur()

    def pencere_olustur(self):
        """Pencere araçlarımızı burada oluşturuyoruz.
        Programımız bir etiket ve düğmeden ibaret..."""

        self.etk = Label(font = "Helvetica 60 bold",
                          fg = "steelblue")

        self.etk.pack(pady=50)

        self.btn = Button(text = "KAPAT",
                           relief = "raised",
                           font = "Helvetica 15 bold",
                           fg = "navy")

        self.btn["command"] = self.kapat
        self.btn.pack()

        #Kullanıcı çarpı düğmesine bastığında kapat
        #fonksiyonu devreye girsin. Eğer bunu yapmazsak
        #program yine de kapanacak, ama kapanırken
        #hata verecektir...
        pencere.protocol("WM_DELETE_WINDOW", self.kapat)

    def kapat(self):
        """Programın kapanmasını sağlayan fonksiyon.
        Burada programı tamamen kapatmadan önce
        pencereyi "withdraw" durumuna alıyoruz.
        Böylece sayacın durmasını sağlıyoruz ve programın
        aniden kapanıp hata vermesine engel oluyoruz"""

        pencere.state("withdraw")

        #sayacın düzgün bir şekilde durabilmesi için
        #programımıza 1 saniyelik bir süre tanıyoruz.
```

```

        time.sleep(1)

        #Pencere "withdraw" durumuna alındı ve sayacımız
        #durdu. Artık programımızı sona erdirebiliriz.
        pencere.destroy()

pencere = Tk()
pencere.geometry("350x300")
app = GUI()

class Sayac(threading.Thread):
    def run(self):
        sayi = -1
        #pencere durumu "withdrawn" olmadığı sürece
        #sayacımız saymaya devam edecek.
        while pencere.state() != "withdrawn":
            sayi += 1
            time.sleep(1)
            app.etk["text"] = sayi

syc = Sayac()
syc.start()

pencere.mainloop()

```

## 25.2 strftime() Fonksiyonu

*strftime()* fonksiyonu yardımıyla gün, yıl, ay, hafta, saat, dakika, saniye gibi bilgileri gösterebiliriz. Örneğin:

```

>>> import time
>>> time.strftime("%Y")

'2010'

```

Gördüğümüz gibi, *strftime()* fonksiyonunu bir parametre ile birlikte kullanıyoruz. Bu örnekteki "%Y" ifadesi yılı göstermemizi sağlıyor. Eğer yılın son iki hanesini görmek istersek şöyle yazabiliriz:

```

>>> time.strftime("%y")

'10'

```

Ayı görmek için:

```

>>> time.strftime("%m")

'08'

```

Günü görmek için:

```

>>> time.strftime("%d")

'16'

```

Ay, gün ve yılı birlikte görmek için:

```
>>> time.strftime("%X")
```

```
'08/16/2010'
```

Tabii buradaki gösterim *ay/gün/yıl* şeklinde. Türkiye'deki tarih gösterimi ise *gün-ay-yıl* şeklindedir. Türkiye'ye uygun bir tarih gösterimi elde etmek için iki yol takip edebilirsiniz. Birincisi:

```
>>> time.strftime("%d.%m.%Y")
```

```
'16.08.2010'
```

Burada gün, ay ve yıl bileşenlerini uygun sırada ve tek tek belirttik.

İkincisi:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, "tr_TR.UTF-8")
>>> time.strftime("%X")
```

```
'16-08-2010'
```

Burada ise *locale* modülünden yararlanarak Türkçe yerelleri etkin hale getirdik. Yukarıdaki kullanım GNU/Linux içindir. Windows kullanıcıları yukarıdaki kodları şu şekilde yazabilir:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, "Turkish_Turkey.1254")
>>> time.strftime("%X")
```

```
'16.08.2010'
```

Eğer *strftime()* modülünü kullanarak saat, dakika ve saniye bilgilerini elde etmek isterseniz şöyle bir şey yazabilirsiniz:

```
>>> time.strftime("%X")
```

```
'2:08:02 PM'
```

Bu da, tıpkı tarih gösteriminde olduğu gibi, Amerikan sistemine uygun bir saat gösterimidir. Bunu Türkiye sistemine uygun bir hale getirmek için yine *locale* modülünden yararlanacağız:

```
>>> locale.setlocale(locale.LC_ALL, "tr_TR.UTF-8")
>>> time.strftime("%X")
```

```
'14:08:02'
```

Gördüğünüz gibi, Amerikan sisteminde 12 saat üzerinden işlem yapılırken, Türkiye sisteminde 24 saat üzerinden işlem yapılıyor.

Burada da, eğer isterseniz saat, dakika ve saniye bilgilerini tek tek alabilirsiniz.

Saati almak için:

```
>>> time.strftime("%H")
```

```
'14'
```

Dakikayı almak için:

```
>>> time.strftime("%M")
```

```
'13'
```

Saniyeyi almak için:

```
>>> time.strftime("%S")
```

```
'53'
```

Elbette bunları birleştirebilirsiniz de:

```
>>> time.strftime("%H:%M:%S")
```

```
'14:13:53'
```

Bunların dışında, *strftime()* fonksiyonu ile birlikte kullanılan başka harfler de bulunur. Bu harfleri şöylece özetleyebiliriz:

İfade	Anlamı
%a	Kısaltılmış gün adı
%A	Tam gün adı
%b	Kısaltılmış ay adı
%B	Tam ay adı
%c	Tam tarih ve saat
%d	Ondalık sayı cinsinden gün (aya göre)
%H	Ondalık sayı cinsinden saat (24 saat hesabına göre)
%I	Ondalık sayı cinsinden saat (12 saat hesabına göre)
%j	Ondalık sayı cinsinden gün (yıla göre)
%m	Ondalık sayı cinsinden ay
%M	Ondalık sayı cinsinden dakika
%S	Ondalık sayı cinsinden saniye
%U	Yıla göre hafta numarası. Pazar haftanın ilk günü olarak alınır
%x	Tam tarih
%X	Tam saat
%y	Yılın son iki hanesi
%Y	Tam yıl gösterimi

## 25.3 localtime() Fonksiyonu

*localtime()* fonksiyonunu kullanarak tarih ve saate ilişkin bütün bilgilerin yer aldığı bir demet elde edebilirsiniz:

```
>>> time.localtime()
```

```
time.struct_time(tm_year=2010, tm_mon=8, tm_mday=16,  
tm_hour=16, tm_min=49, tm_sec=42, tm_wday=0,  
tm_yday=228, tm_isdst=1)
```

Bu demette toplam 9 adet değer bulunur ve bu değerlerin anlamı şudur:

Sıra	Değer	Anlamı
0	tm_year	yıl
1	tm_mon	ay
2	tm_mday	gün
3	tm_hour	saat
4	tm_min	dakika
5	tm_sec	saniye
6	tm_wday	haftanın günü (Pazartesi 0)
7	tm_yday	yıla göre gün
8	tm_isdst	gün ışığından yararlanma uygulamasının olup olmadığını denetler

Bu değerlere şu şekilde ulaşabiliriz:

```
>>> zaman = time.localtime()
>>> zaman.tm_year

2010

>>> zaman.tm_mon

8 #Ağustos ayı...

>>> zaman.tm_mday

16 #Ayın 16'sı...

>>> zaman.tm_hour

16 #Saat akşam 4

>>> zaman.tm_min

49 #49. dakika

>>> zaman.tm_wday

0 #Pazartesi...

>>> zaman.tm_yday

228 #yılın 228. günü...

>>> zaman.tm_isdst

1 #gün ışığından yararlanma uygulaması etkin
```

## 25.4 gmtime(), time() ve ctime() Fonksiyonları

Python'da zaman hesaplaması yapılırken “epoch” adlı bir kavramdan yararlanılır. Epoch zamanın başlangıcıdır. Zamanın başlangıcının ne olduğunu bulmak için *gmtime()* fonksiyonunu kullanabiliriz:

```
>>> print time.gmtime(0).tm_year

1970
```

Demek ki “epoch”, yani zamanın başlangıcı 1970 imiş... *time()* adlı başka bir fonksiyonu kullanarak zamanın başlangıcından bu yana kaç saniye geçtiğini bulabiliriz:

```
>>> time.time()
1282025564.984
```

*ctime()* fonksiyonu ise bize tam tarih ve saat bilgisini gösterir:

```
>>> time.ctime()
'Tue Aug 17 09:13:47 2010'
```

*time()* ve *ctime()* fonksiyonlarını birlikte kullanarak sonraki bir zamanı hesaplayabiliriz. Mesela şu andan 60 saniye sonrasını hesaplamak için şöyle bir şey yazabiliriz:

```
>>> sonraki_tarih = time.time() + 60
>>> print time.ctime(sonraki_tarih)
Tue Aug 17 09:17:44 2010
```

Eğer tam bir saat sonrasını hesaplamak isterseniz şu kodu kullanabilirsiniz:

```
>>> bir_saat_sonra = time.time() + (60 * 60)
>>> print time.ctime(bir_saat_sonra)
Tue Aug 17 10:18:59 2010
```

## ÖZEL KONULAR

---

# len() Fonksiyonu ve ascii'nin Laneti

---

Bildiğiniz gibi, içinde Türkçe karakterler geçen bir program yazdığımızda betiğimizin en başına şu satırı eklememiz gerekir:

```
# -*- coding: utf8 -*-
```

Not: Windows kullananlar “utf8” yerine “cp1254” kodlamasını kullanabilir...

Diyelim ki şöyle bir şey var elimizde:

```
#!/usr/bin/env python  
  
print "Türkçeye özgü karakterler: şçöğü"
```

Burada olduğu gibi, “#--coding:utf8--” satırını eklemezsek şuna benzer bir hata alırız:

```
File "deneme.py", line 3  
SyntaxError: Non-ASCII character '\xc3' in file deneme.py on line 3,  
but no encoding declared; see http://www.python.org/peps/pep-0263.html for details
```

Burada Python bize ASCII olmayan bir karakter kullandığımızı, üstelik uygun bir kodlama düzeni de belirtmediğimizi söylüyor. Bu hatayı almamak için betiğimizi şu hale getirmemiz gerekir:

```
#!/usr/bin/env python  
# -*- coding: utf8 -*-  
  
print "Türkçeye özgü karakterler: şçöğü"
```

Burada eklediğimiz “#--coding:utf8--” satırı yardımıyla betiğimizin kodlama düzenini “utf8” olarak belirledik. Eğer betiğimiz içinde kendimiz bir kodlama düzeni belirtmezsek, Python otomatik olarak “ascii” kodlamasını kullanacaktır. Şu komutun çıktısına baktığımızda gerçekten de Python’un varsayılan kod çözücüsünün “ascii” olduğunu görüyoruz:

```
>>> import sys  
>>> sys.getdefaultencoding()  
  
'ascii'
```



Peki bu “ascii” denen şey de ne oluyor?

“ascii” kelimesi İngilizce “American Standard Code for Information Interchange” (Bilgi Alışverişi için Amerikan Standart Kodu) ifadesinin kısaltması... “ascii” denen karakter kodlama düzeni İngiliz alfabesi temel alınarak 1960’lı yıllarda hazırlanmış. Amacı da o dönemde kullanımda olan yazı makinelerinin standartlaşmasını ve böylece bu makinelerin birbirleriyle uyumlu çalışmasını sağlamak. “ascii” kelimesini “askii” şeklinde telaffuz ediyoruz... “ascii” kodlamasında, İngiliz alfabesindeki her harf bir sayıya karşılık geliyor. Mesela “65” sayısının karşılığı “A” harfidir... Şu kod yardımıyla bunu doğrulayabiliriz:

```
>>> print chr(65)
```

A

Örneğin “100” sayısının hangi karaktere karşılık geldiğine bakalım:

```
>>> print chr(100)
```

d

Demek ki “100” sayısı “d” harfine karşılık geliyormuş. Standart “ascii” kodlamasında toplam 128 karakter bulunur. Bunların ilk 33 tanesi ekranda görünmeyen karakterlerdir. Bu karakterlere, yazılan bir metnin akışını kontrol etme görevi gördükleri için, “kontrol karakterleri” adı veriliyor. Geri kalan 95 karakter ise ekranda bilfiil gördüğümüz karakterlerdir. ascii kodlamasında hangi sayının hangi karaktere denk geldiğini bulmak için <http://www.asciitable.com/> adresindeki tablodan faydalanabilirsiniz. Ayrıca isterseniz şu komutu kullanarak kendiniz de benzer bir ascii tablosu üretebilirsiniz:

```
>>> for i in range(128):  
...     print chr(i)
```

Bu tabloda hangi sayının hangi karaktere karşılık geldiğini açık açık görebilmek için ise şöyle bir şey yazabilirsiniz:

```
>>> for i in range(128):  
...     print "%s => %c"%(i,i)
```

Burada “%s”, 128 aralığındaki sayıların “bildiğimiz” halini; “%c” ise bu sayıların “char” yani “karakter” karşılıklarını gösteriyor... Peki burada neden 128’e kadar olan sayıları kullandık? Çünkü yukarıda da söylediğimiz gibi ascii tablosunda toplam 128 karakter bulunur. Yani bu tabloda sadece 128 adet karakter tanımlanmıştır. Bu durumun bir yansımasını, daha önce bir yerlerde muhtemelen karşınıza çıkmış olan şu hata mesajından da hatırlıyor olabilirsiniz:

```
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc5 in position 0:  
ordinal not in range(128)
```

Buradan anladığımıza göre, Python’un varsayılan kod çözücüsü olan “ascii”, konumu belirtilen karakteri çözümleyemiyormuş. Yani söz konusu karakter, yukarıda bahsettiğimiz 128 karakterlik tablonun dışında kalmış...

ascii tablosuna baktığımız zaman, Türkçe harflerin (şçöğı) bu tabloda bulunmadığını görüyoruz. İşte biz betiğimizde “utf8” gibi bir kodlama düzeni belirtmediğimiz zaman, Python’un varsayılan kodlama düzeni olan “ascii”, tabloda bulunmayan bu karakterleri çözümleyemiyor. Betiğimizde “utf8” kodlaması kullanarak Python’un “ascii” yerine “utf8”i kullanmasını sağlıyoruz. “utf8” kodlaması, “ascii”nin aksine Türkçe karakterleri de çözümleyebiliyor.

Şimdi yukarıda anlattığımız meseleye biraz daha yakından bakalım. Hemen Python komut satırında şunu yazıyoruz:

```
>>> "a"
```

Buradan elde edeceğimiz çıktı tabii ki şöyle olacaktır:

```
'a'
```

Gördüğünüz gibi Python “a” harfini doğru bir şekilde algıladı. Şimdi bir de şuna bakalım:

```
>>> "ş"
```

Burada Türkçe’ye özgü bir harf olan “ş”yi kullandık. Bakalım Python ne yapacak?

```
'\xc5\x9f'
```

Python, bu harfi “a” harfinden farklı bir şekilde yorumladı. Bunun nedeni “a” harfinin “ascii” tablosunda bulunuyorken, “ş” harfinin bu tabloda bulunmuyor olması...

Şimdi bu konuya ufak bir ara verelim. Aslında ara vermekten ziyade aynı konuya başka bir pencereden bakmaya devam edeceğiz...

Hepimizin bildiği gibi, Python’da bir nesnenin boyutunu öğrenmek için len() fonksiyonundan yararlanıyoruz. En basit şekliyle bu fonksiyonu şu şekilde kullanabiliriz:

```
>>> a = "python"
>>> len(a)
6
```

Demek ki “python” karakter dizisi içinde 6 karakter varmış. Bir de şuna bakalım:

```
>>> b = "çilek"
>>> len(b)
6
```

Burada len() fonksiyonu “6” sonucunu verdi. Ama gördüğünüz gibi aslında “çilek” karakter dizisi 6 değil, 5 karakterden oluşuyor!.. Şimdi bu tuhaflığın nereden kaynaklandığını bulmaya çalışalım:

```
>>> len("s")
1
>>> len("a")
1
>>> len("ç")
2
```

Gördüğünüz gibi, “a”, “s” gibi harfler normal olarak 1 karakter uzunluğunda, ama “ç” harfi öyle değil... Dolayısıyla Python “çilek” karakter dizisi içinde geçen “ç” harfini tek başına 2 saydığı için toplam 6 karakter buluyor... Burada “a” ve “s” harflerinin ortak özelliği, bunların “ascii” kod tablosunda yer alıyor olması. Ya da kabaca şöyle diyebiliriz: “a” ve “s” harflerinin ortak özelliği her ikisinin de İngiliz alfabesinde bulunuyor olması... Dolayısıyla bunlar “ascii” ile kodlandığında tek karakter şeklinde temsil edilebiliyor. “ç” harfinin suçu ise Türkçe’ye özgü karakterlerden biri olması... Aynı durumu, Türkçe’ye özgü öteki karakterlerde de görüyoruz:

```
>>> len("1")
2
>>> len("ş")
2
>>> len("ö")
2
>>> len("ğ")
2
```

Yukarıda Python'un "ş" harfini nasıl gösterdiğini hatırlıyoruz:

```
>>> "ş"
'\xc5\x9f'
```

Aynı şekilde öteki Türkçe harfler de İngilizce harflerden farklı görünecektir:

```
>>> "ç", "ö", "ğ", "ş", "ı"
('\xc3\xa7', '\xc3\xb6', '\xc4\x9f', '\xc5\x9f', '\xc4\xb1')
```

Aslında sorunun temelinde şu yatıyor: len() fonksiyonu, çoğu kişinin zannettiği gibi, karakter sayısını saymaz. Bu fonksiyonun yaptığı iş, bir karakter dizisindeki bayt sayısını saymaktır. Yani aslında len() fonksiyonu bir karakter dizisinin bir veya birden fazla karakter (veya harf) içerip içermemesiyle ilgilenmez. Onun ilgilendiği, yalnızca bayt sayısıdır. Çünkü bilgisayarlar sadece sayılardan anlar. Karakterler veya harfler özünde bilgisayara hiç bir şey ifade etmez. Python'un varsayılan kod çözücüsü olan ascii yalnızca 1 baytlık verileri doğru olarak gösterebilir. Türkçe harfler ascii kodlamasına göre 1 baytla gösterilemediği için de yukarıdaki gibi bir durum ortaya çıkar. Esasında yukarıdaki çıktıları incelediğimiz zaman bu durumu net olarak görüyoruz. Mesela "ş" harfinin çıktısına bakalım tekrar:

```
>>> "ş"
'\xc5\x9f'
```

Buradaki çıktıda "xcf" + "x9f" şeklinde gösterilen toplam 2 bayt var. İşte len() fonksiyonu da karakter içindeki bu bayt sayısına bakıyor. Bayt sayısı 1'den fazla olduğunda ise çuvallıyor!.. Bu problem tabii ki yalnızca Türkçe'ye özgü karakterler için geçerli değil. Mesela "avro" işareti de ascii'nin lanetinden payını alıyor:

```
>>> "€"
'\xe2\x82\xac'
>>> len("€")
3
```

Demek ki, "€" işareti tek başına üç bayt içeriyormuş... ("xe2", "x82" ve "xac").

Bütün bu anlattıklarımızdan yola çıkarak, özellikle Türkçe yazılmış bir programda, eğer kullanıcıdan birtakım veriler alıyorsak ve program içinde bu verilerin uzunluğunu ölçüyorsak,

yukarıdaki duruma dikkat etmemiz gerekir. Çünkü len() fonksiyonunu kullanırken aslında bir karakterin boyunu posunu değil, bayt sayısını ölçüyoruz. Bu duruma bir örnek verelim. Diyelim ki elimizde şöyle bir betik var:

```
#!/usr/bin/env python
#-*-coding:utf8 -*-

liste = [ ]

while True:
    soru = raw_input("lütfeñ bir karakter giriniz: ")
    if len(soru) == 1:
        liste.append(soru)
        print liste

    else:
        print "lütfeñ birden fazla karakter girmeyiniz"
```

Burada, kullanıcıdan tek bir karakter girmesini istiyoruz. Amacımız kullanıcının yanlışlıkla veya bilerek birden fazla karakter girmesini engellemek. Böyle bir şeye, örneğin bir adasmaca oyunu yazıyorsanız ihtiyacınız olabilir... Yukarıda yazdığımız kodlara göre, eğer kullanıcı sadece İngilizce’de bulunan harfleri (ya da daha doğru bir ifadeyle ascii tablosunda bulunan karakterleri) girerse sorun olmayacaktır. Ama kullanıcı “şçğ” gibi harfler girmeye kalkışrsa, bunlar da görünüşte tek karakter olmasına rağmen, programımız if bloğu yerine else bloğunu işletecektir. Böyle bir durumda kullanıcıdan küfür yemeye hazırlıklı olun!...

Şu ana kadar bahsettiğimiz, “ascii’nin lanetinden” kurtulmanın tek yolu, karakter dizilerimizi Python’un varsayılan kod çözücüsünün ellerine ve insafına bırakmamaktır. Bunun yerine, karakter dizilerimizi mutlaka “unicode” olarak kodlamamız gerekir (Bu “unicode” konusu başka bir yazıda ayrıntılı olarak ele alınacak). Türkçe için en uygun unicode kodlaması “utf8” olacaktır. (unicode ve utf8 ile ilgili güzel bir belge için bkz: <http://www.cl.cam.ac.uk/~mgk25/unicode.html#unicode>)

Hemen şöyle bir örnek yapalım:

```
>>> a = "ışık"
>>> len(a)

7
```

Burada Python “ı”, “ş” ve “ı” harflerinin her birini çift saydığı için toplam 7 karakter buldu. Yanlış! Ama biz şimdi Python’a doğru yolu göstereceğiz:

```
>>> a = "ışık"
>>> a = unicode(a,"utf8")
>>> len(a)

4
```

Gördüğünüz gibi burada, “a = unicode(a,”utf8”)” satırı yardımıyla a değişkenini “utf8” olarak kodladık. Yani karakter dizimizi ascii’nin elinden kurtardık. Bir de bize daha önce sorun çıkaran “ş”, “ç”, “ğ” gibi harflerin durumuna bakalım:

```
>>> a = "ş"
>>> a = unicode(a,"utf8")
>>> len(a)

1
```

```
>>> b = "ğ"
>>> b = unicode(b,"utf8")
>>> len(b)
```

1

Yukarıda `ascii`'nin yanlış olarak "2 karakter uzunluğunda" gösterdiği bu harflerin "utf8" kodlamasıyla doğru olarak gösterildiğini görüyoruz... Bir de şu zavallı "avro" işaretine bakalım:

```
>>> avro = "€"
>>> len(avro)

3

>>> avro = unicode(avro,"utf8")
>>> len(avro)
```

1

Gördüğünüz gibi, avromuz artık mutlu!!

Bütün bu söylediklerimizden çıkan sonuca göre, bizim yukarıda verdiğimiz, kullanıcıdan karakter girmesini isteyen betiğimizin içinde bir değişiklik yaparak, betiğimizin Python'un varsayılan çözücüsü yerine, mesela "utf8"i kullanmasını sağlamamız gerekiyor. Yani kodlarımızı şöyle yazmalıyız:

```
#!/usr/bin/env python
#-*-coding:utf8 -*-

liste = [ ]

while True:
    soru = raw_input("lütfen bir karakter giriniz: ")
    if len(unicode(soru,"utf8")) == 1:
        liste.append(soru)
        for i in liste:
            print i
    else:
        print "Lütfen sadece tek bir karakter giriniz!"
```

Dikkat ederseniz, "if len(unicode(soru,"utf8")) == 1:" gibi bir satır eklemenin yanısıra, listeyi ekrana yazdırırken bir "for" döngüsü kurarak listedeki Türkçe karakter içeren öğelerin ekrana düzgün yazdırılmasını da sağladık.

Böylece "ascii" ve kodlama meselesine ayrıntılı sayılabilecek bir bakış sunmuş olduk. Daha sonra, bu makalede bahsedilen ve bu konuyla yakından ilişkili olan "unicode" konusuna da değineceğiz...

---

# Python'da id() Fonksiyonu, İşleci ve Önbellekleme Mekanizması

---

Python'da her nesnenin bir "kimliği" (identity) vardır. Kabaca söylemek gerekirse, bu "kimlik" denen şey esasında o nesnenin bellekteki adresini temsil eder. Python'daki her nesnenin kimliği eşsiz, tek ve benzersizdir. Peki bir nesnenin kimliğine nasıl ulaşırız? Python'da bu işi yapmamızı sağlayacak basit bir fonksiyon bulunur. Bu fonksiyonun adı "id()". Yani İngilizce'deki "identity" (kimlik) kelimesinin kısaltması. Şimdi örnek bir nesne üzerinde bu id() fonksiyonunu nasıl kullanacağımıza bakalım:

Bildiğiniz gibi Python'da her şey bir nesnedir. Dolayısıyla örnek nesne bulmakta zorlanmayacağız... Herhangi bir "şey", oluşturduğumuz anda Python açısından bir nesneye dönüşüverektir zaten:

```
>>> a = 100
>>> id(a)

137990748
```

Çıktıda gördüğümüz "137990748" sayısı a değişkeninin tuttuğu "100" sayısının kimliğini gösteriyor. Şimdi id() fonksiyonunu bir de şu şekilde deneyelim:

```
>>> id(100)

137990748
```

Gördüğünüz gibi, Python a değişkenini ve 100 sayısını ayrı ayrı sorgulamamıza rağmen aynı kimlik numaralarını gösterdi. Bu demek oluyor ki, Python iki adet "100" sayısı için bellekte iki farklı nesne yaratmıyor. İlk kullanımda önbelleğine aldığı sayıyı, ikinci kez ihtiyaç olduğunda bellekten alıp kullanıyor. Ama bir de şu örneklerle bakalım:

```
>>> a = 1000
>>> id(a)

138406552

>>> id(1000)
```

```
137992088
```

Bu defa Python a değişkeninin tuttuğu 1000 sayısı ile öteki 1000 sayısı için farklı kimlik numaraları gösterdi. Bu demek oluyor ki, Python a değişkeninin tuttuğu 1000 sayısı için ve doğrudan girdiğimiz 1000 sayısı için bellekte iki farklı nesne oluşturuyor. Yani bu iki 1000 sayısı Python açısından birbirinden farklı... Bu durumu görebileceğimiz başka bir yöntem de Python'daki "is" işlecini kullanmaktır. Deneyelim:

```
>>> a is 1000
```

```
False
```

Gördüğünüz gibi, Python "False" (Yanlış) çıktısını suratımıza bir tokat gibi çarptı... Peki bu ne anlama geliyor? Şöyle ki: Python'da "is" işlecini kullanarak iki nesne arasında karşılaştırma yapmak güvenli değildir. Yani "is" ve "==" işleçleri birbirleriyle aynı işlevi görmez... Bu iki işleç nesnelerin farklı yönlerini sorgular: "is" işleci nesnelerin kimliklerine bakıp o nesnelerin aynı nesneler olup olmadığını kontrol ederken, "==" işleci nesnelerin içeriğine bakarak o nesnelerin aynı değere sahip olup olmadıklarını sorgular. Yani:

```
>>> a is 1000
```

```
False
```

Ama...

```
>>> a == 1000
```

```
True
```

Burada "is" işleci a değişkeninin tuttuğu veri ile 1000 sayısının aynı kimlik numarasına sahip olup olmadığını sorgularken, "==" işleci a değişkeninin tuttuğu verinin 1000 olup olmadığını denetliyor. Yani "is" işlecinin yaptığı şey kabaca şu oluyor:

```
>>> id(a) == id(1000)
```

```
False
```

Şimdiye kadar denediğimiz örnekler hep sayıydı. Şimdi isterseniz bir de karakter dizilerinin durumuna bakalım:

```
>>> a = "python"  
>>> a is "python"
```

```
True
```

Burada "True" çıktısını aldık. Bir de "==" işleci ile bir karşılaştırma yapalım:

```
>>> a == "python"
```

```
True
```

Bu da normal olarak "True" çıktısı veriyor. Ama şu örneğe bakarsak:

```
>>> a = "python güçlü ve kolay bir programlama dilidir"  
>>> a is "python güçlü ve kolay bir programlama dilidir"
```

```
False
```

Ama...

```
>>> a == "python güçlü ve kolay bir programlama dilidir"
```

```
True
```

“is” ve “==” işleçlerinin nasıl da farklı sonuçlar verdiğini görüyorsunuz. Çünkü bunlardan biri nesnelerin kimliğini sorgularken, öbürü nesnelerin içeriğini sorguluyor. Ayrıca burada dikkatimizi çekmesi gereken başka bir nokta da “python” karakter dizisinin önbelleğe alınıp gerektiğinde tekrar tekrar kullanılıyorken, “python güçlü ve kolay bir programlama dilidir” karakter dizisinin ise önbelleğe alınmıyor olmasıdır... Aynı karakter dizisinin tekrar kullanılması gerektiğinde Python bunun için bellekte yeni bir nesne daha oluşturuyor...

Peki neden Python, örneğin, 100 sayısını ve “python” karakter dizisini önbelleklerken 1000 sayısını ve “python güçlü ve kolay bir programlama dilidir” karakter dizisini önbelleğe almıyor. Sebebi şu: Python kendi iç mekanizmasının işleyişi gereğince “ufak” nesneleri önbelleğe alırken “büyük” nesneler için her defasında yeni bir depolama işlemi yapıyor. İsterseniz Python açısından “ufak” kavramının sınırının ne olabileceğini şöyle bir kod yardımıyla sorgulayabiliriz:

```
>>> for k in range(1000):  
...     for v in range(1000):  
...         if k is v:  
...             print k
```

Bu kod 1000 aralığındaki iki sayı grubunu karşılaştırıp, kimlikleri aynı olan sayıları ekrana döküyor... Yani bir bakıma Python’un hangi sayıya kadar önbellekleme yaptığını gösteriyor. Burada aldığımız sonuca göre şöyle bir denetleme işlemi yapalım:

```
>>> a = 256  
>>> a is 256
```

```
True
```

```
>>> a = 257  
>>> a is 257
```

```
False
```

Dediğimiz gibi, id() fonksiyonu ve dolayısıyla “is” işleci, nesnelerin kimliklerini denetler. Örneğin bir listenin kimliğini şöyle denetleyebiliriz:

```
>>> liste = ["elma", "armut", "kebab"]  
>>> id(liste)
```

```
3082284940L
```

Bunu başka bir liste üzerinde daha deneyelim:

```
>>> liste2 = ["elma", "armut", "kebab"]  
>>> id(liste2)
```

```
3082284172L
```

Gördüğümüz gibi, içerik aynı olduğu halde iki listenin kimliği birbirinden farklı... Python bu iki listeyi bellekte iki farklı adreste depoluyor. Ama bu listelerin ait olduğu veritipi (yani “list”), bellekte tek bir adreste depolanacaktır. Çünkü bir nesnenin veritipinin kendisi de başlıbaşına bir nesnedir. Nasıl “liste2” nesnesinin kimlik numarası, bu nesne ortalıkta olduğu sürece aynı kalacaksa, bütün listelerin ait olduğu büyük “list” veritipi nesnesi de tek bir kimlik numarasına sahip olacaktır...



```
>>> id(type(liste))
>>> 3085544992L
>>> id(type(liste2))
3085544992L
```

Bu iki çıktı aynıdır, çünkü Python “list” veritipi nesnesi için bellekte tek bir adres kullanıyor. Ama ayrı listeler için ayrı adres kullanıyor... Aynı durum tabii ki öteki veritipleri için de geçerlidir. Mesela “dict” veritipi (sözlük)

```
>>> sozluk1 = {}
>>> id(sozluk1)
3082285236L

>>> sozluk2 = {}
>>> id(sozluk2)
3082285916L
```

Ama tıpkı “list” veritipinde olduğu gibi, “dict” veritipinin nesne kimliği de hep aynı olacaktır:

```
>>> id(type(sozluk1))
3085549888L

>>> id(type(sozluk2))
3085549888L
```

Peki biz bu bilgidan nasıl yararlanabiliriz? Şöyle: “is” işlecini doğrudan iki nesnenin kendisini karşılaştırmak için kullanamamak da bu nesnelerin veritipini karşılaştırmak için kullanabiliriz. Mesela şöyle bir fonksiyon yazabiliriz:

```
# -*- coding: utf-8 -*-

def karsilastir(a,b):
    if type(a) is type(b):
        print "Bu iki nesne aynı veritipine sahiptir"
        print "Nesnelerin tipi: %s"%(type(a))
    else:
        print "Bu iki nesne aynı veritipine sahip DEĞİLDİR!"
        print "ilk argümanın tipi: %s"%(type(a))
        print "ikinci argümanın tipi: %s"%(type(b))
```

Burada “if type(a) is type(b):” satırı yerine, tabii ki “if id(type(a)) == id(type(b))” da yazılabilir... Çünkü “is” işleci, dediğimiz gibi, iki nesnenin kimlikleri üzerinden bir sorgulama işlemi yapıyor.

“is” işlecini veritipi karşılaştırması yapmak için isterseniz şu şekilde de kullanabilirsiniz:

```
>>> if type(a) is dict:
...     sonuca göre bir işlem...

>>> if type(b) is list:
...     sonuca göre başka bir işlem...
```

```
>>> if type(c) is file:
...     sonuca göre daha başka bir işlem...
```

“is” işlecini aynı zamanda bir nesnenin “None” değerine eş olup olmadığını kontrol etmek için de kullanabilirsiniz. Çünkü “None” değeri bellekte her zaman tek bir adreste depolanacak, dolayısıyla bu değere gönderme yapan bütün nesneler için aynı bellek adresi kullanılacaktır:

```
>>> if b is None:
...     ....
```

gibi...

Sözün özü, “is” işleci iki nesne arasında içerik karşılaştırması yapmak için güvenli değildir. Çünkü Python bazı nesneleri (özellikle “ufak” boyutlu nesneleri) önbelleğine alırken, bazı nesneler için her defasında farklı bir depolama işlemi yapmaktadır. İçerik karşılaştırması için “==” veya “!=” işleçlerini kullanmak daha doğru bir yaklaşım olacaktır.

# Windows'ta Python'ı YOL'a (PATH) Ekleme

GNU/Linux kullananların Python'ın etkileşimli kabuğuna ulaşmak için yapmaları gereken tek şey komut satırında python yazıp ENTER tuşuna basmaktır. Çünkü GNU/Linux dağıtımları Python paketini sisteme eklerken, çalıştırılabilir Python dosyasını da */usr/bin* klasörü içine atar. Bu sayede GNU/Linux kullananlar zahmetsiz bir şekilde Python'ı kurcalamaya başlayabilirler... Ama Windows kullananlar için aynı şeyi söyleyemiyoruz. Çünkü Windows kullananlar <http://www.python.org> adresinden Python programını indirip bilgisayarlarına kurduklarında Python otomatik olarak YOL'a (PATH) eklenmiyor. Bu durumu şu şekilde teyit edebiliriz:

1. *Başlat* > *Çalıştır* yolunu takip edin.
2. Açılan kutucuğa cmd yazıp ENTER tuşuna basın.

Böylece Windows komut satırına ulaşmış olduk.

3. Windows komut satırında python yazıp ENTER tuşuna basın.
4. Şöyle bir hata mesajı almışsanız, Python YOL'a eklenmemiş demektir:

```
'python' iç ya da dış komut, çalıştırılabilir
program ya da toplu iş dosyası olarak tanınmıyor.
```

Bu durumda Python'ın etkileşimli kabuğuna ulaşmak için iki yolunuz var:

Birincisi, Windows komut satırında Python'ın tam yolunu belirteceksiniz:

```
c:/python26/python.exe
```

İkincisi, *Başlat* > *Programlar* > *Python 2.x* > *Python (Command Line)* yolunu takip edeceksiniz.

Birinci yöntem oldukça meşakkatli ve sıkıcıdır. Her program çalıştırışınızda bu kodu girmek zamanla eziyete dönüşebilir. Bu şekilde;

```
python program_adı
```

yazarak programınızı çalıştırma imkanınız da olmayacaktır. Daha doğrusu, bu imkanı elde etmek için Windows komut satırına daha fazla kod yazmanız gerekecek. Yani Python'a ulaşmak için, komut satırına her defasında Python'ın ve/veya kendi programınızın tam yolunu yazmak zorunda kalacaksınız.

İkinci yöntem kolaydır, ancak etkileşimli kabuğa bu şekilde ulaşmanın bazı dezavantajları bulunur. Örneğin bu şekilde kabuğa ulaştığınızda Python'ı istediğiniz dizin içinde başlatamamış oluyorsunuz. Bu şekilde Python'ın etkileşimli kabuğu, Python'ın kurulu olduğu *C:/Python2x* dizini içinde açılacaktır. Etkileşimli kabuk açıkken hangi dizinde olduğunuzu sırasıyla şu komutları vererek öğrenebilirsiniz:

```
>>> import os
>>> os.getcwd()
```

Etkileşimli kabuğu istediğiniz dizinde açmadığınız zaman, örneğin masaüstüne kaydettiğiniz bir modüle ulaşmak için biraz daha fazla uğraşacaksınız demektir.

GNU/Linux kullanıcıları ise, özellikle KDE'yi kullananlar, herhangi bir dizin içinde bulunan bir Python programını çalıştırmak için, o programın bulunduğu dizine girecek ve orada F4 tuşuna basarak bir komut satırı açabilecektir. Bu aşamada sadece python yazıp ENTER tuşuna basarak etkileşimli kabukla oynamaya başlayabilecekler. İşte bu yazımızda buna benzer bir kolaylığa Windows'ta nasıl ulaşabileceğimizi anlatacağız. Yani bu yazımızda, Windows'ta sadece python komutunu vererek nasıl etkileşimli kabuğa erişebileceğimizi öğreneceğiz.

Öncelikle Windows'un masaüstündeki Bilgisayarım simgesine sağ tıklayıyoruz.

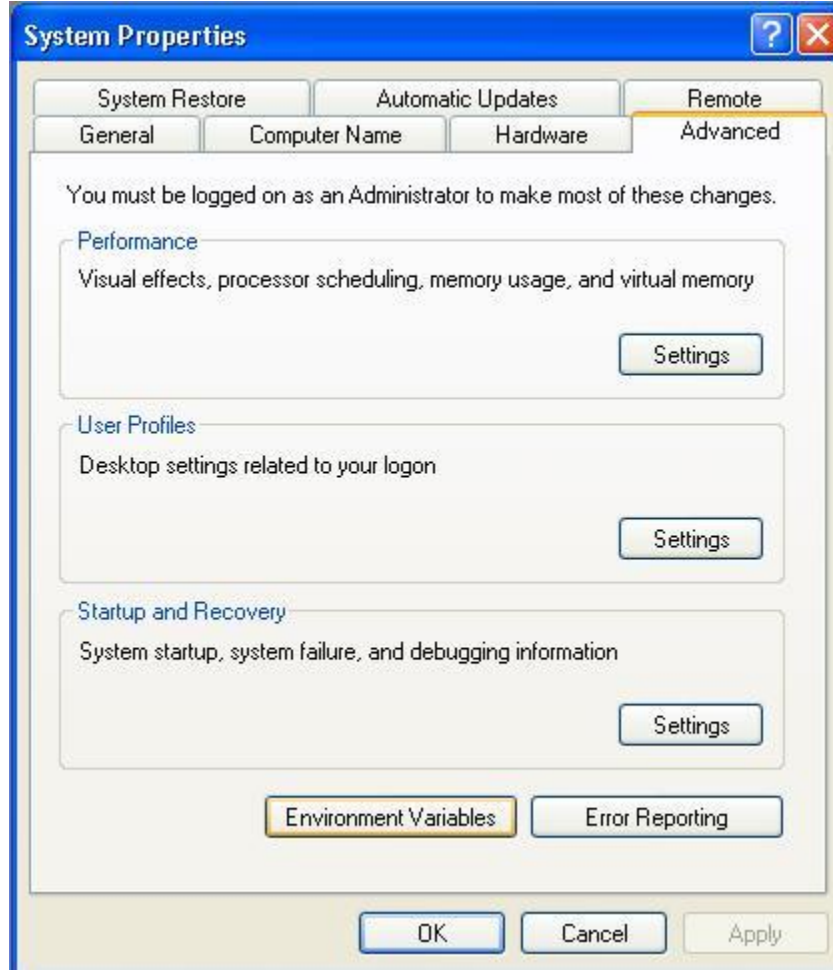


Bilgisayarım simgesine sağ tıkladıktan sonra açılan menünün en altında yer alan “Özellikler”e (*Properties*) giriyoruz.

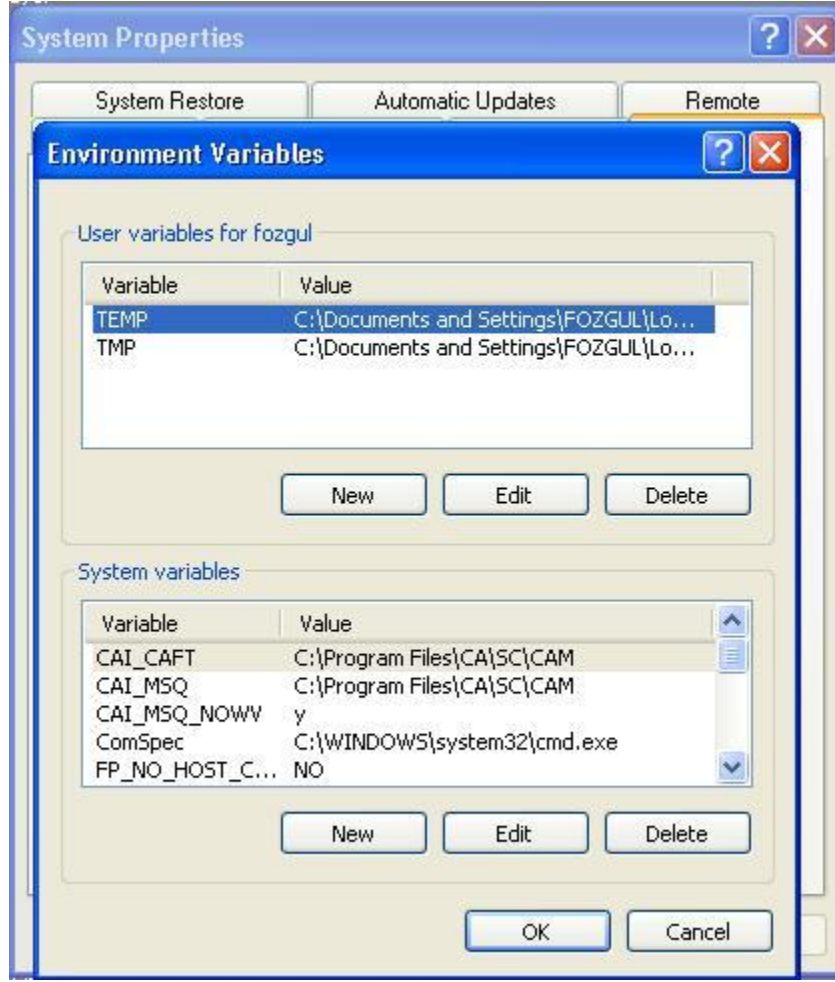
Bu girdiğimiz yerde “Gelişmiş” (*Advanced*) adlı bir sekme göreceğiz. Bu sekmeyi açıyoruz.



“Gelişmiş” sekmesine tıkladığımızda karşımıza şöyle bir ekran gelecek:

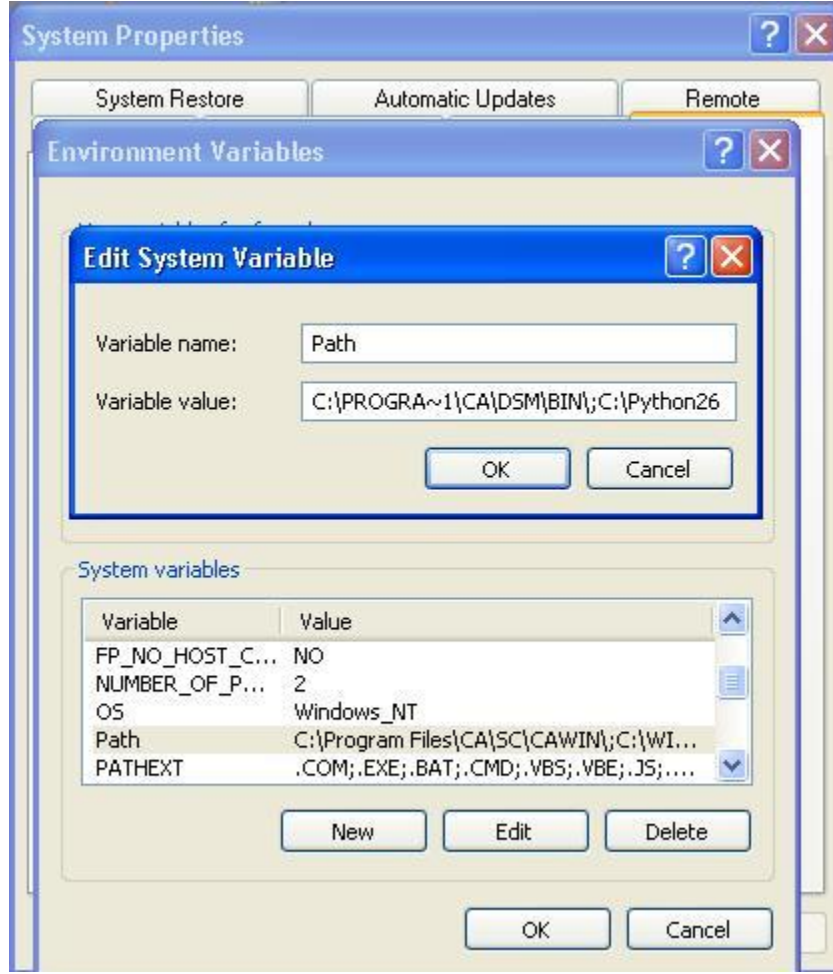


Burada “Çevre Değişkenleri” (*Environment Variables*) düğmesine tıklıyoruz. Bu düğmeye tıkladığımızda şöyle bir pencere açılacak:



Bu ekranda “Sistem Değişkenleri” (*System Variables*) bölümünde yer alan liste içinde *Path* ögesini buluyoruz. Listedeki öğeler alfabe sırasına göre dizildiği için *Path*’i bulmanız zor olmayacaktır.

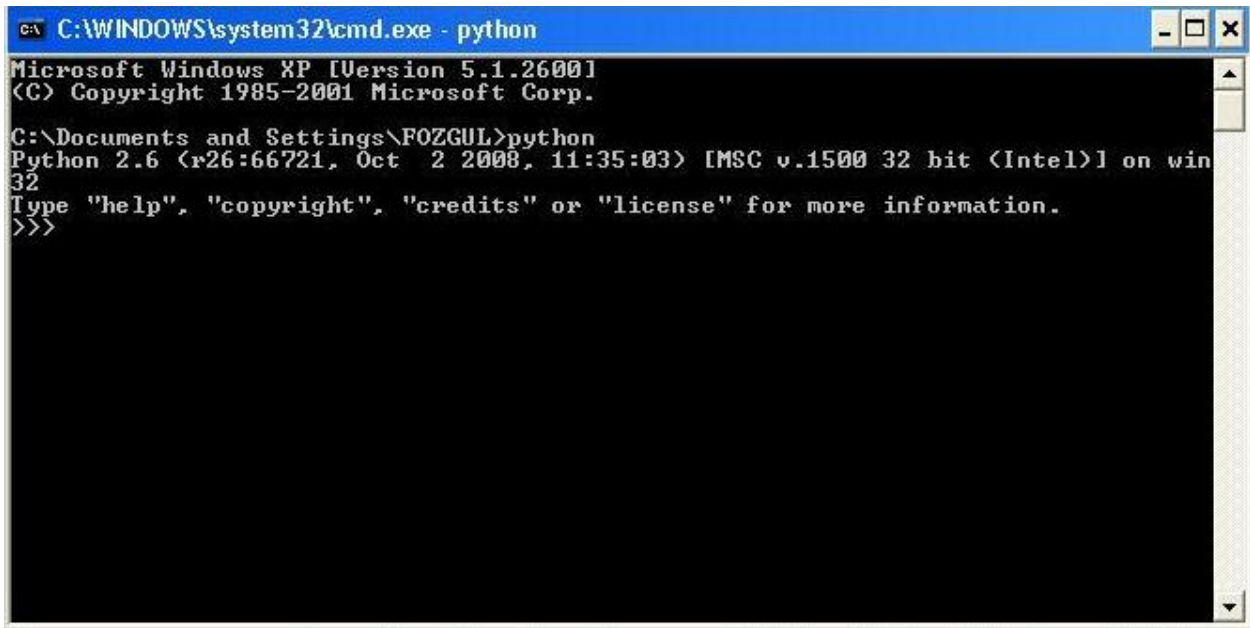
*Path* ögesi seçili iken, “Sistem değişkenleri” bölümündeki “Düzenle” (*Edit*) düğmesine tıklıyoruz. Karşımıza şöyle bir şey geliyor: (Aşağıdaki ekran görüntüsünde “Sistem Değişkenleri” bölümündeki *Path* ögesini de listede görebilirsiniz...)



Bu ekrandaki listenin en sonunda görünen ";C:\Python26" ögesini ben ekledim. Siz de listenin sonuna bu ögeyi aynı şekilde ekleyeceksiniz. Yalnız tabii ki, benim sistemimde Python2.6 kurulu olduğu için oraya ";C:\Python26" yazdım. Sizdeki sürüm farklı olabilir. Sizdeki sürümün tam adını öğrenmek için "C:/" dizinini kontrol edebilirsiniz. Benim sistemimde "C:/" dizini altındaki Python klasörünün adı "Python26". Bu klasörün adı sizdeki sürüme göre farklılık gösterebilir. Doğru sürüm numarasını gerekli yere yazdıktan sonra "Tamam" düğmelerine basarak tekrar masaüstüne dönüyoruz.

Şimdi *Başlat > Çalıştır (Start > Run)* yolunu takip ediyoruz. Açılan kutucuğa cmd yazıp ENTER tuşuna bastıktan sonra karşımıza Windows komut satırı gelecek. Burada artık python komutunu vererek Python'ın etkileşimli kabuğuna ulaşabiliriz.





```
C:\WINDOWS\system32\cmd.exe - python
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\FOZGUL>python
Python 2.6 (r26:66721, Oct 2 2008, 11:35:03) [MSC v.1500 32 bit (Intel)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Etkileşimli kabuktan çıkmak için önce CTRL+Z'ye, ardından da ENTER'e basıyoruz. Artık, cmd ile ulaştığımız bu komut satırında şu komutu vererek Python programlarını çalıştırabiliriz:

```
python program_adı
```

Windows komut satırı Python'ın Windows'ta sunduğu komut satırından biraz daha yeteneklidir. "cmd" komutu ile ulaştığınız ekranda kopyalama/yapıştırma işlemleri de yapabilirsiniz.

Yukarıda Python'ın çalıştırılabilir dosyasını (*executable*) sistem YOL'una nasıl ekleyeceğimizi öğrendik. Ancak çoğu zaman sadece bu dosyayı YOL'a eklemek yeterli olmayacaktır. Sadece çalıştırılabilir dosyayı YOL'a eklediğinizde Python'ı komut satırından çalıştırabilirsiniz, ancak başka bazı modüllerin de çalışabilmesi için *Scripts* adlı dizini de YOL'a eklemenizi tavsiye ederim. Bu dizini YOL'a eklemek oldukça basittir. Bu dizini YOL'a eklemek için yukarıdaki adımları tekrar ediyoruz ve ";C:\Python26\Scripts" satırını da kutucuğa ekliyoruz.

Bu konuyla ilgili herhangi bir sorunuz olması halinde kistihza [et] yahoo [nokta] com adresine yazabilirsiniz...

---

# PEP 3000

---

**Başlığı** PEP 3000 – Python 3000

**Adresi** <http://www.python.org/dev/peps/pep-3000/>

**Yazarı** Guido Van Rossum

**Yazılma Tarihi** 5 Nisan 2006

**Son Değişiklik** 18 Mart 2008

**Türkçesi** Fırat Özgül

---

**Not:** **Çevirmenin Notu:** PEP'in açılımı "Python Enhancement Proposal", yani "Python'u Geliştirme Önerileri"dir. Adından da anlaşılacağı gibi PEP'lerin işlevi Python'un gelişimine ışık tutmaktır. PEP'ler Python topluluğuna çeşitli konularda bilgi vermenin yanı sıra, Python'a ilişkin bir özelliği veya özellik önerisini de tarif eder. PEP'ler genellikle teknik bir dil kullanılarak yazılır. Aşağıdaki metin, yukarıda adresi verilen PEP 3000'in çevirisidir.

---

## 29.1 Özet

Bu PEP Python 3000'in geliştirilmesine ilişkin bir kılavuz niteliğindedir. Tercihen, ilk olarak süreç üzerinde uzlaşılmalı, özellikler konusu ancak süreç kararlaştırılıp kesinleştirildikten sonra irdelenmeye başlanmalıdır. Ama uygulamada özelliklerle sürecin bir arada ele alındığına şahit olacağız. Çoğunlukla belirli bir özellik üzerinde yürütülen tartışmalar, sürecin de sorgulanmasına yol açacaktır.

## 29.2 Adlandırma

Python 3000, Python 3.0 ve Py3K aynı şeyin adıdır. Projenin ismi Python 3000 veya kısaca Py3k'dır. Asıl Python sürümü Python 3.0 olarak adlandırılacak, "python3.0 -V" komutunun çıktısı da bu olacaktır. Dosya adlandırmalarında ise, Python 2.x sürümlerinde tutulan yol benimsenecektir. Çalıştırılabilir dosya için yeni bir ad önermek veya Python kaynak dosyalarının uzantısını değiştirmek istemiyorum.

## 29.3 PEP'lerin Numaralandırılması

Python 3000'e ilişkin PEP'ler PEP 3000 ile başlar. 3000-3099 arası PEP'ler üst-PEP'ler olup; bunlar sürece ilişkin veya bilgi amaçlı PEP'lerdir. 3100-3999 arası PEP'ler özellikler üzerinedir. PEP 3000 (şu anda okuduğunuz PEP) ise özel bir duruma sahiptir: PEP 3000, Python 3000'e ilişkin bütün üst-PEP'lerin üst-PEP'idir. (Başka bir deyişle bu PEP, süreçlerin nasıl tanımlanacağına ilişkin süreci tarif eder). PEP 3100 de özel bir PEP'tir: Bu PEP; asıl Python 3000 sürecine girilmeden önce Python 3000 içinde yer alması kararlaştırılmış (ya da yer alması ümit edilen) özellikleri içeren bir "yapılacaklar listesi" gibidir. Son olarak, PEP 3099 ise, değişmeyecek özellikleri sıralar.

## 29.4 Takvim

Python 2.6 ve 3.0'a ilişkin sürüm takvimini içeren PEP 361'e bakınız. Bu sürümlerin yayınlanma tarihleri başabaş gitmektedir.

Not: Standart kitaplık gelişiminin 3.0a1'den sonra süratlenmesi beklenmektedir.

Python 2.x ve 3.x sürümlerinin bir süre birbirine paralel gitmesini bekliyorum. Python 2.x sürümleri, hata düzeltmelerini içeren geleneksel 2.x.y sürümlerine kıyasla daha uzun bir süre boyunca yayınlanmaya devam edecektir. Normalde 2.(x+1) sürümü çıktığı anda 2.x için hata düzeltme sürümü yayınlamayı bırakıyoruz. Ama 3.0 (son sürüm) çıktıktan sonra bile en az bir veya iki 2.x sürümünün yayınlanacağını tahmin ediyorum. Hatta bu durum 3.1 veya 3.2 sürümlerine kadar devam edebilir. Bunun ne kadar devam edeceği 2.x desteğinin sürdürülmesine yönelik olarak topluluktan gelen taleplere, 3.0 sürümünün kabul görüp kararlı bir hale gelmesine ve gönüllülerin tahammül sınırlarına bağlıdır.

Python 3.0'dan sonra, Python 3.1 ve 3.2'nin çıkışının 2.x serisine kıyasla çok daha çabuk olacağını tahmin ediyorum. Topluluk 3.x sürümlerine alıştıktan sonra 3.x sürüm yayınlama döngüsü rayına oturacaktır.

## 29.5 Uyumluluk ve Geçiş

Python 3.0 ile Python 2.x arasında geriye dönük bir uyumluluk olmayacaktır.

Python 2.6 ile yazılmış kodların Python 3.0 altında hiçbir değişikliğe uğramadan çalışacağını teminatı yoktur. Hatta bir kodun bir bölümünün dahi çalışacağı söylenemez. (Tabii ufak da olsa çalışacak kısımlar vardır, ama ana işlevinden yoksun olarak...)

Python 2.6 ileriye dönük uyumluluğu şu iki şekilde sağlayacaktır:

- Python 2.6 sürümünde, Python 3.0'da artık çalışmayacak özellikler hakkında (mesela `range()` fonksiyonunun dönüş değerinin liste olduğunu varsayarak) dinamik şekilde (yani çalışma esnasında) uyarı verecek olan bir "Py3k uyarı kipi" yer alacaktır.
- Py3k'ya ait pek çok özellik (normalde 2.x'te hata verecek sözdizimleridir bunlar) Python 2.6'ya geri taşınacak, bu özellikler ya `__future__` modülü aracılığıyla ya da eski ve yeni sözdizimi yan yana kullanılarak etkinleştirilebilecektir.

2.6 sürümündeki ileriye dönük uyumluluk özelliklerinin yerine veya bunlara ek olarak ayrı bir kaynak kod dönüştürme aracı da olacaktır [1]. Bu araç, bağlamdan bağımsız olarak kaynaktan kaynağa dönüştürme işlemi yapabilecek, örneğin `apply(f, args)` fonksiyonunu `f(*args)` şekline dönüştürebilecektir. Ancak tabii ki bu araç veri akışı analizi veya tip çıkarsaması yapamaz.

Yani bu araç yukarıdaki gibi bir apply fonksiyonu gördüğünde, bunun o bildiğimiz eski gömülü fonksiyon olduğunu varsayacaktır.

Python 2.6 ve 3.0 sürümlerini birlikte desteklemesi gereken bir proje için önerilen geliştirme modeli aşağıdaki gibidir:

0. Neredeyse tam kapsamlı, kusursuz birim testleri hazırlayın.
1. Projenizi Python 2.6'ya aktarın.
2. Py3k uyarı kipini açın.
3. Uyarı kalmayınca kadar test edin ve kodu düzenleyin.
4. Bu kaynak kodunu 3.0 sürümünün sözdizimine dönüştürmek için 2to3 aracını kullanın. Çıktıyı elle düzenlemeyin!
5. Dönüştürülmüş kaynak kodunu 3.0 sürümü altında test edin.
6. Eğer sorunla karşılaşırsanız, düzeltmeleri kaynak kodunun 2.6 sürümü üzerinde yapın ve 3. basamağa geri dönün.
7. Sürümü yayınlama vakti geldiğinde 2.6 ve 3.0 için ayrı tar dosyaları (veya sürümler için hangi arşivleme biçimini kullanıyorsanız onu) hazırlayın.

Programınızın 2.6 sürümü için vereceğiniz desteği salt bakım seviyesine getirene kadar 3.0 kaynak kodunun düzenlenmesi tavsiye edilmez (yani 2.6 kodunu bakım dalına aktaracağınız ana kadar).

Not: Geçiş sürecine ilişkin konuları ayrıntılı olarak tarif eden bir üst-PEP'e ihtiyacımız var.

## 29.6 Gerçekleme Dili (Implementation)

Python 3000 C dilinde gerçekleştirilecek, yeni sürüm, Python 2 kod tabanından evrilecektir. Bu yaklaşım, baştan aşağı yeniden yazmanın tehlikeleri hakkındaki (Joel Spolsky[2] ile paylaştığım) görüşlerimden kaynaklanmaktadır. Python 3000 Python 2 üzerinde yapılmış nispeten orta düzeyli bir iyileştirme çalışması olduğu için dili baştan aşağı yeniden yazmaya kalkışmazsak çok şey kazanırız. Ben bir yandan da sıfırdan gerçekleştirme çalışmalarına girişilmesine karşı değilim, ama benim kendi çabalarım en iyi bildiğim dile ve gerçeklemeye yönelik olacaktır.

## 29.7 Üst-Katkılar

Bu PEP için yapılacak ilave metin önerileri, yazarı tarafından memnuniyetle karşılanacaktır. Yukarıda anılan ve anılmayan konulara ilişkin taslak şeklindeki üst-PEP'ler ise daha da makbule geçecektir!

## 29.8 Kaynaklar

- [1] [subversion test alanında \(sandbox\) bulunan 2to3 aracı](#)
- [2] [Joel on Software: Katiyen Yapmamanız Gereken Şeyler, Bölüm I](#)
- [3] [PEP 361 \(Python 2.6 ve 3.0 için Sürüm Takvimi\)](#)

## 29.9 Telif Hakkı

Bu belge kamuya açılmıştır.

---

# Python ve OpenOffice

---

---

**Not:** Bu makale henüz taslak halindedir. Zamanla bu makale içindeki bilgiler geliştirilecek ve buraya yeni bilgiler de eklenecektir. Eğer bu makaleye katkıda bulunmak veya makalede bulunduğunuz hataları bildirmek isterseniz kistihza [at] yahoo [nokta] com adresinden bana ulaşabilirsiniz.

---

Çoğu kimsenin bildiği gibi, *Sun Microsystems* firmasının etkin desteğiyle geliştirilen OpenOffice adlı bir özgür yazılımı, *Microsoft* firmasının özgür olmayan ofis paketine alternatif olarak bilgisayarlarımızda kullanabiliyoruz. OpenOffice hem GNU/Linux'ta hem de Windows'ta çalışabiliyor. Özellikle GNU/Linux kullanıcıları, OpenOffice'nin GNU/Linux'taki birincil ve en gelişmiş ofis paketi olmasından ötürü, OpenOffice'yi Windows kullanıcılarına nazaran çok daha yakından tanır... Biz bu bölümde OpenOffice ve Python'u nasıl bağdaştırabileceğimizi, Python'u kullanarak OpenOffice belgelerini nasıl yönetebileceğimizi öğrenmeye çalışacağız.

OpenOffice; Writer, Calc ve öbür ofis bileşenlerini yönetmek, evirip çevirmek, düzenlemek veya değiştirmek için programcılara oldukça gelişmiş bir API (uygulama programlama arayüzü) sunuyor. OpenOffice API'sinin özelliği dilden bağımsız olmasıdır. Yani farklı programlama dillerini kullanan programcılar (örneğin C++, Java, Python, CLI, StarBasic, JavaScript, OLE programcıları), OpenOffice API'sine erişerek ofis bileşenlerini yönetebilirler...

API'ye erişebilmek için "UNO" (Universal Network Objects — Evrensel Ağ Nesneleri) adlı bileşen modelinden faydalanıyoruz. Farklı programlama dilleri, kendilerine özgü bir UNO köprüsü (UNO bridge) kullanarak OpenOffice'nin UNO bileşenlerine ve dolayısıyla OpenOffice'nin API'sine erişebilirler. Şu anda kullanılabilecek olan UNO köprüleri şöyle listelenebilir:

- Uno/Binary
- Uno/CLI
- Uno/Cpp
- Uno/Delphi
- Uno/Java
- Uno/OLE
- Uno/PyUno
- Uno/Remote
- Perl Uno

- Tcl Uno

Gördüğünüz gibi, listede “Uno/PyUno” da var. İsminden de anlaşılacağı gibi, bu köprü, Python ile OpenOffice’nin bileşen modeli arasında bağlantı kurabilmemizi sağlıyor.

## 30.1 PyUno’nun Kurulumu

Daha önce de dediğimiz gibi, Python ve OpenOffice’yi birlikte kullanabilmek için PyUno adlı bir yazılıma ihtiyacımız olacak. O halde isterseniz bu yazılımın sistemimize nasıl kurulacağını inceleyelim ilk iş olarak. Bu işlemin nasıl yapılacağını GNU/Linux ve Windows için ayrı ayrı inceleyeceğiz. Önce GNU/Linux’tan başlayalım.

### GNU/Linux

PyUno, OpenOffice ile birlikte gelen bir modüldür. O yüzden PyUno’yu bilgisayarınıza kurmak için herhangi bir işlem yapmanıza gerek yok. Eğer sisteminizde OpenOffice kurulu ise PyUno da kuruludur.

PyUno’nun kurulu olup olmadığını denetlemek için Python’un etkileşimli kabuğunda şu komutu verebilirsiniz:

```
>>> import uno
```

Eğer hiçbir şey olmadan bir alt satıra geçiliyorsa, PyUno modülü sisteminizde kurulu demektir. Eğer yukarıdaki komut bir hata mesajı veriyorsa okumaya devam edin...

Benim kullandığım işletim sistemi *Ubuntu Jaunty Jackalope*. O yüzden ben bu yazıda PyUno’nun Ubuntu üzerine nasıl kurulacağını anlatacağım. Başka GNU/Linux dağıtımlarını kullananlar, kendi paket yöneticileri aracılığıyla aşağıda anlattıklarım benzer işlemleri gerçekleştirebilirler...

Önce Ubuntu paketleri arasında küçük bir araştırma yapalım. Acaba PyUno paketinin tam adı neymiş?

```
aptitude search uno
```

Bu komut bize şuna benzer bir çıktı verir:

```
plasma-widget-fortunoid
python-uno
python2.6-uno
uno-libs3
uno-libs3-dbg
unoconv
```

Burada “python-uno” ve “python2.6-uno” adlı iki farklı paket görüyoruz. Eğer “python2.6-uno” paketini kurarsanız, PyUno’nun Python 2.6’ya uygun sürümü kurulacaktır. “python-uno” paketi ise, sistemdeki en yeni resmi Python sürümü hangisiyse bilgisayarınıza ona uygun olan PyUno paketini kuracaktır. Şu an itibariyle *Ubuntu Jaunty Jackalope*’deki en yeni resmi Python sürümü 2.6 olduğu için, “python-uno”, bilgisayarımıza PyUno’nun Python 2.6 ile uyumlu olan sürümünü kuracaktır. Dolayısıyla “python-uno” ve “python2.6-uno” bugün itibariyle aynı paketleri kurar... Tabii siz bu yazıyı okurken Ubuntu’da işler değişmiş olabilir... Ancak çoğu durumda en iyi yol “python-uno” adlı paketi kurmak olacaktır. O halde hemen bu paketi sistemimize kuralım:

```
sudo apt-get install python-uno
```

Bu komut, PyUno adlı paketi bilgisayarımıza kuracaktır. Ancak başta da dediğimiz gibi, PyUno OpenOffice ile birlikte geldiği için, muhtemelen bu işlemi yapmak zorunda kalmayacaksınız.

## Microsoft Windows

Microsoft Windows işletim sisteminde de PyUno paketi OpenOffice kurulumu ile birlikte geliyor. Dolayısıyla eğer sisteminizde OpenOffice kurulu ise başka herhangi bir şey kurmanıza gerek yok. OpenOffice'yi ise <http://www.openoffice.org/> adresinden indirebilirsiniz.

Eğer kurulum aşamasını başarıyla geçtiyse emin adımlarla yolumuza devam edebiliriz.

## 30.2 OpenOffice'yi Dinleme Kipinde Açmak (listening mode)

Python'u kullanarak OpenOffice ile herhangi bir işlem yapabilmek için atmamız gereken ilk adım OpenOffice'yi "dinleme kipinde" açmak olmalıdır... Bu sayede Python'la yaptığımız işlemlerin OpenOffice tarafından algılanmasını sağlayacağız.

Bu işlemi sırasıyla GNU/Linux'ta ve Windows'ta nasıl yapacağımızı görelim... Bu arada, aşağıda anlatacaklarımı "GNU/Linux" ve "Microsoft" diye ayırmış olsam da, hangi işletim sistemini kullanıyor olursanız olun hem GNU/Linux hem de Windows açıklamalarını okumalısınız. Çünkü GNU/Linux bölümünde Windows kullanıcılarının da işine yarayacak bilgiler bulunduğu gibi, Windows bölümünde GNU/Linux kullanıcılarının işine yarayacak bilgiler de yer alıyor...

### GNU/Linux

OpenOffice'yi dinleme kipinde açabilmek için "-accept" adlı bir parametreden faydalanacağız. Bu parametre hakkında kısa bir bilgi almak için konsolda şu komutu verebilirsiniz:

```
soffice -h
```

Bu komutun çıktısı içinde şu satırları göreceksiniz:

```
-accept=<accept-string>  
Specify an UNO connect-string to create an UNO acceptor through which  
other programs can connect to access the API
```

Buradan anladığımıza göre, "-accept" parametresi, başka programların OpenOffice API'sine bağlanmak için kullanabileceği bir "bağlantı kabul etme mekanizması" (başka deyişle bir "UNO alıcısı") oluşturulmasını sağlıyor...

Yine yukarıdaki kısa bilgiden anlıyoruz ki, OpenOffice API'sine bağlanmada kullanacağımız kabul mekanizmasını oluşturabilmek için bir "UNO bağlantı dizisi", yani UNO aracılığıyla OpenOffice API'sine bağlanmamızı sağlayacak bilgileri içeren bir karakter dizisi de belirlememiz gerekiyor.

OpenOffice API'sine ilişkin [resmi kılavuzun](#) söylediğine göre bu işlemi yapabilmek için OpenOffice'yi şu şekilde başlatmamız gerekiyor:

```
soffice -accept=socket,host=0,port=2002;urp;
```

Bu komutu alıp olduğu gibi kullanamıyoruz. Bunu konsola yazmadan önce şu sorunu çözmemiz gerek:

Yukarıdaki parametre içinde gördüğümüz ";" işaretini Unix kabuğunun farklı algılamasından ötürü, yukarıdaki komut OpenOffice'yle bağlantı kurmamızı sağlayamayacaktır. Unix kabuğunun bize sorun çıkarmaması için yukarıdaki parametreyi tırnak içine almamız yeterlidir. Yani komutumuzu şöyle yazacağız (Bu arada, bu komutu vermeden önce sistemde hiçbir OpenOffice belgesinin açık olmamasına dikkat ediyoruz)



```
soffice -accept="socket,host=0,port=2002;urp;"
```

Dilerseniz bu komuttaki parametreleri teker teker incelemeye çalışalım:

**soffice:** Bu ifadenin ne işe yaradığını biliyoruz. Sistemimizdeki OpenOffice’yi çalıştırmamızı sağlayan dosyanın adı “soffice”dir.

**-accept:** Bu parametrenin ne olduğunu da biraz önce görmüştük. Bu parametre temel olarak, OpenOffice’nin kendisine gelen bağlantı girişimlerini kabul etmesini sağlıyor. OpenOffice, güvenlik gerekçesiyle, varsayılan olarak hiç bir bağlantı girişimini kabul etmez.

**socket:** Bu ifade, OpenOffice’ye bağlanırken kullanacağımız bağlantı tipini gösteriyor. Biz burada bir “soket bağlantısı” kurmayı deniyoruz... “soket”, sunucu ve istemci arasında çift yönlü bağlantı kurmamızı sağlayan bir mekanizmadır.

**host:** Bu parametre, OpenOffice’nin dinleyeceği konak sistemin adını veya IP numarasını gösteriyor. Bu parametreye “0” veya “localhost” değerlerini verebiliriz. “0” değeri, müsait olan bütün ağ arayüzleri üzerinden dinleme işlemi gerçekleştirilmesini sağlar. Yukarıdaki komutta “host=0” yerine “host=localhost” da yazabiliriz...

**port:** Bu parametre, OpenOffice ile bağlantıyı hangi port üzerinden gerçekleştireceğimizi belirtmemizi sağlıyor. Biz burada, resmi kılavuzun önerisine uyarak, “2002” portunu kullandık.

**urp:** Bu ifade, “Uno Remote Protocol” teriminin kısaltmasıdır. Bu terim Türkçe’de “Uno Uzak Bağlantı Protokolü” anlamına gelir ve kurulacak bağlantının hangi protokol üzerinden gerçekleştirileceğini ifade eder.

Kısacası yukarıdaki komut yardımıyla, “OpenOffice’ye 2002 portu üzerinden bir soket bağlantısı gerçekleştirilerek Uno Uzak Bağlantı Protokolü (URP) aracılığıyla OpenOffice’nin bizi dinlemeye almasını sağladık”. Bu arada, eğer bu komutu verdikten sonra “*unable to get gail version number*” gibi bir hata alırsanız, bunu önemsemeyin... Ancak bu komutu verdikten sonra eğer “*Could not find a Java Runtime Environment*” gibi bir hata mesajı alıyorsanız şu komut yardımıyla gerekli paketi sisteminize kurmalısınız:

```
sudo apt-get install openoffice.org-java-common
```

‘soffice -accept="socket,host=0,port=2002;urp;" komutu verildikten sonra boş bir OpenOffice Writer belgesi açılacaktır. Bu belgeyi kapatmadan yolumuza devam edelim... Ama önce Windows kullanıcılarının durumunu bir gözden geçirelim.

## Microsoft Windows

Windows kullananlar, yukarıda GNU/Linux kullanıcılarının verdiği “soffice -accept...” komutunu, OpenOffice’nin kurulu olduğu dizin içinde verecek. Windows’ta OpenOffice programı “C:\Program Files\OpenOffice 3” dizini altında bulunur. Bu dizinin içinde “program” adlı bir başka dizin daha göreceksiniz. İşte “soffice.exe” dosyasının bulunduğu dizin burasıdır... Şimdi *Başlat>Run*” yolunu takip ederek “cmd” komutunu veriyoruz ve Windows’un komut satırına ulaşıyoruz. Orada önce şu komutu vererek “C:\Program Files\OpenOffice 3\program” dizinine ulaşıyoruz:

```
cd C:\Program Files\OpenOffice.org 3\program
```

Eğer sizdeki OpenOffice dizini farklıysa, yukarıdaki komutu ona göre düzenlemeniz gerekiyor.

Bu noktada, sistemimizde hiçbir OpenOffice belgesinin açık olmadığından emin olmamız gerekiyor. Ayrıca Windows’ta saat simgesinin yanına yerleşen “OpenOffice QuickStarter” uygu-

lmasının da kapalı olması lazım. Eğer kapalı değilse bu uygulamayı da kapatıp şu komutu veriyoruz:

```
soffice -accept=socket,host=localhost,port=2002;urp;
```

Bu komutu verdikten sonra OpenOffice açılacaktır. Eğer ne tür bir belge açmak istediğinizi soran bir ekranla karşılaşırsanız “Text Document” (veya “Boş Belge”) seçeneğini işaretleyerek boş bir metin belgesi açın.

Yukarıdaki komuta ek olarak bazı başka parametreler de verebilirsiniz. Vereceğiniz bu parametreler OpenOffice programının açılış şeklini ekleyecektir:

```
soffice -accept=socket,host=localhost,port=2002;urp; -nofirstwizard
```

Buradaki “-nofirstwizard” parametresi OpenOffice’nin doğrudan boş bir metin belgesi açmasını sağlayacaktır. Bu parametre sayesinde OpenOffice’nin “ne tür bir belge açmak istersiniz?” sorusunu sormamasını sağlıyoruz.

Eğer OpenOffice’nin, ilk açılırken karşımıza çıkan açılış ekranını (splashscreen) da görmek istemiyorsak yukarıdaki komutu şu şekilde de verebiliriz:

```
soffice -accept=socket,host=localhost,port=2002;urp; -nologo
```

Böylece OpenOffice programı, açılış ekranını göstermeden açılacaktır. Eğer ne açılış ekranı ne de “ne tür belge?” ekranı görmek isterseniz yukarıdaki iki parametreyi birlikte de kullanabilirsiniz:

```
soffice -accept=socket,host=localhost,port=2002;urp; -nofirstwizard, -nologo
```

OpenOffice’yi bu parametrelerle birlikte çalıştırdığınızda OpenOffice’nin her zamankinden biraz daha hızlı açıldığını göreceksiniz...

Bu arada, yukarıdaki komutların GNU/Linux’takinden biraz farklı olduğuna dikkat edin. Bu komutu Windows’ta verirken “host=0” yerine “host=localhost” yazıyoruz. Komutta “host=0” yazmak Windows’ta OpenOffice ile bağlantı kuramamamıza sebep olabiliyor... Ayrıca Windows’tayken “socket,host=localhost,port=2002;urp;” satırını tırnak içine almamıza gerek olmadığına da dikkatinizi çekmek isterim...

## 30.3 OpenOffice’ye Bağlanmak

Bir önceki adımda OpenOffice’yi dinleme kipinde açtık. Bu, atmamız gereken ilk adımdı. Şimdi ise Python aracılığıyla OpenOffice’ye bağlanmayı deneyeceğiz. Yine sırasıyla GNU/Linux ve Microsoft kullanıcılarının ne yapması gerektiğini anlatacağız. Tabii hangi işletim sistemini kullanırsanız kullanın, her iki bölümü de dikkatle okumanın zararını değil faydasını görürsünüz...

### GNU/Linux

Öncelikle Python’un etkileşimli kabuğunu başlatıyoruz:

```
python
```

Gayet güzel... Madem işimiz Uno ile, o halde hemen “uno” modülünü içe aktaralım:

```
>>> import uno
```

Böylece “uno” modülünün bütün nimetlerinden faydalanabileceğiz...

Biraz sonra, OpenOffice'ye nasıl bağlanacağımızı inceleyeceğiz. Bu konuya geçmeden önce dilerseniz, bağlantı işleminin hangi aşamalardan oluştuğunu görelim. Bu arada, aşağıda kullanacağımız kavramlara çok takılmayın. Bunların ne olduğunu biraz sonra açıklayacağız:

- Önce “uno” modülünü içe aktarıyoruz,
- Ardından “Bileşen Bağlamını” içe aktarıyoruz,
- “Servis Yöneticisi” yardımıyla OpenOffice bağlantı servisini çalıştırıyoruz,
- Daha sonra OpenOffice'ye bağlantı kuruyoruz,
- Bağlantıyı kurduktan sonra, en başta “soffice” komutuyla açtığımız OpenOffice belgesine erişmemizi sağlayacak servisi çalıştırıyoruz,
- Hemen ardından mevcut belgeye erişiyoruz,
- Belge içinde bir imleç oluşturuyoruz,
- Ve istediğimiz bir metni belgeye yazdırıyoruz.

Şimdi gelelim yukarıda sıralanan aşamaları gerçek hayata aktarmaya...

Bu noktada karşımıza “Bileşen Bağlamı” (Component Context) adı verilen bir kavram çıkıyor. Bileşen Bağlamı kavramını, *“bütün bileşenlerin bir arada bulunduğu bir ortam”* olarak düşünebiliriz. Şimdi yapmamız gereken şey, bu “bileşen bağlamını” içe aktarmak, yani bu bağlamı yerel ortamımızda kullanılabilir hale getirmektir. Bu işlemi, “uno” modülünün “getComponentContext()” adlı metodu yardımıyla gerçekleştireceğiz:

```
>>> uno.getComponentContext()
```

Dilerseniz bu metodu bir değişkene atayalım, ki kullanması kolay olsun:

```
>>> yerel = uno.getComponentContext()
```

Bileşen bağlamını, yani bütün bileşenlerin bir arada bulunduğu ortamı kendi çalışma alanımıza aktardığımız göre, bu noktadan sonra OpenOffice'ye bağlanmamızı sağlayacak olan “UnoUrlResolver” adlı servise erişmeye çalışabiliriz. “UnoUrlResolver”i kullanabilmek için Servis Yöneticisinden (yani “ServiceManager”den) ve onun “createInstanceWithContext()” adlı metodundan yararlanarak “bağlantı servisini” çalışır hale getireceğiz:

```
>>> baglanti_servisi = yerel.ServiceManager.createInstanceWithContext(  
... "com.sun.star.bridge.UnoUrlResolver", yerel)
```

Bir önceki adımda Bileşen Bağlamını yerel ortamımıza aktardığımız için, “ServiceManager”e ve onun “createInstanceWithContext()” adlı metoduna doğrudan yerel ortamımızın bir ögesi olarak erişebiliyoruz. Bu yüzden komutumuzu “yerel.ServiceManager...” şeklinde yazabildik. Burada “createInstanceWithContext()” metodunun iki parametre aldığına dikkat edin. İlk parametre “UnoUrlResolver”, ikinci parametre ise biraz önce hazırladığımız “yerel” adlı değişkendir... Bu arada, “UnoUrlResolver”i nasıl kullandığımıza dikkat edin. “com.sun.star.bridge” OpenOffice API’sinin öğelerinden biridir.

Bağlantı servisini çalışır hale getirdikten sonra “resolve()” metodu yardımıyla OpenOffice'ye bağlanma işlemini gerçekleştireceğiz:

```
>>> baglanti = baglanti_servisi.resolve(  
... "uno:socket,host=0,port=2002;urp;StarOffice.ComponentContext")
```

Burada “resolve()” metoduna verdiğimiz parametrenin, en başta “soffice -accept...” komutuna verdiğimiz parametreyle hemen hemen aynı olduğuna dikkat edin.

Bağlantıyı kurduktan sonra, yine Servis Yöneticisi'nin, yani "ServiceManager"ın "createInstanceWithContext()" metodunu kullanarak mevcut belgeye erişmemizi sağlayacak olan "Desktop" adlı servisi hazır hale getiriyoruz.

```
>>> belge_servisi = baglanti.ServiceManager.createInstanceWithContext(  
... "com.sun.star.frame.Desktop", baglanti)
```

Hemen ardından da mevcut belgeye erişiyoruz:

```
>>> belge = belge_servisi.getCurrentComponent()
```

Artık, en başta "soffice..." komutuyla açtığımız Writer belgesi elimizin altında. Bu belgeye müdahale edebilmek için, bu belge içinde bir imleç oluşturmamız gerekiyor. Şimdi bu imleci oluşturalım:

```
>>> imlec = belge.Text.createTextCursor()
```

Son olarak belgeye bir metin yazdırıyoruz:

```
>>> belge.Text.insertString(imlec, "Elveda Zalim Dünya!", 0)
```

Şimdi bir kenarda açık halde bekleyen Writer belgesine bakın bakalım içinde ne yazıyor...

Dilerseniz bu kodları topluca görelim:

```
>>> import uno  
  
>>> yerel = uno.getComponentContext()  
  
>>> baglanti_servisi = yerel.ServiceManager.createInstanceWithContext(  
... "com.sun.star.bridge.UnoUrlResolver", yerel)  
  
>>> baglanti = baglanti_servisi.resolve(  
... "uno:socket,host=0,port=2002;urp;StarOffice.ComponentContext")  
  
>>> belge_servisi = baglanti.ServiceManager.createInstanceWithContext(  
... "com.sun.star.frame.Desktop", baglanti)  
  
>>> belge = belge_servisi.getCurrentComponent()  
  
>>> imlec = belge.Text.createTextCursor()  
  
>>> belge.Text.insertString(imlec, "Elveda Zalim Dünya!", 0)
```

## Microsoft Windows

Hatırlarsanız, bir önceki adımda OpenOffice programını başlatabilmek için, "cd" komutu yardımıyla "C:\Program Files\OpenOffice.org 3\program" dizini altına gitmiş ve orada "soffice.exe" adlı çalıştırılabilir dosyayı başlatabilmek için "soffice" komutunu vermiştik... Yine "C:\Program Files\OpenOffice.org 3\program" dizini altında "python.exe" adlı bir dosya göreceksiniz. Bu, OpenOffice ile birlikte gelen Python sürümüdür ve sisteminizdeki asıl Python'dan farklıdır. Windows'ta Python'u OpenOffice ile birlikte kullanabilmek için, OpenOffice ile birlikte gelen Python sürümünden yararlanmamız gerekiyor. Sistemdeki öbür Python programı Windows'ta OpenOffice ile bağlantı kurmamızı sağlayamaz... "C:\Program Files\OpenOffice.org 3\program" dizini altındayken komut satırında "python" komutunu verirsek, OpenOffice'nin içindeki Python sürümü çalışmaya başlayacaktır. Eğer arzu ederseniz, "program" dizini altındaki "python.exe" dosyasına çift tıklayarak da OpenOffice'ye özgü Python sürümünü başlatabilirsiniz. O halde şimdi istediğimiz bir yöntemi kullanarak Python'un etkileşimli kabuğunu başlatalım. Orada şu komutu veriyoruz:

```
>>> import uno
```

Bu komutu verdiğimizde Python'un etkileşimli kabuğu sessizce bir alt satıra geçecektir. Eğer OpenOffice ile birlikte gelen Python sürümünü değil de, sisteminizdeki asıl Python sürümünü çalıştırıp "import uno" komutunu verdiyseniz Python size bu modülün bulunamadığına dair bir hata mesajı gösterecektir... O yüzden Windows'ta OpenOffice ile birlikte gelen Python sürümünü kullanmak büyük önem taşıyor.

Şimdi GNU/Linux bölümünde anlattığımız gibi "Bileşen Bağlamını" içe aktarıyoruz:

```
>>> yerel = uno.getComponentContext()
```

Bundan sonra, yine GNU/Linux bölümünde anlattığımız şekilde gerekli servisleri başlatabiliriz:

```
>>> baglanti_servisi = yerel.ServiceManager.createInstanceWithContext(
...     "com.sun.star.bridge.UnoUrlResolver", yerel)

>>> baglanti = baglanti_servisi.resolve(
...     "uno:socket,host=localhost,port=2002;urp;StarOffice.ComponentContext")

>>> belge_servisi = baglanti.ServiceManager.createInstanceWithContext(
...     "com.sun.star.frame.Desktop", baglanti)

>>> belge = belge_servisi.getCurrentComponent()
```

Şu anda yapmamız gereken, belge içinde bir imleç oluşturup, bu imleci de kullanarak belge içine bir karakter dizisi yazdırmaktır:

```
>>> imlec = belge.Text.createTextCursor()

>>> belge.Text.insertString(imlec, u"Elveda Zalim Dünya!", 0)
```

Bu satırlarla ilgili daha geniş açıklamaya GNU/Linux bölümünden ulaşabilirsiniz.

Eğer "*baglanti\_servisi = yerel.ServiceManager.cre...*" satırında bir hata alırsanız, "*\*import uno*" satırından önce "*import socket*" komutuyla "socket" adlı modülü içe aktarmayı deneyin.

Ayrıca GNU/Linux kodlarıyla Windows kodları arasındaki bazı ufak farklılıkları da gözden kaçırmayın. Örneğin "*baglanti = baglanti\_servisi.resolve('uno.....')*" satırında "*host=0*" yerine "*host=localhost*" yazdığımıza dikkat edin...

Bunun dışında, belgeye yazdırdığımız "Elveda Zalim Dünya!" cümlesindeki Türkçe karakterlerin düzgün görüntülenebilmesi için karakter dizimizi "unicode" olarak tanımlamayı da unutuyoruz. (u"Elveda Zalim Dünya!")

## 30.4 Karakter Biçimlendirme

Gördüğümüz gibi, OpenOffice'yle bağlantı kurup boş bir belgeye bir şeyler yazdırmak o kadar da zor değil. Biz bu ilk adımı aştığımıza göre artık OpenOffice'ye yazdığımız metinleri biçimlendirme konusunu incelemeye geçebiliriz...

Unutmayın, OpenOffice'ye bağlanmak için kullandığımız şey Python programlama dilidir. Dolayısıyla Python'un kendi özelliklerini kullanarak OpenOffice belgelerini çok rahat bir biçimde evirip çevirebiliriz. Mesela belgemize yeni satır veya sekme eklemek istersek Python'un kendi araçları emrimize amadedir:

```
>>> belge.Text.insertString(imlec, u"Uzak... \t\tçok uzak...", 0)
```

Burada, bu satıra kadar olan kodları halihazırda yazmış olduğunuzu varsayıyorum...

Aynı şekilde “\n” kaçış dizisini kullanarak yeni bir satır da oluşturabilirsiniz:

```
>>> belge.Text.insertString(imlec, u"Birinci satır\nİkinci satır\nÜçüncü satır", 0)
```

Mesela Python’un “datetime” modülünü kullanarak bugünün tarihini belgemize yazdıralım:

```
>>> import datetime
>>> bugün = datetime.date.today()
>>> belge.Text.insertString(imlec, "%s"%bugün, 0)
```

Bunun dışında, PyUno aracılığıyla OpenOffice belgelerindeki yazıların koyu, yana yatık, altı çizgili, üstü çizgili, vb. şekillerde görünmesini de sağlayabilirsiniz. Bu özellikler tek tek inceleyelim:

### 30.4.1 Kalın Karakterler (bold)

OpenOffice belgesine yazdıracağımız harflerin koyu olabilmesi için, “setPropertyValue()” adlı bir metottan yararlanacağız. “setPropertyValue()” imlecin bir metodudur. Bu ne demek oluyor? Hemen bir örnek verelim...

Hatırlarsanız, PyUno yardımıyla OpenOffice belgelerine bağlantı kurma aşamasında, istediğimiz bir metni belgeye yazdırmadan önce mutlaka bir imleç oluşturmamız gerekiyordu. Bu imleci şöyle oluşturuyorduk:

```
>>> imlec = belge.Text.createTextCursor()
```

İşte oluşturduğumuz bu imlecin “setPropertyValue()” adlı metodunu kullanarak, metne yazdırdığımız harflerin koyu olmasını sağlayabileceğiz:

```
>>> imlec.setPropertyValue("CharWeight", 150)
```

Dilerseniz, daha anlaşılır olması için yukarıdaki kodları bir örnek içinde kullanalım...

Önce komut satırında “soffice” komutuyla OpenOffice’yi başlatalım:

```
soffice -accept="socket,host=localhost,port=2002;urp"
```

Ardından Python’u çalıştıralım. GNU/Linux kullanıcıları kendi sistemlerindeki Python’u kullanabilir, ancak Windows kullanıcılarının OpenOffice ile birlikte gelen Python sürümünü kullanmaları gerekiyor. Sistemimize uygun şekilde Python’u başlattıktan sonra Python’un etkileşimli kabuğunda şu komutları veriyoruz:

```
>>> import uno

>>> yerel = uno.getComponentContext()

>>> baglanti_servisi = yerel.ServiceManager.createInstanceWithContext(
...     "com.sun.star.bridge.UnoUrlResolver", yerel)

>>> baglanti = baglanti_servisi.resolve(
...     "uno:socket,host=localhost,port=2002;urp;StarOffice.ComponentContext")

>>> belge_servisi = baglanti.ServiceManager.createInstanceWithContext(
...     "com.sun.star.frame.Desktop", baglanti)
```

```
>>> belge = belge_servisi.getCurrentComponent()
>>> imlec = belge.Text.createTextCursor()
>>> imlec.setPropertyValue("CharWeight", 150)
>>> belge.Text.insertString(imlec, "deneme", 0)
```

“imlec”in “setPropertyValue()” adlı metodunu nasıl kullandığımıza dikkat edin. Burada “CharWeight” adlı bir özellikten yararlandık. “CharWeight” bir karakterin kalınlık-incelik durumunu belirler. Gördüğünüz gibi, “setPropertyValue()” metodu “CharWeight” dışında bir parametre daha alıyor. Bu parametre temel olarak iki farklı değer alır: “100” ve “150”.

“100” kalın olmayan karakter oluştururken, “150” kalın karakter oluşturur.

### 30.4.2 Yana Yatık Karakterler (italic)

Yukarıda harfleri kalın yazabilmek için “CharWeight” adlı bir özellikten yararlandık. Harfleri yatık yazmak için ise “CharPosture” adlı bir özellikten yararlanacağız:

```
>>> imlec.setPropertyValue("CharPosture", 1)
```

Harfleri yana yatık yazabilmek için “setPropertyValue()” metodunun ikinci parametresi olarak “1” değerini kullanıyoruz. Eğer bu değeri “0” yapacak olursak harflerimiz düz görünecektir...

### 30.4.3 Altı Çizgili Karakterler (underline)

Altı çizgili harfler üretmek için ise “CharUnderline” adlı bir özellikten yararlanacağız. Bu özellik, 0’dan 19’a kadar (19 hariç) olan sayıları değer olarak alabilir. Bu özelliği temel olarak şöyle kullanıyoruz:

```
>>> imlec.setPropertyValue("CharUnderline", 1)
```

Bu komut, yazdıracağımız metnin altını çizecektir. Eğer öteki değerlerin ne işe yaradığını merak ediyorsanız şöyle bir betik yazabilirsiniz:

```
>>> import uno
>>> yerel = uno.getComponentContext()
>>> baglanti_servisi = yerel.ServiceManager.createInstanceWithContext(
... "com.sun.star.bridge.UnoUrlResolver", yerel)
>>> baglanti = baglanti_servisi.resolve(
... "uno:socket,host=localhost,port=2002;urp;StarOffice.ComponentContext")
>>> belge_servisi = baglanti.ServiceManager.createInstanceWithContext(
... "com.sun.star.frame.Desktop", baglanti)
>>> belge = belge_servisi.getCurrentComponent()
>>> imlec = belge.Text.createTextCursor()
>>> for i in range(20):
```

```
... imlec.setPropertyValue("CharUnderline", i)
... belge.Text.insertString(imlec, "%s deneme"%i, 0)
```

Böylece hangi sayının nasıl bir alt çizgi ürettiğini rahatlıkla görebilirsiniz...

### 30.4.4 Üstü Çizili Karakterler (strikeout)

Harflerin üstünü çizmek için "CharStrikeout" adlı özellikten yararlanacağız:

```
>>> imlec.setPropertyValue("CharStrikeout", 1)
```

Yukarıdaki komut, kelimenin üstüne tek çizgi çeker. Eğer "1" yerine "2" yazarsak, bu komut kelimenin üzerine çift çizgi çekecektir. "0" değeri ise kelimenin üzerinde herhangi bir çizgi olmayacağını gösterir.

### 30.4.5 Yanıp Sönen Karakterler (flash)

Eğer yazdıracağımız karakterlerin yanıp sönmesini istersek "CharFlash" adlı bir özellikten yararlanabiliriz:

```
>>> imlec.setPropertyValue("CharFlash", True)
```

"CharFlash", True ve False olmak üzere iki farklı değer alır. Eğer Bu özelliği "True" değeriyle birlikte kullanırsak, yazdırdığımız karakter (veya karakterler) yanıp sönecektir. False ise yanıp sönmeyi iptal eder.

### 30.4.6 Gölge Karakterler (shadow)

Eğer karakterlerimize gölge efekti kazandırmak istersek "CharShadowed" adlı bir özellikten yararlanacağız:

```
>>> imlec.setPropertyValue("CharShadowed", True)
```

Bu özellik de True ve False olmak üzere iki farklı değer alır.

### 30.4.7 İçi Boş Karakterler (contour)

Eğer içi boş karakterler üretmek isterseniz "CharContoured" özelliğinden faydalanabilirsiniz:

```
>>> imlec.setPropertyValue("CharContoured", True)
```

Bu özellik de True ve False olmak üzere iki farklı değer alır.

## 30.5 Karakterleri Renklendirme

Yukarıda karakterleri nasıl biçimlendireceğimizi gördük. Şimdi ise bu karakterleri nasıl renklendireceğimize bakacağız. Bu bölümde "CharColor" ve "CharBackColor" olmak üzere, iki temel renklendirme özelliğinden bahsedeceğiz.



### 30.5.1 CharColor

"CharColor" özelliği, karakterlerin kendi renklerine işaret eder. Dolayısıyla bu özelliği kullanarak bir karakterin kendi rengini değiştirebiliriz. Bu özellik şöyle kullanılıyor:

```
>>> imlec.setPropertyValue("CharColor", renk)
```

Burada "renk" ifadesinin yerine uygun bir renk kodu yazmamız gerekiyor. OpenOffice onluk sayı biçimindeki renk kodlarını kabul eder. Normal olarak, renk kodları çoğu yerde karşımıza onaltılık (hexadecimal) sayı biçiminde çıkar. Dolayısıyla internet üzerindeki herhangi bir renk tablosundan alacağımız onaltılık sayı biçimindeki renk kodlarını OpenOffice'de kullanabilmek için öncelikle bunları onluk sayıya çevirmemiz gerekir.

Mesela renk kodlarını öğrenmek için <http://html-color-codes.info/> adresinden yararlanabiliriz. Bu adresteki renk paleti içinden herhangi bir renk seçtiğimizde bu rengin kodu "Selected color code is:" ifadesinin yanındaki kutucuk içinde gösterilecektir. Diyelim ki orada "#0174DF" kodunu gördük. Bu kodun "0174DF" kısmını alıp, bu onaltılık sayıyı onluk sayı sistemine çevirmemiz gerekiyor. Bunu kodlarımız içine şöyle uygulayabiliriz:

```
>>> imlec.setPropertyValue("CharColor", int("0174DF", 16))
```

Burada `int("0174DF", 16)` kodu bir onaltılık sayı olan "0174DF"yi onluk sayı sistemine çevirecektir. "0174DF" sayısının onluk sistemdeki karşılığı "95455"tir. Dolayısıyla isterseniz önce sayıyı onluk sisteme çevirip, ardından elde edilen sayıyı yukarıdaki koda eklemeyi de tercih edebilirsiniz:

```
>>> imlec.setPropertyValue("CharColor", 95455)
```

Dilerseniz bu kodları bağlam içinde gösterelim:

```
>>> import uno

>>> yerel = uno.getComponentContext()

>>> baglanti_servisi = yerel.ServiceManager.createInstanceWithContext(
... "com.sun.star.bridge.UnoUrlResolver", yerel)

>>> baglanti = baglanti_servisi.resolve(
... "uno:socket,host=localhost,port=2002;urp;StarOffice.ComponentContext")

>>> belge_servisi = baglanti.ServiceManager.createInstanceWithContext(
... "com.sun.star.frame.Desktop", baglanti)

>>> belge = belge_servisi.getCurrentComponent()

>>> imlec = belge.Text.createTextCursor()

>>> imlec.setPropertyValue("CharColor", int("0174DF", 16))

>>> belge.Text.insertString(imlec, u"renkli yazı", 0)
```

Bu kodları yazdığımızda, OpenOffice belgesi içindeki karakterlerin renklerinin seçtiğimiz renge göre değiştiğini göreceğiz...

### 30.5.2 CharBackColor

“CharBackColor” özelliğinin kullanımı “CharColor” özelliğinin kullanımına çok benzer. “CharBackColor” özelliği yardımıyla bir karakterin arkaplan rengini değiştiriyoruz:

```
>>> imlec.setPropertyValue("CharBackColor", int("BDBDBD", 16))
```

## 30.6 PyUno, Python ve OpenOffice Hakkında Bilgi Veren Kaynaklar

Python ve OpenOffice ilişkisi üzerine çalışmak isteyenler interneti araştırdıklarında bu konu üzerine yazılmış ayrıntılı ve açıklayıcı belgelerin çok az olduğunu görecekler. Eldeki kısıtlı kaynaklar da (tabii ki) İngilizce. Türkçe olarak bu konu üzerine bir şeyler bulmak pek mümkün değil. Benim bu belgeyi hazırlamaktaki amacım, en azından bu işe başlayacak olanlara yol gösterebilmektir.

Eğer internet üzerinde bu konuyla ilgili hangi kaynakların mevcut olduğunu merak ediyorsanız, birkaç tanesini hemen burada sıralayalım:

[0] [DevGuide](#)

[1] [udk.openoffice.org](http://udk.openoffice.org)

[2] [LucasManual](#)

---

# Python'da Paket Kurulumu - Kullanımı

---

---

**Not:** Bu yazı Sayın Samet Aras tarafından hazırlanmıştır.

---

Bu bölümde, Python'da paket kurulumu ve kullanımı anlatılacaktır.

Python için yazılmış paketler [The Python Package Index](#) sayfasında bulunur.

## Başlangıç

Python'un resmi sitesindeki [The Python Package Index](#) adlı sayfada şuan **7788** adet paket bulunmaktadır. Ben bir tane paket seçtim, bu paket üzerinden "Python'da Paket kurulumu - kullanımını" anlatacağım.

## Kurulum

```
sudo python setup.py --help
```

komutu ile paket hakkında bilgi edinebilirsiniz. Örneğin paket yazarını öğrenmek istersek:

```
sudo python setup.py --author
```

komutunu kullanabiliriz. Anlatımda kullanacağım paket Richard Jones tarafından yazılmış. Paket yazarının E-Posta adresini öğrenmek istersek şu komutu veriyoruz:

```
sudo python setup.py --author-email
```

Siz --help komutu ile diğer seçeneklere de bakabilirsiniz.

Anlatım için [simple, elegant HTML/XHTML generation](#) paketini yeğledim.

## Paket içeriği

```
setup.py    #kurulum dosyasıdır.  
html.py     #paket dosyasıdır.  
PGK-INFO    #bilgi dosyasıdır.
```

Şimdi kurulumu geçelim:

```
samet@samet:~$ cd /home/samet/Masaüstü/html-1.6
samet@samet:~/Masaüstü/html-1.6$ sudo python setup.py install
[sudo] password for samet:
running install
running build
running build_py
creating build
creating build/lib.linux-i686-2.6
copying html.py -> build/lib.linux-i686-2.6
running install_lib
copying build/lib.linux-i686-2.6/html.py -> /usr/local/lib/python2.6/dist-package
byte-compiling /usr/local/lib/python.2.6/dist-packages/html.py to html.pyc
running install_egg_info
Writing /usr/local/lib/python2.6/dist-packages/html-1.6.egg-info
samet@samet:~/Masaüstü/html-1.6$
```

Burada sudo komutu vermeliyiz. Zira yetki engelleri ile karşılaşabiliriz.

## Kullanım

Söz konusu paketi kurduk. Şimdi kullanıma değinelim:

```
>>> from html import HTML

>>> h = HTML()

>>> p = h.p('Merhaba dünya !\n')

>>> p.text('more → text', escape=False)

>>> h.p
<HTML p 0x9432b0c>

>>> print h
<p>Merhaba dünya !
more → text</p>
<p>
```

*from html import HTML* satırı ile, yüklediğimiz paketi dâhil ettik.

## Örnek

Artık Python'da paket kurulumu ve kullanımını biliyoruz. Şimdi yüklediğimiz paket ile bir Python programı yazalım. Sözgelimi, bir HTML dosyası oluşturalım, içerisine HTML paketimiz yardımıyla bir liste ekleyelim.

Bir Python dosyası oluşturalım:

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-
```

HTML modülünü dâhil edelim:

```
from html import HTML
```

Şimdi HTML modülünü **h** isimli bir nesneye atayalım:

```
h = HTML()
```

Şimdi HTML listemizi oluşturalım:

```
l = h.ul
l.li('C')
l.li('Python')
l.li("JavaScript")
print h
```

Yukarıda liste oluşturduk, ve içerisine 3 adet eleman ekledik (C, Python ve JavaScript). Kodları topluca görelim:

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-

from html import HTML

h = HTML()
l = h.ul
l.li('C')
l.li('Python')
l.li("JavaScript")
print h
```

Bildiğiniz gibi, <ul> HTML’de liste oluşturmaya, <li> ise eleman eklemeye yarıyor. Burada ilk elemanın C olmasının nedeni, Python’un da HTML gibi kodları okumaya yukarıdan başlamasıdır.

Şimdi bu programı biraz daha geliştirelim. Oluşturduğumuz listeyi ekrana yazdırmak yerine, bir HTML dosyası oluşturup içerisine ekleyelim:

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-

from html import HTML

h = HTML()
l = h.ul
l.li('C')
l.li('Python')
l.li("JavaScript")

dosya = open("liste.html", "w") # liste.html dosyamızı oluşturuyoruz
dosya.write("<html>\n")          # html başlangıç
dosya.write("<head>\n")          # head kısmı
dosya.write("</head>\n")         # head bitiş
dosya.write("<body>\n")          # body kısmı

dosya.write(str(h)+"\n")         # liste kısmı

dosya.write("</body>\n")         # body bitiş
dosya.write("</html>\n")         # html bitiş
dosya.close()                   # dosya kapatıldı
```

Yukarıda HTML modülü sayesinde bir HTML listesi oluşturduk ve “liste.html” isimli bir HTML dosyası açtık. Temel HTML komutlarını bizzat kendimiz ekledik. Akabinde:

```
dosya.write(str(h)+"\n")
```

Kodu ile HTML dosyamıza listemizi dâhil ettik. Artık Python’a modül dâhil etmeyi, kullanmayı ve program geliştirmeyi öğrendiniz. Bilgilerinizi pekiştirmek ve ilerletmek açısından, [The Python Package Index](#) bağlantısını sık sık incelemenizde fayda vardır. Sözgelimi, bu modülü kullanarak bir **HTML Code Generator** uygulaması yapabilirsiniz.

# reStructuredText

## 32.1 Giriş

Diyelim ki bir program yazdınız ve kullanıcılarınıza yol göstermek amacıyla, bu program için bir kullanma kılavuzu oluşturmak istiyorsunuz. Ya da benim yaptığım gibi, bir programlama dilini öğreten bir site işletiyor da olabilirsiniz. Eğer hazırladığınız belgeleri internet üzerinde yayımlayacaksanız bu belgeleri HTML biçiminde düzenlemeniz gerekir. İşte bu noktada, belgelendirme işlemi tam bir çileye dönüşür. Çünkü yazdığınız belgeleri internet üzerinde sergilenebilecek bir hale getirmek, asıl belgeleri hazırlamaktan daha vakit alıcıdır. Yani asıl yapmanız gereken işe ayıracağınız zamanı, siteyi yayımlanabilir hale getirmeye harcadığınızı farkedersiniz. Üstelik tek tek hazırladığınız HTML belgelerinin bakımını yapmak da bir hayli meşakkatlidir.

Bunun dışında, oluşturduğunuz belgeleri farklı biçimlerde yayımlamak istediğiniz zaman da büyük zorluklarla karşılaşsınız. Örneğin oluşturduğunuz HTML belgelerini daha sonra PDF olarak da okurlarınıza sunmak isterseniz omzunuzdaki yük daha da artacaktır.

İşte bu noktada “reStructuredText” ya da kısa adıyla “rst” devreye girer. Eğer belgelerinizi HTML veya herhangi başka bir biçimde hazırlamak yerine rst biçiminde hazırlarsanız, bu belgeleri daha sonra pek çok farklı biçime dönüştürmek epey kolay olacaktır. Yani bir kez rst biçimli belge hazırladıktan sonra bu belgeleri daha sonra isterseniz PDF, HTML ve ODT gibi biçimlere dönüştürebilirsiniz. Peki bu rst denen şey tam olarak nedir? Gelin isterseniz bu sorunun cevabını bulmaya çalışalım.

Bu soruyu çok kısa bir şekilde şöyle cevaplayabiliriz: reStructuredText bir metin işaretleme dili/sistemidir. Peki “metin işaretleme” ne demek? Şöyle düşünün: Diyelim ki elinizde dümdüz bir metin var. Mesela şöyle bir şey:

Python Programlama Dili

Python Kelimesinin Anlamı

Her ne kadar Python programlama dili ile ilgili çoğu görsel malzemenin üzerinde bir “piton” resmi görsek de, “Python” kelimesi aslında çoğu kişinin zannettiği gibi “piton yılanı” anlamına gelmiyor. “Python Programlama Dili”, ismini Guido Van Rossum’un çok sevdiği, Monty Python adlı altı kişilik bir İngiliz komedi grubunun Monty Python’s Flying Circus adlı gösterisinden alıyor.

Burada “Python Programlama Dili” bir üst başlıktır. “Python Kelimesinin Anlamı” ise bir alt başlık... Bunun altında ise asıl metin yer alıyor. Şimdi bu metni şu hale getirelim:

```
*****
Python Programlama Dili
*****
```

### Python Kelimesinin Anlamı

=====

Her ne kadar Python programlama dili ile ilgili çoğu görsel malzemenin üzerinde bir “piton” resmi görsek de, “Python” kelimesi aslında çoğu kişinin zannettiği gibi “piton yılanı” anlamına gelmiyor. “Python Programlama Dili”, ismini Guido Van Rossum’un çok sevdiği, Monty Python adlı altı kişilik bir İngiliz komedi grubunun Monty Python’s Flying Circus adlı gösterisinden alıyor.

İşte bu şekilde yukarıdaki metni reStructuredText adlı işaretleme dilini kullanarak “işaretlemiş” olduk. Burada kullandığımız işaretler “\*” ve “=”.

Elbette rst işaretleme dili sadece bu işaretlerden ibaret değildir. Düz bir metni özel işaretlerle işaretleyerek farklı biçimlerde metinler ortaya çıkarabiliriz. Ancak bir metni sadece bu tür işaretlerle işaretleme tek başına hiç bir işe yaramaz. Burada bizim ihtiyacımız olan şey bu işaretleri anlayacak özel bir programdır. Şöyle düşünün: Diyelim ki Python programlama dilini kullanarak şu kodları içeren bir dosya oluşturduunuz:

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-

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

Bu kodları içeren dosya bu haliyle hiç bir işe yaramaz. Sizin ihtiyacınız olan şey, bu dosyada yazılı kodları tanıyacak ve yorumlayacak özel bir programdır. Burada o özel program Python’dur. Dolayısıyla yukarıdaki kodların bir işe yarayabilmesi için sisteminizde Python’un kurulu olması gerekir... İşte aynen burada olduğu gibi, rst işaretleme dilini kullanarak oluşturduğunuz metinlerin de bir işe yarayabilmesi için bu rst işaretlerini anlayacak bir programa sahip olmalısınız. Piyasada rst işaretleme dilini anlayan farklı programlar bulunur. Biz bu bölümde bunlar içinde en gelişmişlerinden biri olan Sphinx adlı yazılımı kullanacağız. Ancak o kısma geçmeden önce isterseniz durumu biraz daha somutlaştırmak için bir uygulama yapalım.

Dediğimiz gibi, rst dilini anlayan farklı programlar vardır piyasada. Siz bunlardan birini edinin rst belgeleriyle çalışmaya başlarsınız. Ama bir de internet üzerinden hizmet veren bir rst servisi de bulunur. Bu servis yardımıyla oluşturduğunuz rst belgelerini çok hızlı bir şekilde test edebilirsiniz. Bu servise ulaşmak için <http://www.tele3.cz/jbar/rest/rest.html> adresine gidelim. Orada gördüğümüz kutucuğa yukarıda yazdığımız şu rst metnini yapıştıralım:

```
*****
Python Programlama Dili
*****
```

### Python Kelimesinin Anlamı

=====

Her ne kadar Python programlama dili ile ilgili çoğu görsel malzemenin üzerinde bir “piton” resmi görsek de, “Python” kelimesi aslında çoğu kişinin zannettiği gibi “piton yılanı” anlamına gelmiyor. “Python Programlama Dili”, ismini Guido Van Rossum’un çok sevdiği, Monty Python

adlı altı kişilik bir İngiliz komedi grubunun Monty Python's Flying Circus adlı gösterisinden alıyor.

Bu metni o kutucuğa yapıştırdıktan sonra alt tarafta görünen “render” düğmesine bastığımızda karşımıza yukarıdaki rst belgesinin HTML biçiminde yorumlanmış hali gelecek. Gördüğünüz gibi, yukarıda “\*” ile işaretlediğimiz kısım metnin ana başlığı, “=” ile işaretlediğimiz kısım ise metnin alt başlığı oldu.

Böylece bir rst metninin neye benzediği ve bu metnin yorumlanması sonucu nasıl bir şeyin ortaya çıktığı konusunda temel bir fikir edinmiş olduk. Artık reStructuredText'in derinliklerine dalabiliriz.

## 32.2 RestructuredText Dosyaları ve rst2html Betiği

Giriş bölümünde, yazdığınız bir rst belgesini test etmek için <http://www.tele3.cz/jbar/rest/rest.html> adresinden yararlanabileceğinizi söylemiştik. Ancak bu yol her zaman tercih edilmeyebilir. Mesela kendiniz rst belgelerini html belgelerine dönüştürmek isterseniz bu yöntem işinize yaramayacaktır. Bu bölümde size rst belgelerini nasıl kaydedip farklı biçimlere dönüştürebileceğinizi anlatmaya çalışacağım.

Öncelikle şunu söyleyeyim. Bir rst dosyası mutlaka .rst uzantısına sahip olmak zorunda değildir. rst belgelerinizi .txt uzantısıyla da kaydedebilirsiniz. Mesela yukarıdaki verdiğimiz örnek rst metnini “deneme.rst” veya “deneme.txt” adıyla bilgisayarınıza kaydedin. Yani elinizde, içeriği şu olan, “deneme.txt” veya “deneme.rst” adlı bir dosya olsun:

```
*****
Python Programlama Dili
*****
```

```
Python Kelimesinin Anlamı
=====
```

Her ne kadar Python programlama dili ile ilgili çoğu görsel malzemenin üzerinde bir “piton” resmi görsek de, “Python” kelimesi aslında çoğu kişinin zannettiği gibi “piton yılanı” anlamına gelmiyor. “Python Programlama Dili”, ismini Guido Van Rossum'un çok sevdiği, Monty Python adlı altı kişilik bir İngiliz komedi grubunun Monty Python's Flying Circus adlı gösterisinden alıyor.

Şimdi bu rst dosyasını bir html dosyasına dönüştüreceğiz. En başta da söylediğim gibi, rst belgeleri tek başına bir işe yaramaz. Bu belgeleri farklı biçimlere dönüştürebilmek için bazı özel programlardan yararlanmamız gerekir. Dediğim gibi, biz bu iş için Sphinx adlı bir yazılımdan faydalanacağız. Ama Sphinx bu alandaki tek dönüştürücü yazılım değildir. Bu yazılım, oldukça ayrıntılı ve yeteneklidir, ancak şu aşamada bizim için biraz ağır kaçabilir. O yüzde biz bu giriş bölümünde şimdilik “rst2html” adlı bir betikten yararlanacağız. Bu basit betik sayesinde, hiç başka şeylerle kafamızı karıştırmadan, reStructuredText denen şeye aşinalık kazanacağız. Peki biz bu “rst2html” adlı betiği nereden edineceğiz? Bu betik “docutils” adlı paketin bir bileşenidir. GNU/Linux kullanıcıları, dağıtımlarının paket yöneticisi aracılığıyla Docutils'i sistemlerine kurabilir. Eğer Ubuntu GNU/Linux'u kullanıyorsanız şu komutu vermeniz yeterli olacaktır:

```
sudo aptitude install python-docutils
```

Windows kullanıcıları ise <http://docutils.sourceforge.net/docutils-snapshot.tgz> adresinden, en son Docutils paketini indirebilir. Bu paketi indirdikten sonra, paketi açıp “docutils” klasörü içinde şu komutu vererek Docutils'i sisteminize kurabilirsiniz:



```
python setup.py install
```

Docutils paketini dağıtımlarının paket yöneticisinde bulamayan veya Docutils'in en son sürümünü kullanmak isteyen GNU/Linux kullanıcıları da yukarıdaki adresten indirdikleri Docutils paketini aynı komut yardımıyla sistemlerine kurabilir. Ancak GNU/Linux kullanıcıları yukarıdaki komutu yetkili kullanıcı olarak vermeye özen göstermeli...

Dediğim gibi, rst2html adlı betik Docutils paketinin bir bileşenidir. Dolayısıyla Docutils paketini sistemimize kurduktan sonra bu betiğe erişebileceğiz. Şimdi yukarıda oluşturduğumuz "deneme.txt" veya "deneme.rst" dosyasının bulunduğu klasöre giderek şu komutu veriyoruz:

```
rst2html deneme.txt > deneme.html
```

Tabii ben burada dosya adının "deneme.txt" olduğunu varsaydım. Eğer siz farklı bir dosya adı kullandıysanız yukarıdaki komutu ona göre düzenlemeniz gerekir.

Yukarıdaki komutu verdikten sonra, komutu verdiğimiz klasör içinde "deneme.html" adlı bir dosya oluştuğunu göreceğiz. Bu dosyayı herhangi bir internet tarayıcısı yardımıyla görüntüleyebilirsiniz. Bu dosyayı bir metin düzenleyici ile açarsanız, bunun standart bir html dosyası olduğunu göreceksiniz.

Oluşan bu html dosyasını internet tarayıcınız yardımıyla açtığınızda rst belgesindeki özel işaretlerin nasıl yorumlandığına dikkat edin. Gördüğünüz gibi "\*" işareti ile gösterilen kısım html belgesinde üst başlık olarak, "=" işareti ile gösterilen kısım ise alt başlık olarak yorumlanmış. Başlıkların altında yer alan işaretsiz kısım ise html belgesine normal bir paragraf olarak yerleştirilmiş.

## 32.3 Paragraf Stilleri

Bundan önceki bölümlerde reStructuredText'in yapısı hakkında bir-iki şey söyledik. Öğrendiğimiz gibi, bir rst belgesi, özel bazı işaretlerden oluşuyor. Mesela "\*" işaretinin üst başlıkları belirtmek için, "=" işaretinin ise alt başlıkları belirtmek için kullanıldığını öğrenmiştik. Ancak elbette reStructuredText bunlardan ibaret değildir. İşte biz de bu bölümde bir rst belgesi içinde hangi işaretlerin hangi amaçlara hizmet etmek üzere kullanılabileceğini tartışacağız.

Dilerseniz en baştan başlayalım:

### 32.3.1 Bölüm Başlıkları

reStructuredText biçimli metinleri oluştururken, bölüm başlıklarını göstermek için özel bazı işaretlerden yararlanıyoruz. Esasında bu işaretlerin ne olacağı tamamen size kalmış. Yani bu konuda çok sıkı bir zorlama yok. Buradaki en temel kural, bölüm başlıklarını gösterirken harf veya sayı kullanmamak. Yani - ' : ' " ~ ^ \_ \* + # < > işaretlerinden herhangi birini rahatlıkla kullanabilirsiniz. Önemli olan, kullandığınız işaretlerde tutarlı olmanızdır. Mesela biz yukarıda verdiğimiz örnekte "\*" işaretini üst başlıklar için, "=" işaretini ise alt başlıklar için kullandık. Ben "\*" işaretini kullanırken, metin içinde üst başlıklar daha kolay ayırt edilebilsin diye "\*" işaretini hem başlığın üst tarafına hem de alt tarafına yerleştirdim. Siz isterseniz bu işaretler yerine başkalarını da kullanabilirsiniz. Örneğin üst başlıkları belirtmek için "\_" işaretini, alt başlıkları göstermek için ise ":" işaretini tercih edebilirsiniz. Dediğim gibi, önemli olan tutarlı olmanız ve aynı seviyedeki başlıkları bütün metin boyunca aynı işaretle göstermenizdir. Ben bu durumlar için size şöyle bir şablon önerebilirim:

\*\*\*\*\*  
ÜST BAŞLIK  
\*\*\*\*\*

Alt Başlık  
=====

Alt alt-başlık  
^^^^^^^^^^^^

Elbette siz de isterseniz kendi göz zevkinize uygun başka bir şablon belirleyip her zaman onu kullanabilirsiniz. Mesela Docutils'in resmi sayfasında kullanılan başlık şablonunu inceleyerek bir fikir edinebilirsiniz. Sayfaya <http://docutils.sourceforge.net/index.txt> adresinden erişebilirsiniz. Docutils'in sayfasındaki html belgelerinin rst kaynaklarını görmek için, her sayfanın en altında yer alan "View document source" bağlantısını takip edebilirsiniz. İnternet üzerinde bulunan belgelerin rst kaynaklarını incelemek size bu belge biçimi hakkında önemli ipuçları verecektir.

rst belgelerindeki başlıklar hakkında söylenmesi gereken en önemli şeylerden biri, kullandığınız işaretlerin sayısıdır. Bir başlığın alt tarafına yerleştireceğiniz işaretler en az o başlıktaki karakter sayısı kadar olmalıdır. Yani şöyle bir kullanım hatalı olacaktır:

\*\*\*  
BAŞLIK  
\*\*\*

Burada üst ve alt tarafa en az altı adet "\*" işareti yerleştirmemiz gerekir:

\*\*\*\*\*  
BAŞLIK  
\*\*\*\*\*

### 32.3.2 Paragraflar

reStructuredText biçimli metinler yazarken paragrafları göstermek için özel herhangi bir işaret kullanmıyoruz:

\*\*\*\*\*  
Çilek  
\*\*\*\*\*

Çilek (Fragaria), gülgiller (Rosaceae) familyası içinde yer alan bir bitki cinsi ve bu cins içinde yer alan türlerin meyvelerinin ortak adıdır.

Dünyada, adlandırılmış 20'den fazla çilek türü vardır; ayrıca, çeşitli melezler ve kültürvarlar da bulunur. Dünya çapında ticari olarak en çok yetiştirilen çilekler, bahçe çileği olarak adlandırılan *Fragaria × ananassa* melezinin kültürvarlarıdır.

Çilekler, değerli C vitamini kaynağıdır.

Not: Bu metin <http://tr.wikipedia.org/wiki/%C3%87ilek> adresinden alınmıştır.

Gördüğünüz gibi, paragrafları göstermek için özel bir işaret kullanmıyoruz. Burada tek dikkat edeceğimiz şey her paragraf arasında bir boşluk bırakmaktır. Ayrıca başlığı yazdıktan sonra ilk paragrafı yazarken de, okunaklılık açısından başlıkla ilk paragraf arasında bir boşluk bırakmanızı tavsiye ederim.

Yukarıda verdiğimiz örneklerin html olarak nasıl görüldüğünü test etmek için <http://www.tele3.cz/jbar/rest/rest.html> adresindeki çevrimiçi servisten, ya da Docutils paketi içinde yer alan rst2html adlı betikten yararlanabileceğinizi biliyorsunuz.

### 32.3.3 Yana yatık (italic) Metinler

rst metinlerinde kelimeleri yana yatık harflerle yazabilmek için yine “\*” işaretinden yararlanacağız. Mesela şu örneğe bir bakalım:

```
*Python* kıvrak bir programlama dilidir.
```

Gördüğünüz gibi, “Python” kelimesini “\*” işaretleri arasına aldık. Böylece bu kelime, mesela html’ye dönüştürüldüğünde yana yatık olarak görünecektir.

### 32.3.4 Kalın (bold) Metinler

Eğer bir paragraf içindeki herhangi bir kelimeyi kalın harflerle göstermek isterseniz yine “\*” işaretinden yararlanabilirsiniz. Ancak bu defa bu işareti çift olarak kullanacağız:

```
**Python** ve **Django** iyi bir ikilidir.
```

Bu metni html biçimine dönüştürdüğünüzde “Python” ve “Django” kelimelerinin kalın yazıldığını göreceksiniz.

### 32.3.5 Vurgulu Metinler

Eğer bir kelimeyi ve kelime grubunu, öteki kelimelerden farklı olarak vurgulu bir biçimde göstermek isterseniz “'''” işaretinden yararlanabilirsiniz:

```
‘‘Python’’ ve ‘‘Django’’ iyi bir ikilidir.
```

## 32.4 Listeler

reStructuredText dilini kullanarak liste oluşturmak oldukça kolaydır. Biz bu bölümde listeleri “temel listeler” ve “özel listeler” olarak iki başlık altında inceleyeceğiz.

### 32.4.1 Temel Listeler

Temel listeler, sıkça kullanmamız gerekebilecek bilindik liste türleridir. “numaralı listeler” ve “numarasız listeler” bu sınıfa girer. Hemen ufak bir örnek verelim:

```
* Elma
* Armut
* Çilek
* Erik
```

Burada “\*” işaretini kullanarak numarasız bir liste oluşturduk. İsterseniz “\*” işareti yerine “+” veya “-” işaretlerini de kullanabilirsiniz. Liste oluştururken dikkat etmemiz gereken şey, işaret ile öge arasında bir boşluk bırakmaktır. Yani mesela “\*” işareti ile “Elma” ögesi arasında bir boşluk bıraktığımızdan emin olmalıyız.

Eğer amacımız numarasız listeler değil de numaralı listeler oluşturmaksa yine bunu çok basit bir şekilde halledebiliriz:

1. Elma
2. Armut
3. Çilek
4. Erik

Burada numaralı bir liste oluşturmak için sayılardan yararlandık. Aynı şekilde büyük ve küçük harfleri kullanmak da mümkündür:

- A. Elma
- B. Armut
- C. Çilek
- D. Erik

Veya:

- a. Elma
- b. Armut
- c. Çilek
- d. Erik

Hatta:

- i. Elma
- ii. Armut
- iii. Çilek
- iv. Erik

Dikkat ederseniz yukarıda numaralandırma işaretlerinin sağına birer adet nokta koyduk. Siz isterseniz nokta yerine "(" veya ")" işaretlerini de kullanabilirsiniz:

- 1) Elma
- 2) Armut

Veya:

- (1) Elma
- (2) Armut

reStructuredText ile numaralı liste oluşturmanın bir yolu da "#." işaretlerini kullanmaktır:

- #. Elma
- #. Armut
- #. Çilek
- #. Erik

Yukarıdaki rst kodlarının çıktısı şöyle olacaktır:

1. Elma
2. Armut
3. Çilek
4. Erik

Yukarıda gösterilen numaralı ve numarasız listelerin yanısıra, dilerseiz tanım listeleri de oluşturabilirsiniz. Tanım listesinin ne olduğunu isterseniz bir örnekle görmeye çalışalım:

```
Ubuntu
    Debian tabanlı bir GNU/Linux dağıtımıdır

Mint
    Ubuntu tabanlı bir GNU/Linux dağıtımıdır

Windows
    Microsoft firmasının, özgür olmayan bir ürünüdür
```

Sanırım bu örnek “tanım listesi”nin ne demek olduğunu açıklıyor... Bazı çalışmalarınızda bu liste türünün oldukça işinize yaradığını göreceksiniz. Bu arada, yukarıdaki listedeki başlıkların kalın olması isterseniz bu örneği şöyle de yazabilirsiniz:

```
**Ubuntu**
    Debian tabanlı bir GNU/Linux dağıtımıdır

**Mint**
    Ubuntu tabanlı bir GNU/Linux dağıtımıdır

**Windows**
    Microsoft firmasının, özgür olmayan bir ürünüdür
```

## 32.4.2 Özel Listeler

Bu liste türü, adından da anlaşılacağı gibi, özel bir amaca hizmet eder. Mesela bir programın kullanım kılavuzu içinde gördüğümüz seçenek listeleri; program hakkında yazar, sürüm, lisans v.b bilgiler veren listeler hep bu sınıfa girer. İlk olarak seçenek listelerinden başlayalım. Mesela Python’u `python -h` komutuyla başlattığımızda çıkan seçenek listesinin bir bölümünü örnek olarak alalım:

```
-B      don't write .py[co] files on import; also PYTHONDONTWRITEBYTECODE=x
-c cmd  program passed in as string (terminates option list)
-d      debug output from parser; also PYTHONDEBUG=x
-E      ignore PYTHON* environment variables (such as PYTHONPATH)
-h      print this help message and exit (also --help)
-i      inspect interactively after running script; forces a prompt even
        if stdin does not appear to be a terminal; also PYTHONINSPECT=x
-m mod  run library module as a script (terminates option list)
```

Gördüğünüz gibi, seçenekleri ve bu seçeneklerin açıklamalarını, birbirlerinden boşluklarla ayırarak yerleştirdik. Eğer isterseniz seçenek tanımlarını alt satıra da alabilirsiniz:

```
-B
    don't write .py[co] files on import; also PYTHONDONTWRITEBYTECODE=x
-c cmd
    program passed in as string (terminates option list)
-d
    debug output from parser; also PYTHONDEBUG=x
-E
    ignore PYTHON* environment variables (such as PYTHONPATH)
-h
    print this help message and exit (also --help)
-i
    inspect interactively after running script; forces a prompt even
    if stdin does not appear to be a terminal; also PYTHONINSPECT=x
```

```
-m mod  
run library module as a script (terminates option list)
```

Burada seçenek açıklamalarını alt satıra aldıktan sonra, tanımı, ait olduğu seçeneğe göre girintili yazdığımıza dikkat edin. Bu yapı, özellikle uzun seçenek adlarının bulunduğu listelerde görünüş açısından tutarlılık elde etmenizi sağlayabilir.

Özel listeler kategorisinde değerlendirilebilecek bir başka liste türü de “künye listeleri”dir. Örneğin yazdığınız bir programda kullanıcılarınıza yazar adı, sürüm numarası, yayımlanma tarihi gibi künye bilgilerini derli toplu bir biçimde vermek isterseniz bu liste türünden yararlanabilirsiniz:

```
:Program Adı: HARMAN  
:Yazarı: Fırat Özgül  
:Sürüm: 0.8  
:Dil: Türkçe, İngilizce
```

## 32.5 Köprüler/Bağlantılar

Yazdığınız belgelerin pek çok yerinde başka belgelere veya adreslere bağlantılar vermek isteyeceksiniz. Bu işlemi reStructuredText ile yapmak oldukça kolaydır.

Biz burada, bir metne farklı şekillerde nasıl bağlantı ekleyebileceğinizi göstereceğiz.

### 32.5.1 Doğrudan Bağlantı

Eğer metnimize bir web sitesinin bağlantısını doğrudan eklemek isterseniz şu yöntemi kullanabilirsiniz:

```
Python, Guido Van Rossum adlı Hollandalı bir programcı tarafından yazılmış bir  
programlama dilidir. Eğer Guido Van Rossum’un neye benzediğini merak ediyorsanız,  
onun http://www.python.org/~guido/pics.html adresindeki fotoğraflarını inceleyebilirsiniz.
```

Gördüğünüz gibi, özel hiçbir işaret kullanmaya gerek olmadan, doğrudan web adresini metne ekleyerek amacımıza ulaşabiliyoruz. Yalnız reStructuredText biçimli metinlere bu şekilde bağlantı eklerken “<http://>” önekini yazmayı unutmuyoruz. Aksi halde bağlantı çalışmayacaktır...

### 32.5.2 Gizli Bağlantı

Bağlantıyı doğrudan görünür kılmak yerine bir kelime veya kelime grubunun altına da gizlemek isteyebilirsiniz. O zaman şöyle bir şey yazmanız gerekir:

```
Eğer Guido Van Rossum’un neye benzediğini merak ediyorsanız,  
onun ‘şu <http://www.python.org/~guido/pics.html>’_ adresteki  
fotoğraflarını inceleyebilirsiniz.
```

Burada şöyle bir yapı kullandık: ‘bağlantı metni <bağlantı adresi>’\_. Bu yapıda kullanılan işaretlere çok dikkat ediyoruz. Formülde gördüğümüz ‘ işaretinin kesme işareti (‘) olmadığına da özellikle dikkat edin. Bu “çentik” işareti genellikle klavyede virgül işaretinin olduğu tuşun üzerindedir. Bu işareti üretmek için ALT + çentik tuşlarına basıyoruz. Bazı sistemlerde bu işareti çıkarabilmek için ALT + çentik tuşlarına art arda iki kez basmanız gerekebilir. Eğer bu şekilde iki adet çentik işareti oluşuyorsa birini silersiniz olur biter!..

Yukarıda gösterilen yöntem dışında, metin altına bağlantı eklemenin farklı yolları da vardır. Örneğin:

```
Arama yapmak için 'tıklayınız'_  
.. _tıklayınız: http://www.google.com
```

Veya daha basit bir şekilde:

```
Arama yapmak için 'tıklayınız'__  
__ http://www.google.com
```

Şu son iki örnek, görüntü açısından daha derli toplu reStructuredText metinleri hazırlamanıza yardımcı olabilir. Mesela içinde birden fazla bağlantı içeren paragrafları şu şekilde yazarak kendi açınızdan daha düzenli ve tertipli bir rst belgesi elde edebilirsiniz:

```
Bu konu ile ilgili daha ayrıntılı bilgilere 'şuradan'__,  
'şuradan'__ ve 'şuradan'__ erişebilirsiniz.
```

Ayrıca bu konuya ilişkin çalışmalarınızda yukarıdakilerin dışındaki kaynakları da incelemenizi tavsiye ederim.

```
__ http://www.google.com  
__ http://www.yahoo.com  
__ http://www.hotmail.com
```

Burada gördüğünüz gibi, “şuradan” kelimesi birden fazla kullanıldığında, kelimelerin yazılış sırası dikkate alınıyor.

reStructuredText metinlerinde bağlantı vermenin değişik yolları olsa da siz muhtemelen yukarıda verilen ilk iki yöntemi kullanmayı tercih edeceksiniz. Ancak elbette bu yöntemler içinde size en kolay gelen yöntemi seçmekte serbestsiniz...

## 32.6 Değişkenler

Bazen bir metin hazırlarken, özel bir kelime veya kelime grubunu değişken olarak kullanmak isteyebilirsiniz. Bunun şöyle bir faydası vardır: Diyelim ki yazdığınız metnin pek çok yerinde aynı web sitesine göndermede bulunuyorsunuz. Bu web sitesinin adresini her defasında elle yazmak yerine, bunu bir değişkene atayıp metin içindeki gerekli yerlerle kullanmak isteyebilirsiniz:

```
Bu konuda yazılmış en iyi kaynak |web sitesi| adresinde bulunur.  
Ancak |web sitesi| adresinden yararlanabilmek için bu siteye üye  
olmanız gerekmektedir.
```

```
|web sitesi| sitesinin sahipleri, inceleme yaptığınız konuda  
size ellerinden geldiğince yardımcı olmaya çalışacaklardır.
```

```
.. |web sitesi| replace:: http://www.falanca.com
```

Burada replace adlı özel bir ifadeden yararlandığımıza dikkat edin. Ayrıca kullanılan öteki işaretlere de özellikle dikkatinizi çekmek isterim. Bu yöntemin, işlerinizi ne kadar kolaylaştıracığını tahmin edebilirsiniz. Yukarıdaki metinde geçen [www.falanca.com](http://www.falanca.com) adresini sadece tek bir yerde değiştirerek, bu değişikliğin bütün metne uygulanmasını sağlayabilirsiniz.

Yukarıdaki gibi, metin değişkenleri kullanabilmenin yanısıra, reStructuredText bize resimleri de aynı şekilde kullanma imkanı sağlar. Bunun için replace yerine image adlı bir ifadeden yararlanacağız:

```
|dikkat| simgesi, programı kullanırken dikkat etmeniz gereken şeyleri gösterir.
```

```
.. |dikkat| image:: dikkat.jpg
```

## 32.7 Kod Alanları

reStructuredText'i kod içeren belgeleri hazırlamak için kullanacağınızı düşünürük, işinize bir hayli yarayacak bir özellikten bahsetmemizde yarar var. Şu örneğe bir bakalım:

Tkinter kullanarak yazılabilecek örnek bir program şöyle olabilir::

```
#-*-coding: utf-8 -*-
from Tkinter import*

class Uygulama(object):

    def __init__(self):
        self.gui_pen_ar()

    def gui_pen_ar(self):
        self.label = Label(text = "Programımıza hoşgeldiniz...")
        self.label["bg"] = "light yellow"
        self.label["font"] = "helvetica 15 bold"
        self.label["relief"] = GROOVE
        self.label.pack(pady = 10, padx = 10)

        for k, v in [("Tamam", None), ("Çık", pencere.quit)]:
            self.btn = Button()
            self.btn["text"] = k
            self.btn["command"] = v
            self.btn["width"] = 10
            self.btn["font"] = "verdana 13 italic"
            self.btn["relief"] = RIDGE
            self.btn.pack(side = LEFT,
                          padx = 20,
                          pady = 20)

pencere = Tk()
uyg = Uygulama()
pencere.title("Merhaba")
pencere.mainloop()
```

Eğer yukarıdaki örneği rst2html betiği yardımıyla HTML'ye dönüştürecek olursanız, kod alanının metnin geri kalanından sık bir şekilde ayrıldığını ve gri bir arkaplan rengiyle gösterildiğini göreceksiniz. İlerde Sphinx yazılımını kullandığımız zaman buradaki kodların daha zarif bir şekilde renklendiğine de şahit olacağız...

Burada dikkat etmemiz gereken en önemli şey kod bloğunun başlayacağını gösteren :: işaretidir. Tkinter kullanarak yazılabilecek örnek bir program şöyle olabilir cümlesinin ardından :: işaretini koyarak alt satıra geçtikten sonra kodlarımızı girintili olarak yazı-



oruz. Buradaki :: işareti, HTML çıktısında cümle sonuna bir adet de iki nokta üst üste işareti yerleştirecektir. Eğer siz iki nokta üst üste işareti yerine mesela noktalı virgül koymak isterseniz ilk cümleyi şöyle yazabilirsiniz:

```
Tkinter kullanarak yazılabilecek örnek bir program şöyle olabilir; ::
```

Burada ";" işaretinden sonra getirdiğimiz ":" işaretini noktalı virgülden ayrı yazdığımıza dikkat edin. Ayrıca bu cümleyi yazdıktan sonra, kodlarımızı yazmaya başlamadan önce bir satır boşluk bırakmayı da unutmuyoruz...

## 32.8 Dipnotları

Yazdığınız metinlere dipnotlarını eklemek için şöyle bir yol izleyebilirsiniz:

```
2009 yılına ait verilere göre [#]_ Python programlama dili yükseliştedir.
```

```
.. [#] http://www.falanca.com
```

Eğer dipnotlarını elle numaralandırmak isterseniz yukarıdaki kodları şöyle yazmanız gerekir:

```
2009 yılına ait verilere göre [2]_ Python programlama dili yükseliştedir.
```

```
.. [2] http://www.falanca.com
```

## 32.9 Resimler

Yazdığınız metinlerde resim gösterebilmek için şöyle bir yapıdan faydalanıyoruz:

```
.. image:: resim.jpeg
```

Eğer arzu ederseniz resmin ekranda nasıl görüneceğini belirlemek için bazı seçeneklerden de yararlanabilirsiniz:

```
:height: 100px
:width: 200 px
:scale: 50 %
:alt: resmi tanımlayan bir metin
:align: right
```

Burada "height" resmin boyunu, "width" resmin enini, "scale" resmin ölçeğini, "alt" resmi tanımlayan bir metni, "align" ise resmin ne tarafa hizalanacağını gösteriyor. Eğer resminizi sağa yaslamak isterseniz :align: değerini "right", sola yaslamak isterseniz "left", ortalamak isterseniz "center" şeklinde belirleyebilirsiniz. Sonuç olarak, bir metin içindeki resmi göstermek için şu biçimi kullanıyoruz:

```
.. image:: picture.jpeg
   :height: 100px
   :width: 200 px
   :scale: 50 %
   :alt: resmi tanımlayan bir metin
   :align: right
```

Burada elbette bütün seçenekleri kullanmak zorunda değilsiniz. İsterseniz varsayılan değerleri de kabul edebilirsiniz.

## 32.10 Yorumlar

Eğer yazdığınız metinlere yorum eklemek isterseniz şöyle yapmanız gerekir:

```
.. Bu bir yorumdur.
   Bu da bir yorumdur.
   Bu da...
```

.. işaretinden sonra alt satırda girintili olarak gösterdiğiniz her şey bir yorum olarak değerlendirilecek, mesela HTML çıktısında görünmeyecektir.

Böylece Sphinx adlı yazılımı rahatlıkla kullanmamızı sağlayacak kadar reStructuredText bilgisi edinmiş olduk. Şimdi gönül rahatlığıyla Sphinx'i anlatmaya başlayabiliriz.

## 32.11 Alıntılar

Hazırladığınız belgelerde başka bir yazardan veya metinden alıntı yapmak isterseniz şöyle bir yapı kullanabilirsiniz:

```
Ana paragraf buradan başlıyor. Hemen altta ise
alıntı bir metin yer alıyor.
```

```
    Alıntı metin burada başlıyor.
    Alıntı metnin devamı...
```

```
Alıntı metin sona erdi. Artık ana paragraftayız.
```

Yukarıdaki rst metninin HTML çıktısı ise şöyle olacaktır:

Ana paragraf buradan başlıyor. Hemen altta ise alıntı bir metin yer alıyor.

Alıntı metin burada başlıyor. Alıntı metnin devamı...

Alıntı metin sona erdi. Artık ana paragraftayız.

## 32.12 Tablolar

reStructuredText'in en zor, daha doğrusu en sıkıcı yanlarından birisi de tablo oluşturmaktır. Yazacağımız belgelere bir tablo eklemek için tabloyu elle çizmemiz gerekiyor!

+	+	+	+	+
	İSİM		SOYİSİM	
	ADRES		MESLEK	
+	+	+	+	+
	Ferhat		Gider	
	İstanbul		Yazar	
+	+	+	+	+
	Ahmet		Erden	
	Ankara		Çevirmen	
+	+	+	+	+
	Mehmet		Artır	
	İzmir		Çizer	
+	+	+	+	+
	Kezban		Güler	
	Adana		Mühendis	
+	+	+	+	+
	Selin		Güleç	
	Bursa		Öğretmen	
+	+	+	+	+
	Selim		Gelir	
	İzmit		Esnaf	
+	+	+	+	+

Bu çizim, HTML'ye dönüştürüldüğünde şöyle bir çıktı verecektir:

İSİM	SOYİSİM	ADRES	MESLEK
Ferhat	Gider	İstanbul	Yazar
Ahmet	Erden	Ankara	Çevirmen
Mehmet	Artır	İzmir	Çizer
Kezban	Güler	Adana	Mühendis
Selin	Güleç	Bursa	Öğretmen
Selim	Gelir	İzmit	Esnaf

Yukarıdaki tabloda, istediğiniz görünümü elde etmek için çizgilerin yerlerini değiştirebilir veya gereksiz gördüğünüz çizgileri silebilirsiniz.

reStructuredText biçimli belgelerde yukarıdaki gibi tabloları daha zahmetsiz bir biçimde oluşturmak isterseniz [rstab modülü](#)’nden yararlanabilirsiniz.

---

# Sphinx

---

Bir önceki bölümde reStructuredText'in ne olduğundan, neye benzediğinden ve nasıl kullanıldığından bahsettik. Hatırlarsanız o bölümde rst2html adlı bir betikten de söz etmiştik. Bu betik yardımıyla rst biçiminde oluşturduğumuz belgeleri rahatlıkla HTML'ye çevirebiliyoruz. Ancak rst2html ile üretilen HTML dosyaları görünüş olarak oldukça sadedir. Pek çok kimse rst2html ile üretilen HTML sayfalarını yavan ve çirkin bulabilir. Eğer siz de böyle düşünüyorsanız, rst2html'ye kıyasla çok daha cazip çıktılar üretebilen yazılımlardan faydalanabilirsiniz.

Bir önceki bölümde de bahsettiğimiz gibi, rst2html betiği, reStructuredText biçiminde hazırlanmış metinleri ayıklayarak bunlardan HTML belgeleri üretebilen bir uygulamadır. Ancak aynı işi çok daha yetenekli ve şık bir biçimde yapan başka programlar da bulunur. İşte bu bölümde inceleyeceğimiz Sphinx adlı yazılım da bunlardan biridir.

## 33.1 Sphinx Hakkında

Dilerseniz Sphinx'i kullanmaya başlamadan önce biraz Sphinx'ten bahsedelim. Nedir bu Sphinx?

Dediğimiz gibi Sphinx, reStructuredText biçiminde yazılmış metinleri yorumlayıp, bu metinlerden HTML ve PDF gibi farklı biçimlerde belgeler üreten oldukça yetenekli bir yazılımdır.

Sphinx, Python topluluğunun da aktif bir üyesi olan Georg Brandl tarafından Python Programlama Dili'ne ait resmi belgelendirme çalışmalarını düzene sokmak için yazılmıştır. Dolayısıyla bu yazılımın Python diline desteği oldukça kuvvetlidir. Ancak bu, Sphinx'le sadece Python belgelendirme çalışmaları yapılabilir anlamına gelmiyor. <http://sphinx.pocoo.org/examples.html> adresinde de göreceğiniz gibi, bugün halihazırda Sphinx'i kullanan pek çok proje vardır. Bu sayfada verilen projeleri inceleyerek Sphinx'le neler yapabileceğiniz konusunda iyi bir fikir edinebilirsiniz. Tahmin edebileceğiniz gibi, <http://docs.python.org/> adresinde bulunan Python belgeleri de Sphinx ile hazırlanıyor. Bunun yanı sıra, ben de [istihza.com](http://istihza.com)'daki bütün belgeleri bu Sphinx adlı yazılım yardımıyla hazırlıyorum.

Sphinx'in resmi sitesi <http://sphinx.pocoo.org/> adresindedir. Sphinx'e ilişkin en detaylı bilgiye bu adresten ulaşabilirsiniz.

## 33.2 Sphinx Nasıl Kurulur?

Elbette Sphinx'le herhangi bir şey yapabilmek için öncelikle bu yazılımı bilgisayarımıza kurmamız gerekiyor. Biz bu bölümde Sphinx'i GNU/Linux ve Windows işletim sistemlerine nasıl kuracağımızı inceleyeceğiz. Dilerseniz önce GNU/Linux'tan başlayalım.

### GNU/Linux

GNU/Linux sistemlerinde Sphinx'i kurmak için iki yöntem var. Önce kolay olanından başlayalım:

Sphinx pek çok GNU/Linux dağıtımının depolarında bulunur. Dolayısıyla Sphinx'i, kullandığınız dağıtımın paket yöneticisi aracılığıyla bilgisayarınıza kurabilirsiniz. Örneğin Ubuntu kullanıyorsanız şu komut yardımıyla Sphinx'i kurabilirsiniz:

```
sudo aptitude install python-sphinx
```

Siz de kendi dağıtımınızın paket yöneticisinde “sphinx” kelimesiyle bir arama yaparak, kuracağınız paketin hangisi olduğunu öğrenebilirsiniz.

Ancak bu yöntemin bir dezavantajı vardır. GNU/Linux dağıtımlarının depolarında bulunan Sphinx en yeni sürüm olmayabilir. Eğer Sphinx'in en son sürümünü kullanmak isterseniz Sphinx'i sitesinden indirip kaynaktan kurmayı da tercih edebilirsiniz. Ancak Sphinx'in bağımlı olduğu bazı ilave paketler vardır:

- Setuptools
- Docutils
- Jinja2
- Pygments

Sphinx'i kurmadan önce en azından Setuptools'u sisteminize kurmuş olmanız lazım. Setuptools'u kurmak için şunları yapıyoruz:

1. <http://pypi.python.org/pypi/setuptools> adresine gidiyoruz,
2. [Downloads](#) bölümünden *setuptools-x.y.tar.gz* adlı dosyayı bilgisayarımıza indiriyoruz,
3. İndirdiğimiz sıkıştırılmış dosyayı açıyoruz ve dizin içinde şu komutu veriyoruz.

```
sudo python setup.py install
```

Böylece Setuptools'u sistemimize kurmuş olduk. Öteki üç bağımlılığı kurmasanız da olur, çünkü Sphinx kurulurken o üç bağımlılık da otomatik olarak kurulacaktır.

**Not:** Eğer Sphinx'i internete erişimi olmayan bir bilgisayara kurmak isterseniz, Sphinx paketinden önce, Setuptools, Docutils, Pygments ve Jinja2 paketlerinin hepsini indirip kurmalısınız. Bu paketlerin nereden indirilip nasıl kurulacağını öğrenmek için aşağıdaki “Windows” başlığını inceleyebilirsiniz.

Şimdi Sphinx'i nasıl kuracağımıza bakalım...

Bunun için şu adımları takip ediyoruz:

1. Öncelikle <http://pypi.python.org/pypi/Sphinx> adresine gidiyoruz,
2. Buradan *Sphinx-0.x.x.tar.gz* adlı dosyayı bilgisayarımıza indiriyoruz.
3. İndirdiğimiz sıkıştırılmış dosyayı açıyoruz ve dizin içinde şu komutu veriyoruz.

```
sudo python setup.py install
```

Sphinx kurulurken, eğer sisteminizde Docutils, Jinja2 ve Pygments adlı yazılımlar yoksa, bunlar otomatik olarak tek tek indirilip kurulacaktır.

Böylece Sphinx'i kaynaktan kurmuş olduk.

Eğer Sphinx'i paket yöneticiniz aracılığıyla kuracaksanız, paket deponuzdaki Docutils sürümüne dikkat edin. Deponuzdaki sürüm 0.6'dan küçükse bir sorunuz var demektir. Çünkü Docutils'in 0.6 öncesi sürümleri Türkçe karakterlerle iyi anlaşıyor. Bu yüzden, Docutils'in 0.6'dan küçük bir sürümünü kullanırsanız, Sphinx ile ürettiğiniz HTML belgelerindeki URL adreslerinde yer alan Türkçe karakterler görünmeyecektir. Mesela ben, Ubuntu'daki Docutils sürümü 0.6'dan küçük olduğu için Docutils'i kaynaktan kurarak kullanıyorum.

Eğer Sphinx kurulumu ile ilgili herhangi bir sorun yaşarsanız veya bu konuda herhangi bir sorunuz olursa bana **kistihza[at]yahoo[nokta]com** adresinden ulaşabilirsiniz.

## Windows

Sphinx'i Windows'a kurabilmek için sistemimizde bazı programların halihazırda kurulu olması gerekiyor. Sphinx'in bağımlı olduğu programları şöyle sıralayabiliriz:

1. **Python.** Elbette Sphinx'in en önemli bağımlılığı Python'dur. Eğer sisteminizde Python kurulu değilse Sphinx'i de kullanamazsınız. Python'u nasıl kuracağınızı öğrenmek için [Temel Bilgiler](#) bölümüne bakabilirsiniz.
2. **Setuptools.** Sphinx'in Python'dan sonraki en önemli bağımlılığı Setuptools'tur. Bu programı kurduktan sonra, Sphinx'in öteki bağımlılıklarını kurmadan da Sphinx'i kurabilirsiniz. Çünkü bu program sayesinde Sphinx, ihtiyacı olan programları internetten otomatik olarak indirip kurabilecektir. Kullandığınız Python sürümüne uygun olan .exe dosyasını <http://pypi.python.org/pypi/setuptools> adresinden indirebilirsiniz (Mesela `setuptools-0.6c11.win32-py2.6.exe`)
3. Eğer bilgisayarınız internete bağlıysa bu aşamada yukarıdakilere ilave olarak sadece **Sphinx** paketini indirip kurmanız yeterli olacaktır. Sphinx paketini edinmek için <http://pypi.python.org/pypi/Sphinx> adresindeki .tar.gz uzantılı dosyayı bilgisayarınıza indirebilirsiniz. Bu dosyayı indirdikten sonra WinRar gibi bir program yardımıyla paketin içeriğini dışarı çıkarın ve program dizini içinde şu komutu verin:

```
python setup.py install
```

Bu komutun ardından, eğer her şey yolunda gittiyse Sphinx bilgisayarınıza kurulmuş olacaktır. Ama eğer bilgisayarınızda internet bağlantınız yoksa ve niyetiniz internet bağlantısı olan bir bilgisayar aracılığıyla gerekli bütün paketleri indirip internetsiz bilgisayarınıza kurmaksa, o zaman Python ve Setuptools'tan sonra üç program daha indirmeniz gerekiyor:

1. **Docutils.** Bu programı <http://pypi.python.org/pypi/docutils> adresinden indirebilirsiniz. Bu programı kurmak için, indirdiğimiz sıkıştırılmış dosyayı açıp dizin içinde şu komutu veriyoruz:

```
python setup.py install
```

2. **Pygments.** Bu programı <http://pypi.python.org/pypi/Pygments> adresinden indirebilirsiniz. Bu programı kurmak için, indirdiğimiz sıkıştırılmış dosyayı açıp dizin içinde şu komutu veriyoruz:

```
python setup.py install
```

3. **Jinja2.** Bu programı <http://pypi.python.org/pypi/Jinja2> adresinden indirebilirsiniz. Bu programı kurmak için, indirdiğimiz sıkıştırılmış dosyayı açıp dizin içinde şu komutu veriyoruz:

```
python setup.py install
```

4. Yukarıdaki paketleri kurduktan sonra, biraz önce gösterdiğimiz şekilde Sphinx paketini de kurarak işinizi bitirebilirsiniz.

Ancak Sphinx'in çalışabilmesi için yukarıdaki işlemler yeterli değildir. Bunun için sistemimizdeki `C:\Python26` ve `C:\Python26\Scripts` dizinlerinin de sistem yoluna (PATH) eklenmiş olması lazım. Eğer zaten bu dizinleri yola eklemişseniz ilave bir şey yapmanıza gerek yok. Eğer bu dizinleri YOL'a eklemediyseniz ve nasıl ekleyeceğinizi de bilmiyorsanız <http://www.istihza.com/py2/windows-path.html> adresindeki makalemizden yararlanabilirsiniz. Unutmayın, Sphinx'in çalışabilmesi için hem `C:\Python26` hem de `C:\Python26\Scripts` dizinlerinin YOL'a eklenmiş olması gerekir.

Sphinx kurulumu konusundaki sorularınız için **kistihza[at]yahoo[nokta]com** adresinden bana ulaşabilirsiniz.

### 33.3 Sphinx Nasıl Kullanılır?

Sphinx'i başarıyla kurduğumuza göre artık bu programı nasıl kullanacağımızı incelemeye başlayabiliriz.

Sphinx'i çalıştırmadan önce bazı ön hazırlıklar yapmamız gerekiyor. Öncelikle proje dosyalarımızı barındıracak bir dizin oluşturmamız gerekiyor. Mesela projemizin adı "python3" olsun. Şimdi bilgisayarımızda *python3* adlı bir dizin oluşturalım ve bu boş dizinin içine girelim. Artık Sphinx'i çalıştırmaya hazırız.

Sphinx programının grafik bir arayüzü yoktur. Bu yüzden Sphinx'i komut satırından çalıştıracğız. Şimdi kullandığımız sisteme uygun bir şekilde, *python3* adlı dizinin içinde komut satırını açalım ve orada şu komutu verelim:

```
sphinx-quickstart
```

Bu komutu verdiğimizde karşımıza şöyle bir ekran gelecek:

```
Welcome to the Sphinx quickstart utility.
```

```
Please enter values for the following settings (just press Enter to
accept a default value, if one is given in brackets).
```

```
Enter the root path for documentation.
```

```
> Root path for the documentation [.]:
```

Bu ekranda Sphinx bize bazı sorular soracak. Hazırlayacağımız projenin temel bazı özelliklerini bu ekranda karşımıza gelecek soruları cevaplayarak oluşturacağız. Her bir sorudan sonra ":" işaretini gördüğümüz yere bazı değerler gireceğiz.

Yukarıda gördüğünüz gibi, ilk sorumuz şu:

```
Enter the root path for documentation.
```

```
> Root path for the documentation [.]:
```

Burada Sphinx bizden, hazırlayacağımız projeye ilişkin bütün dosyaların hangi dizinde tutulacağını soruyor. Eğer burada hiçbir değer girmeden ENTER tuşuna basarsak, Sphinx, proje dosyalarını o anda içinde bulunduğumuz dizin içinde tutmak istediğimizi varsayacaktır. Biz biraz önce, oluşturduğumuz *python3* adlı dizin içinde bir komut satırı açarak sphinx-quickstart komutunu verdiğimiz için burada ENTER tuşuna basabiliriz. Böylece Sphinx, proje dosyalarını

*python3* adlı dizin içinde tutmak istediğimizi anlayacaktır. Eğer siz sphinx-quickstart komutunu başka bir konumda verdiyseniz, doğrudan ENTER tuşuna basmak yerine, proje dosyalarını tutacağınız dizinin yolunu elle de yazabilirsiniz. Ancak en kolay yöntem, önce proje dosyalarının tutulacağı dizine girmek, ardından bu dizin içinde bir komut satırı açmak, daha sonra komut satırında sphinx-quickstart komutunu vermek ve ilk soruya cevap olarak da doğrudan ENTER tuşuna basmaktır.

Böylece projeye ilişkin bütün dosyalarımız *python3* adlı dizin içine yerleştirilecektir.

Birinciye soruyu cevaplayıp ENTER tuşuna bastıktan sonra karşımıza ikinci soru gelecek:

```
You have two options for placing the build directory for Sphinx output.
Either, you use a directory "_build" within the root path, or you separate
"source" and "build" directories within the root path.
> Separate source and build directories (y/N) [n]:
```

Sphinx'le hazırlanan projelerde iki önemli kavram vardır: kaynak dosyalar ve inşa dosyaları. Kaynak dosyalar, projemizdeki belgeleri oluşturan ham verilerdir. Yani reStructuredText biçiminde hazırladığımız bütün belgeler bizim kaynak dosyalarımız oluyor. İnşa dosyaları ise, Sphinx'le yapılan derleme işlemi sonucu, bu reStructuredText belgelerinden üretilen HTML veya PDF dosyalarıdır. İşte Sphinx'in bize sorduğu bu ikinci soru bu konuyla ilgili. Burada Sphinx bize bu kaynak ve inşa dosyalarını ayrı dizinlerde tutmak isteyip istemediğimizi soruyor. Burada "y" [evet] ve "n" [hayır] olmak üzere iki farklı cevap verebilirsiniz. Öntanımlı cevap "n", yani hayırdır.

Eğer bu soruya "y" [evet] cevabı verirsiniz, *python3* adlı proje dizininiz içinde *build* ve *source* adlı iki farklı dizin oluşacak, derlenen HTML ve PDF dosyaları *build* adlı dizin içinde, *.rst* veya *.txt* uzantılı kaynak dosyalar ise *source* dizini içinde yer alacaktır.

Eğer "n" [hayır] cevabı verirsiniz *.rst* veya *.txt* uzantılı kaynak dosyalar doğrudan *python3* dizini içinde, derlenen HTML veya PDF dosyaları ise *python3* dizini altındaki *\_build* adlı bir inşa dizini içinde yer alacaktır.

Ben, projelerinizin daha derli toplu görünmesi açısından kaynak ve inşa dosyalarını ayrı dizinlere koymanızı tavsiye ederim. Eğer siz de benim gibi düşünüyorsanız burada önce "y" harfine, hemen ardından da ENTER tuşuna basın. Kaynak ve inşa dizinlerinizi ayırmak istemiyorsanız doğrudan ENTER tuşuna basabilirsiniz.

Sıra geldi üçüncü sorumuza:

```
Inside the root directory, two more directories will be created; "_templates"
for custom HTML templates and "_static" for custom stylesheets and other static
files. You can enter another prefix (such as ".") to replace the underscore.
> Name prefix for templates and static dir [_]:
```

Sphinx projelerinde iki temel dizin vardır. *templates* ve *static*. *templates* dizini içinde, HTML şablonları, *static* dizini içinde ise HTML dosyaları için kullanacağınız css ve resimler gibi sabit dosyalar yer alır. Bu iki dizin Sphinx'te öntanımlı olarak "\_" önekine sahiptir. Yani yukarıdaki soruya doğrudan ENTER tuşuna basarak cevap verirsiniz proje dizini içinde oluşturulacak *templates* ve *static* dizinleri *\_templates* ve *\_static* adlarına sahip olacaktır. İşte burada Sphinx size bu "\_" önekini değiştirme imkanı tanıyor. Eğer geçerli bir mazeretiniz yoksa, bu soruya doğrudan ENTER tuşuna basarak cevap verebilirsiniz. Böylece Sphinx öntanımlı önek olan "\_" işaretini kullanacaktır.

Sıradaki soruya bakalım:

```
The project name will occur in several places in the built documentation.
> Project name:
```



Burada Sphinx bize proje adının ne olacağını soruyor. Biz de ona uslu uslu cevap veriyoruz. Mesela proje adımız “python” olsun... Öyleyse bu sorunun hemen karşısına “python” yazıp ENTER tuşuna basalım. Burada gireceğiniz proje ismi, Sphinx ile üreteceğiniz HTML ve PDF belgelerinin bazı yerlerinde geçecektir. Ama endişelenmeyin, buraya yazdığınız ismi daha sonra değiştirebilirsiniz. O yüzden proje ismini çocuğunuza isim veriyormuşsunuz gibi uzun uzun düşünmenize gerek yok...

Sıradaki soru, yazar ismidir:

```
> Author name(s):
```

Buraya projeyi yazan kişi veya kişilerin adını yazıyoruz. Eğer projeyi hazırlayan birden fazla kişi varsa, bu kişilerin adlarını virgülle ayırarak yazabilirsiniz. Buraya yazdığınız isim veya isimler HTML çıktısında sayfanın dip kısmında *Copyright 2010, isim soyisim* şeklinde, PDF çıktısında ise ilk sayfada proje başlığının altında görünecektir.

Şimdi geldi sürüm numarasını belirtmeye:

```
Sphinx has the notion of a "version" and a "release" for the
software. Each version can have multiple releases. For example, for
Python the version is something like 2.5 or 3.0, while the release is
something like 2.5.1 or 3.0a1. If you don't need this dual structure,
just set both to the same value.
> Project version:
```

Burada, projemiz için bir sürüm numarası belirtiyoruz. Mesela 0.1 yazabilirsiniz.

Sonraki soruda ise alt sürüm numarası soruluyor:

```
> Project release [0.1]:
```

Dikkat ederseniz köşeli parantez içinde, bir önceki adımda verdiğimiz sürüm numarası gösteriliyor. Eğer ENTER tuşuna basacak olursanız, sürüm numarası aynı zamanda alt sürüm numarası olarak da kullanılacaktır. Eğer alt sürüm numarasının farklı bir şey olmasını isterseniz, elbette bunu da belirtebilirsiniz...

Sonraki soru önemli:

```
The file name suffix for source files. Commonly, this is either ".txt"
or ".rst". Only files with this suffix are considered documents.
> Source file suffix [.rst]:
```

Burada Sphinx bize kaynak dosyaların hangi uzantıya sahip olacağını soruyor. reStructuredText konusunu işlerken, rst dosyalarının .rst veya .txt uzantısına sahip olabileceğini söylemiştik. Dolayısıyla bu soruya cevap olarak .txt de yazabilirsiniz. Eğer hiçbir şey yazmadan ENTER tuşuna basarsanız öntanımlı uzantı .rst olarak belirlenecektir. Sphinx yalnızca bu soruda belirttiğiniz uzantıya sahip dosyaları kaynak dosya olarak dikkate alacaktır.

Yine önemli bir soruya geldi sıra:

```
One document is special in that it is considered the top node of the
"contents tree", that is, it is the root of the hierarchical structure
of the documents. Normally, this is "index", but if your "index"
document is a custom template, you can also set this to another filename.
> Name of your master document (without suffix) [index]:
```

Burada içindekiler dizininin tutulacağı dosyanın adını belirtiyoruz. Mesela ben istihza.com'daki Python 2.x bölümü için bu dosyanın adını “icindekiler\_python” olarak belirledim. Siz burada doğrudan ENTER tuşuna basarak, Sphinx'in size önerdiği “index” adını da kabul edebilirsiniz.

Eğer daha sonra başka bir isim kullanmaya karar verirsiniz bunu ileride değiştirme imkanınız var. O yüzden bu sorunun üzerinde de çok düşünmenize gerek yok.

Şimdi art arda birkaç soru dizisiyle karşılaşacağız:

```
Please indicate if you want to use one of the following Sphinx extensions:
> autodoc: automatically insert docstrings from modules (y/N) [n]:
> doctest: automatically test code snippets in doctest blocks (y/N) [n]:
> intersphinx: link between Sphinx documentation of different projects (y/N) [n]:
> todo: write "todo" entries that can be shown or hidden on build (y/N) [n]:
> coverage: checks for documentation coverage (y/N) [n]:
> pngmath: include math, rendered as PNG images (y/N) [n]:
> jsmath: include math, rendered in the browser by JSMath (y/N) [n]:
> ifconfig: conditional inclusion of content based on config values (y/N) [n]:
```

Bu soruların hepsini, şu aşamada, doğrudan ENTER tuşuna basarak geçebilirsiniz.

Son iki soru ise şöyle:

```
A Makefile and a Windows command file can be generated for you so that you
only have to run e.g. 'make html' instead of invoking sphinx-build
directly.
> Create Makefile? (Y/n) [y]:
> Create Windows command file? (Y/n) [y]:
```

Bu sorulara “y”, yani evet karşılığını vermenizi öneririm. Böylece sadece `make html` gibi bir komut kullanarak HTML dosyalarını oluşturabileceğiz.

Eğer her şey yolunda gitmişse göreceğimiz son cümleler şöyle olacaktır:

```
Finished: An initial directory structure has been created.

You should now populate your master file ./source/index.txt and create other documentation
source files. Use the Makefile to build the docs, like so:
    make builder
where "builder" is one of the supported builders, e.g. html, latex or linkcheck.
```

Şimdi en başta oluşturduğumuz proje dizinini açalım. Orada *build* ve *source* adında iki klasör, *make.bat* ve *Makefile* adında da iki dosya göreceğiz. Eğer Sphinx’in size sorduğu sorular arasında yer alan ikinci soruya “n”, yani hayır cevabı verdiyseniz proje dizini içinde *source* adlı ayrı bir dizin olmayacak, normalde bu dizin içinde bulunması gereken dosyalar doğrudan ana klasör altında yer alacaktır. Ben sizin o soruya “e” cevabı verdiğinizi varsayacağım.

*source* adlı dizinin içine girdiğinizde, orada *conf.py* adlı bir dosya göreceksiniz. İşte bu dosya Sphinx’in projemiz için otomatik olarak oluşturduğu ayar dosyasıdır. Biraz önce Sphinx’in bize sorduğu sorulara verdiğimiz cevaplar bu ayar dosyasına işlendi. Eğer istersek *conf.py* adlı bu dosya üzerinde değişiklik yapabiliriz. Burada yapacağımız değişiklikler projemizin HTML ve PDF olarak nasıl görüneceğini belirleyecektir. Bu yüzden *conf.py* son derece önemli bir dosyadır. Gelin isterseniz bu dosyaya biraz daha yakından bakalım.

## 33.4 *conf.py* Dosyası

Dediğimiz gibi, *conf.py* projemizle ilgili bütün ayarları tutan, son derece önemli bir dosyadır. Bu dosya bir bakıma Sphinx’in kalbidir. O halde gelin isterseniz bu önemli dosyayı biraz didikleylelim.

Uzantısından da anlayacağınız gibi, *conf.py* esasında bir Python programıdır. Dolayısıyla bu dosyaya herhangi bir Python betiğine nasıl davranıyorsanız öyle davranabilirsiniz.

*conf.py* dosyasının içini açtığımızda bunun, yukarıda sphinx-quickstart komutunu çalıştırdıktan sonra karşımıza çıkan sorulara verdiğimiz yanıtları da içeren bir ayar dosyası olduğunu görüyoruz. Dolayısıyla, yukarıdaki sorulara verdiğiniz cevapları bu dosyanın içinden rahatlıkla değiştirebilirsiniz. Dilerseniz bu dosyayı olduğu gibi de kullanabilirsiniz. Bu dosyadaki öntanımlı değerler de Sphinx’le iyi bir başlangıç yapmanız için yeterli olacaktır.

Şimdi bu *conf.py* dosyası içindeki değişkenlerin en önemlilerini incelemeye başlayalım:

**source\_suffix** Bu değişken, proje klasörü içinde yer alan source dizini içindeki kaynak dosyalarımızın hangi uzantıya sahip olacağını gösteriyor. Biz burada *.rst* veya *.txt* uzantılarından birini seçebiliriz. Hangisini seçtiğimizin önemi yok. Eğer kaynak dosyalarımız *.txt* uzantısına sahipse buraya *.txt*, eğer kaynak dosyalarımız *.rst* uzantısına sahipse buraya *.rst* yazacağız. Mesela benim istihza.com için hazırladığım belgeler *.txt* uzantısına sahip. O yüzden *source\_suffix* değerini ben *.txt* olarak belirledim.

**master\_doc** Bu değişken, içindekiler dizinini barındıracak dosyanın adını belirliyor. Öntanımlı değer “index”tir. Siz elbette farklı bir ad da belirleyebilirsiniz. Mesela ben istihza.com’daki Python 2.x için hazırladığım *conf.py*’de bu değeri “icindekiler\_python” olarak belirledim.

**project** Bu değer, projenin adını gösteriyor. Bu değişkenin karşısına projenizin adını yazacaksınız.

**copyright** Buraya copyright bilgilerini yazacağız. Mesela u’2010, Fırat Özgül’ gibi...

**language** Bu değer, oluşturulacak HTML dosyasının hangi dilde olacağını gösteriyor. Şu anda bu seçeneğin değeri şunlardan biri olabilir:

- cs – Çekce
- de – Almanca
- en – İngilizce
- es – İspanyolca
- fi – Fince
- fr – Fransızca
- it – İtalyanca
- nl – Felemenkçe
- pl – Lehçe
- pt\_BR – Portekizce (Brezilya)
- ru – Rusça
- sl – Slovence
- uk\_UA – Ukraynaca
- zh\_TW – Çince (Geleneksel)

**pygments\_style** Burada kod renklendirme işlemi için hangi renk şemasını kullanacağımızı belirtiyoruz. Buraya şu değerleri verebilirsiniz:

- fruity
- autumn
- manni
- perldoc
- pastie

- friendly
- default
- vs
- native
- colorful
- murphy
- bw
- tango
- emacs
- vim
- borland
- trac

**html\_theme** Kullanılacak HTML temasını gösterir. Buraya “default” veya “sphinxdoc” değerlerinden birini yazabilirsiniz. Eğer “default” değerini kullanırsanız HTML çıktılarınızın görünüşü <http://docs.python.org/> adresindeki gibi, eğer “sphinxdoc” değerini kullanırsanız [http://www.istihza.com/py2/icindekiler\\_python.html](http://www.istihza.com/py2/icindekiler_python.html) adresindeki gibi olacaktır.

**html\_title** Bu değer HTML dosyalarının başlığını belirler. Mesela istihza.com’daki Python 2.x bölümü için ben bu değeri “Python Kılavuzu” olarak belirledim.

## 33.5 index Dosyası

sphinx-quickstart komutunun ardından gelen sorulara cevap verdikten sonra *sources* dizini altında bir adet *conf.py* dosyasının oluştuğunu görmüştük. İşte bu *conf.py* dosyasının bulunduğu dizin içinde *index.rst* veya *index.txt* adlı başka bir dosya daha bulunur. sphinx-quickstart soruları arasında yer alan “Name of your master document” ve “source file suffix” sorularına verdiğiniz cevaba bağlı olarak bu dosyanın adı sizde tamamen farklı olabilir. Mesela istihza.com’da bu dosyanın adı *icindekiler\_python.txt*.

Bu dosyayı açtığımızda şöyle bir içerikle karşılaşacağız:

```
.. python documentation master file, created by
   sphinx-quickstart on Sat Mar 20 00:37:25 2010.
   You can adapt this file completely to your liking,
   but it should at least contain the root 'toctree'
   directive.
```

Welcome to python's documentation!

=====

Contents:

```
.. toctree::
   :maxdepth: 2
```

Indices and tables

=====

```
* :ref:'genindex'
```

```
* :ref:'modindex'  
* :ref:'search'
```

Gördüğünüz gibi, reStructuredText biçimli bir dosyadır bu. Bu dosya, dönüştürme işleminin ardından bizim HTML giriş sayfamızı oluşturacaktır.

Bu sayfayı istediğiniz gibi düzenleyebilirsiniz. Ancak sayfada en azından `.. toctree::` yönergesinin kalmasına dikkat etmeniz gerekiyor. Yani yukarıdaki dosyanın içeriği en sade haliyle şöyle olabilir:

```
.. toctree::
```

Mesela istihza.com için hazırladığım *icindekiler\_python.txt* dosyasının içeriği şöyle:

```
.. meta::  
   :description: Python Programlama Dilinin 2.x Sürümü için Türkçe Kaynak  
   :keywords: python, türkçe, belgelendirme, 2.x
```

### Python 2.x için Türkçe Kılavuz =====

**.. warning::** Aşağıdaki bilgiler Python'un 2.x sürümleri içindir. Eğer kullandığınız sürüm Python'un 3.x sürümlerinden biriye ['şuradaki <http://www.istihza.com/py3/icindekiler\\_python.html>](http://www.istihza.com/py3/icindekiler_python.html)'\_ belgeleri inceleyebilirsiniz.

-----

#### **\*\*TEMEL KONULAR\*\***

**.. note::** Temel Konular bölümünde Python programlama dilinin özünü oluşturan, karakter dizileri, sayılar, listeler, demetler, sözlükler, fonksiyonlar, dosya işlemleri ve hata yakalama gibi temel düzeydeki konular işlenmektedir. Bu bölümde ayrıca düzenli ifadeler ve nesne tabanlı programlama gibi orta-ileri düzey konulara da yer verilmiştir. Bu bölümü bitirdiğinizde Python'la konsol uygulamaları yazabilecek düzeye geleceksiniz. Eğer grafik arayüze sahip programlar geliştirmek isterseniz sitemizde bulunan ['Tkinter <http://www.istihza.com/tk2/icindekiler\\_tkinter.html>](http://www.istihza.com/tk2/icindekiler_tkinter.html)'\_ veya ['PyGTK <http://www.istihza.com/gtk/icindekiler\\_pygtk.html>](http://www.istihza.com/gtk/icindekiler_pygtk.html)'\_ bölümlerini inceleyebilirsiniz.

```
.. toctree::  
   :numbered:
```

```
temel_bilgiler.txt  
kosul.txt  
dongu.txt  
liste-demet-sozluk.txt  
fonksiyon.txt  
modul.txt  
dosya.txt  
hata.txt  
kardiz.txt  
regex.txt  
nesne.txt  
unicode.txt  
modifier.txt
```

## **\*\*ÖZEL KONULAR\*\***

**.. note::** Özel Konular bölümü Python programlama diline ilişkin bir konunun özel bir yönünü inceleyen makalelere ayrılmıştır. Bu bölümde math modülü ve pypdf modülü gibi ileri düzey modüllerin yanı sıra, Python'un önbellekleme mekanizmasına ilişkin bir inceleme yazısı ve Python'un Windows işletim sisteminde nasıl sistem yoluna (PATH) ekleneceğini anlatan bir makale yer almaktadır. Bu bölümde ayrıca Python ve OpenOffice ilişkisini inceleyen bir makale de bulacaksınız. Bunların dışında bu bölümde, belgelendirme çalışması yapanlar için büyük kolaylık sağlayan reStructuredText işaretleme dili ve Sphinx yazılımı da ayrıntılı olarak incelenmektedir.

**.. toctree::**  
**:numbered:**

```
math.txt
lenascii.txt
id_is.txt
windows-path.txt
pep3000.txt
pypdf.txt
openoffice.txt
paketler.txt
restructuredtext.txt
sphinx.txt
```

Bu rst dosyası Sphinx yardımıyla HTML'ye dönüştürüldüğünde [http://www.istihza.com/py2/icindekiler\\_python.html](http://www.istihza.com/py2/icindekiler_python.html) adresinde gördüğünüz sayfa ortaya çıkıyor.

Bu arada, yukarıdaki örnek sayfada bulunan pek çok reStructuredText niteliği size yabancı gelmiş olabilir. Ama hiç endişe etmeyin. Bu bölümde o sayfada gördüğünüz her şeyi tek tek inceleyeceğiz.

## **33.6 Özel Yönergeler (Directives)**

reStructuredText biçimli belgelerdeki bir kelimenin, satırın veya paragrafın nasıl işleneceğini gösteren özel komutlara yönerge adı verilir (yabancılar buna *directive* diyor). Mesela *reStructuredText* bölümünde gördüğümüz **.. image::** bir yönergedir.

Sphinx'te yönergelerin özel bir söz dizimi bulunur. Buna göre bir yönerge şöyle gösterilir:

```
.. yönerge_adı::
```

Yönergeler, tanımladıkları kelimenin, satırın ya da paragrafın özel bir şekilde yorumlanmasını sağlar. Dilerseniz sözü daha fazla uzatmadan, yönerge kavramını örnekler vasıtasıyla anlamaya çalışalım.

Bu bölümde inceleyeceğimiz ilk yönergemiz **.. toctree::** yönergesi...

### 33.6.1 .. toctree:: Yönergesi

Bu yönergenin görevi projeniz için bir içindekiler tablosu oluşturmaktır. Yukarıda da söylediğimiz gibi, bir *index* dosyasında en azından bir adet .. toctree:: yönergesi olmalı.

Bu yönergeyi şöyle kullanıyoruz:

```
.. toctree::  
  
    dosya1.rst  
    dosya2.rst  
    dosya3.rst
```

Burada görünen *dosya1.rst*, *dosya2.rst* ve *dosya3.rst* projeniz içindeki reStructuredText biçimli belgelerin dosya adıdır. Bu yönergeye göre, proje dizinimizdeki *source* klasörü içinde *dosya1.rst*, *dosya2.rst* ve *dosya3.rst* adlı üç adet dosya var. Dosyalarımızın uzantısı *.txt* veya *.rst* olabilir. Ben dosya uzantılarını genellikle *.txt* olarak belirliyorum, ama dosya uzantısının ne olduğunun hiç bir önemi yok. Tek önemli nokta, *conf.py* dosyasındaki *source\_suffix* değişkeninde belirttiğiniz uzantı adı ile proje dizini içindeki dosyaların uzantısının aynı olması gerektiğidir.

.. toctree:: yönergesini bu şekilde yazdığımızda, *dosya1.rst*, *dosya2.rst* ve *dosya3.rst* adlı dosyaların içinde bulunan bütün başlıklar “içindekiler” kısmındaki yerini alacaktır. Mesela [http://www.istihza.com/py2/icindekiler\\_python.html](http://www.istihza.com/py2/icindekiler_python.html) adresinde bulunan içindekiler tablosuna bakın. Orada “Temel Bilgiler” başlığı altında gördüğünüz bütün o alt-başlıkları oluşturan satır, yukarıdaki örnek dosyada yer alan “temel\_bilgiler.txt” satırıdır.

.. toctree:: yönergesi içinde yer alan başlıkların davranışını belirleyen iki önemli nitelik vardır. Bu niteliklerden ilki *:numbered:*, ikincisi ise *:maxdepth:* niteliğidir. Öncelikle *:numbered:* niteliğine değinelim. Eğer .. toctree:: yönergesi içinde bir *:numbered:* niteliği belirtirsek, içindekiler tablosunda yer alan bütün başlıklar kendi içinde numaralandırılacaktır. Örneğin:

```
.. toctree::  
    :numbered:  
  
    dosya1.rst  
    dosya2.rst  
    dosya3.rst
```

İkinci niteliğimiz de *:maxdepth:* niteliğidir. Bu nitelik, içindekiler tablosundaki başlıkların ayrıntı düzeyini belirler. Örneğin:

```
.. toctree::  
    :maxdepth: 2  
  
    dosya1.rst  
    dosya2.rst  
    dosya3.rst
```

Burada *:maxdepth: 2* satırı, içindekiler tablosunda ana başlıkların ve bir alt düzeyde yer alan başlıkların gösterileceğini belirtiyor. Eğer 2 yerine 3 yazarsak, ana başlıklarla birlikte iki alt düzeyde yer alan başlıklar gösterilecektir. Eğer bu niteliği hiç belirtmezsek, dosya içinde yer alan bütün başlıklar içindekiler tablosunda görünecektir.

### 33.6.2 .. meta:: Yönergesi

Bu yönerge, bir HTML belgesinin <head> kısmında bulunması gereken *meta* imlerini belirlememizi sağlar. Mesela:

```
.. meta::
  :description: Python Programlama Dilinin 2.x Sürümü için Türkçe Kaynak
  :keywords: python, türkçe, belgelendirme, 2.x
```

Burada .. meta:: yönergesiyle birlikte :description: ve :keywords: adlı iki tane de nitelikten yararlandık. :description: niteliği HTML belgesinin tanım kısmını oluşturur. :keywords: ise HTML belgesinin anahtar kelimelerini sıralamamızı sağlar. Bu yönerge HTML çıktısının <head> kısmında şöyle bir girdi oluşturacaktır:

```
<meta name="description"
      content="Python Programlama Dilinin 2.x Sürümü için Türkçe Kaynak">
<meta name="keywords"
      content="python, türkçe, belgelendirme, 2.x">
```

### 33.6.3 .. highlight:: Yönergesi

Bu yönerge, belge içinde geçen kodların hangi dile göre renklendirileceğini belirler. Mesela Python'la ilgili bir belgelendirme çalışması yapıyorsanız, yazacağınız örnek kodların düzgün renklendirilebilmesi için bu yönergeyi kullanabilirsiniz:

```
.. highlight:: py

range() fonksiyonunu for döngüsüyle birlikte şu şekilde
kullanabilirsiniz::

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

Burada .. highlight:: py yönergesi, bir Python kodu yazacağımızı gösteriyor. Aynı metin içinde birden fazla .. highlight:: yönergesi kullanabiliriz. Örneğin bir paragrafta C kodu, başka bir paragrafta Python kodu yazacaksak her biri için ayrı bir .. highlight:: yönergesi belirtebiliriz:

```
.. highlight:: c

C kodu::

    #include <stdio.h>

    int main()
    {
        int a = 1;
        if (a == 1)
        {
            printf("Elveda Zalim Dünya!\n");
            return 0;
        }
    }

.. highlight:: py
```



Python kodu::

```
a = 1
if a == 1:
    print "Elveda Zalim Dünya"
```

.. **highlight**:: yönergesiyle birlikte, Pygments'in desteklediği bütün dilleri kullanabilirsiniz. Pygments'in hangi dilleri desteklediğini görmek için <http://pygments.org/docs/lexers/> adresini ziyaret edebilirsiniz.

### 33.6.4 .. **note**:: Yönergesi

Bu yönerge, okurlarınıza incelediğiniz konuyla ilgili şık bir **not** göstermenizi sağlar. Mesela:

```
.. note:: Temel Konular bölümünde Python programlama dilinin
özünü oluşturan, karakter dizileri, sayılar, listeler, demetler,
sözlükler, fonksiyonlar, dosya işlemleri ve hata yakalama gibi
temel düzeydeki konular işlenmektedir. Bu bölümde ayrıca düzenli
ifadeler ve nesne tabanlı programlama gibi orta-ileri düzey
konulara da yer verilmiştir.
```

### 33.6.5 .. **warning**:: Yönergesi

Bu yönerge, okurlarınıza incelediğiniz konuyla ilgili şık bir **uyarı** göstermenizi sağlar. Mesela:

```
.. warning:: Aşağıdaki bilgiler Python'un 2.x sürümleri içindir.
Eğer kullandığınız sürüm Python'un 3.x sürümlerinden biriye
'suradaki <http://www.istihza.com/py3/icindekiler-python.html>'_
belgeleri inceleyebilirsiniz.
```

[...DEVAM EDECEK...]



---

# Dizin

---

32 bit, 368  
64 bit, 368  
add()  
    Küme, 130  
addPage()  
    pyPdf, 197  
alan formülü  
    üçgen, 37  
Anahtar  
    Key, 114  
append(), 97  
architecture  
    platform, 368  
arg, 155  
Argüman, 144  
    isimli, 148  
    sıralı, 148  
    varsayılan değerli, 152  
args, 155  
argv  
    sys, 363  
Aritmetik işlemler, 16  
ASCII, 38  
author  
    pyPdf, 195  
Belgelendirme, 169  
belgelendirme dizisi  
    docstring, 169  
Biçim Düzenleyiciler, 258  
    c harfi, 262  
    d harfi, 260  
    f harfi, 262  
    i harfi, 261  
    o harfi, 261  
    X harfi, 262  
    x harfi, 261  
break, 83  
Builtin Functions  
    Gömülü Fonksiyonlar, 162  
C, 2, 57  
    programlama dili, 361  
C++, 2  
Çıkış, 7  
    quit(), 7  
    sys.exit(), 7  
Çalıştırma, 5  
    cmd, 6  
    gnome-terminal, 5  
    GNU/Linux'ta, 5  
    konsole, 5  
    Windows'ta, 6  
capitalize()  
    Karakter Dizileri, 230  
center()  
    Karakter Dizileri, 236  
chdir()  
    os, 187  
chmod, 29  
clear(), 121  
çok katmanlı, 392  
Comment  
    Yorum, 59  
Comment Out  
    Yorum İçine Alma, 60  
Compiler  
    Derleyici, 57  
continue, 84  
copy()  
    Küme, 130  
count(), 103, 112  
    Karakter Dizileri, 241  
cp1254, 39  
creator

- pyPdf, 196
- datetime, 384
  - date, 385
  - strftime, 388
  - strftime(), 385
  - timedelta(), 390
  - today(), 385
- Değer
  - Value, 114
- değişkenler, 19
- def, 141
- del, 100, 119
- Demet
  - öğelerinin sırasını bulmak, 112
  - öğelerinin sayısını bulmak, 112
  - count(), 112
  - index(), 112
  - metotları, 112
  - Tuple, 109
- denetim masası, 6
- Derleme
  - kaynaktan, 4
- Derleyici
  - Compiler, 57
- dict, 114
- dict(), 128
- Dictionary
  - Sözlük, 113
- difference()
  - Küme, 131
- difference\_update()
  - Küme, 131
- dil yereli
  - locale, 389
- dilimleme
  - slicing, 226
- discard()
  - Küme, 132
- Dive into Python, 194
- doc, 169
- docstring
  - belgelendirme dizisi, 169
- Dosyalar, 200
- dosyalar
  - dosya oluşturmak, 201
  - dosyaya yazmak, 205
- e-posta, 50
- eşittir, 51
- echo, 31
- elif, 53
- else, 56
- encoding, 38
- endswith()
  - Karakter Dizileri, 241
- enumerate(), 105
- env
  - betiği, 28
- Environment variables
  - Ortam değişkenleri, 6
- Etkileşimli kabuk
  - Interactive shell, 9
- Etkisizleştirme
  - Raw, 20
- exit
  - sys, 366
- expandtabs()
  - Karakter Dizileri, 245
- export, 31
- extend(), 99
- fahrenheit, 39
- Ferhat, 56
- find()
  - Karakter Dizileri, 245
- Fonksiyon, 139
  - tanımlamak, 141
- Fonksiyon çağırısı
  - Function call, 142
- Fonksiyonlar
  - belgelendirilmesi, 169
- for, 80
- Format Modifiers, 258
- from \_\_future\_\_ import division, 18
- frozenset
  - Küme, 138
- fstab, 30
- Function call
  - Fonksiyon çağırısı, 142
- Gömülü Fonksiyonlar
  - Builtin Functions, 162
- Geany, 58
- Gedit
  - Metin düzenleyici, 27
- get(), 122
- getcwd()
  - os, 186
- getdefaultencoding()
  - sys, 367
- getDocumentInfo()
  - pyPdf, 194
- getNumPages()
  - pyPdf, 198
- Girintileme
  - Indentation, 52, 57
- global, 163

- Gmail, 50
- Google, 2
- Guido Van Rossum
  - özgeçmişi, 2
  - fotografları, 2
  - Hollandalı, 2
- has\_key(), 122
- hata
  - yakalama, 85
- Hesap makinesi, 55
- Hewlett Packard, 2
- Hoary Hedgehog, 109
- hosts, 35
- HTML, 59
- içe aktarma
  - import, 175
- İşlem öncelik sırası, 18
- iexplore.exe, 34
- if, 51
- immutable
  - değiştirilemeyen, 223
- import, 101
  - içe aktarma, 175
- in
  - deyimi, 107
- Indentation
  - Girintileme, 52, 57
- index(), 102, 112
  - Karakter Dizileri, 247
- input(), 46
- insert(), 99
- Interactive shell
  - Etkileşimli kabuk, 9
- Interpreter
  - Yorumlayıcı, 9
- intersection()
  - Küme, 132
- intersection\_update()
  - Küme, 134
- IronPython, 8
- isalnum()
  - Karakter Dizileri, 244
- isalpha()
  - Karakter Dizileri, 242
- isdigit()
  - Karakter Dizileri, 243
- isdisjoint()
  - Küme, 134
- isim alanı
  - namespace, 164
- isimli argüman
  - istenen sayıda, 159

- islower()
  - Karakter Dizileri, 232
- isspace()
  - Karakter Dizileri, 233
- issubset()
  - Küme, 135
- issuperset()
  - Küme, 135
- istitle()
  - Karakter Dizileri, 233
- isupper()
  - Karakter Dizileri, 232
- items(), 121
- iteritems(), 121
- iterkeys(), 122
- itervalues(), 122
- join(), 191
  - Karakter Dizileri, 248
- Jython, 8
- Küme
  - öge sıralaması, 130
  - add(), 130
  - clear(), 129
  - copy(), 130
  - difference(), 131
  - difference\_update(), 131
  - discard(), 132
  - frozenset, 138
  - intersection(), 132
  - intersection\_update(), 134
  - isdisjoint(), 134
  - issubset(), 135
  - issuperset(), 135
  - metotları, 128
  - oluşturmak, 126
  - pop(), 138
  - remove(), 132
  - Set, 125
  - symmetric\_difference(), 137
  - symmetric\_difference\_update(), 137
  - union(), 136
  - update(), 136
- Kümeler, 215
- Kaçış dizileri, 20
- Karakter Dizileri
  - biçimleyiciler, 20
  - birleştirme, 15
  - capitalize(), 230
  - center(), 236
  - count(), 241
  - dilimleme, 226
  - endswith(), 241

- expandtabs(), 245
- find(), 245
- index(), 247
- isalnum(), 244
- isalpha(), 242
- isdigit(), 243
- islower(), 232
- isspace(), 233
- istitle(), 233
- isupper(), 232
- join(), 248
- ljust(), 237
- lstrip(), 252
- metotları, 229
- partition(), 251
- replace(), 239
- rfind(), 247
- rindex(), 248
- rjust(), 237
- rpartition(), 252
- rsplit(), 255
- rstrip(), 252
- split(), 253
- splitlines(), 256
- startswith(), 240
- Strings, 11, 218
- strip(), 252
- swapcase(), 231
- title(), 230
- translate(), 250
- upper(), 231
- zfill(), 238
- Karmaşık Sayılar
  - Sayılar, 18
- Kate
  - Metin düzenleyici, 27
- Kayan noktalı sayılar
  - Sayılar, 18
- Keramet
  - Su, 60
- Key
  - Anahtar, 114
- keys(), 120
- Koşula bağlı durumlar, 50
- kullanıcı adı, 50
- Kurulum
  - GNU/Linux'ta, 3
  - kaynaktan, 4
  - make altinstall, 4
  - Windows'ta, 4
- Kwrite
  - Metin düzenleyici, 27
- len(), 82

- List
  - Liste, 91
- listdir()
  - os, 184
- Liste
  - öge silmek, 100
  - öğelerine erişmek, 93
  - öğelerini sıralamak, 101
  - öğelerinin sırasının bulmak, 102
  - öğelerinin sayısını bulmak, 103
  - List, 91
  - metotları, 96, 97
  - oluşturmak, 92
- ljust()
  - Karakter Dizileri, 237
- locale, 101, 388
  - dil yereli, 389
- localtime()
  - time, 395
- lstrip()
  - Karakter Dizileri, 252
- Lucida console, 41
- makedirs()
  - os, 187
- Matematik işlemleri, 16
- max(), 108
- Metin düzenleyici, 26
  - Gedit, 27
  - IDLE, 31
  - Kate, 27
  - Kwrite, 27
- min(), 108
- mkdir()
  - os, 187
- Modüller, 172
  - üçüncü şahısların yazdığı, 192
  - çeşitleri, 173
  - kendi yazdığınız, 174
  - yol, 198
- Monty Python's Flying Circus, 2
- MS-DOS, 6, 41
- multi-threading, 392
- mutable
  - değiştirilebilir, 223
- name
  - os, 181
- namespace
  - isim alanı, 164
- NASA, 2
- Net çatısı, 8
- Non-ASCII, 38
- None, 166

- Notepad, 26
- Notepad++, 58
- Notepad2, 58
- Orçun
  - Kunek, 59
- Ortam değişkenleri
  - Environment variables, 6
- os, 180
  - chdir(), 187
  - environ, 226
  - getcwd(), 186
  - listdir(), 184
  - makedirs(), 187
  - mkdir(), 187
  - name, 181
  - removedirs(), 189
  - rmdir(), 189
  - sep, 190
  - startfile, 183
  - system, 182
- os.path.abspath(), 213
- os.path.basename(), 213
- os.path.dirname(), 213
- os.path.exists(), 213
- os.path.isdir(), 214
- os.path.isfile(), 214
- os.path.split(), 215
- os.path.splitext(), 215
- Parametre, 144
- Pardus, 3
- parola, 50
- partition()
  - Karakter Dizileri, 251
- pass
  - deyimi, 90
  - fonksiyonlarda, 168
- PATH, 6
  - YOL, 6, 30
- path
  - sys, 198, 367
- PATHEXT, 36
- pdf, 193
- PdfFileReader()
  - pyPdf, 194
- PdfFileWriter()
  - pyPdf, 194
- platform, 368
  - architecture, 368
  - sys, 368
- pop(), 100, 121
  - Küme, 138
- popitem(), 121

- pow(), 221
- print
  - komutu, 9
- printf, 57
- producer
  - pyPdf, 196
- pydoc, 8
- pyPdf, 193
  - addPage(), 197
  - author, 195
  - creator, 196
  - getDocumentInfo(), 194
  - getNumPages(), 198
  - PdfFileReader(), 194
  - PdfFileWriter(), 194
  - producer, 196
  - title, 194
- Python
  - 'paytın', 3
  - 'piton', 3
  - 2.x, 5
  - 3.x, 5
  - kurulum, 3
  - peltek s, 3
  - piton yılanı, 2
  - telaaffuzu, 3
- python.exe, 35
- range(), 82
- Raw
  - Etkisizleştirme, 20
- raw\_input(), 43
  - ile çift tıklama, 37
- reload(), 177
- remove(), 100
  - Küme, 132
- removedirs()
  - os, 189
- replace()
  - Karakter Dizileri, 239
- return, 166
- reverse(), 101
- rfind()
  - Karakter Dizileri, 247
- rindex()
  - Karakter Dizileri, 248
- rjust()
  - Karakter Dizileri, 237
- rmdir()
  - os, 189
- root, 30
- rpartition()
  - Karakter Dizileri, 252
- rsplit()

- Karakter Dizileri, 255
- rstrip()
  - Karakter Dizileri, 252
- Sözlük
  - öge eklemek, 116
  - öğelerine erişmek, 115
  - öğelerini değiştirmek, 119
  - öğelerini silmek, 119
  - Dictionary, 113
  - metotları, 120
  - oluşturmak, 114
  - sıra kavramı, 124
- Sıralı Argüman
  - istenen sayıda, 155
- santigrat, 39
- Sayılar, 16
  - Karmaşık Sayılar, 18
  - Kayan noktalı sayılar, 18
  - Tamsayılar, 18
  - Uzun sayılar, 18
- Sekme
  - Tab, 20
- sep
  - os, 190
- Set
  - Küme, 125
- set()
  - fonksiyonu, 126
- shebang, 27
- sleep()
  - time, 391
- slicing
  - dilimleme, 226
- sort(), 101
- SPACE, 52
- split()
  - Karakter Dizileri, 253
- splitlines()
  - Karakter Dizileri, 256
- startfile
  - os, 183
- startswith()
  - Karakter Dizileri, 240
- stdout.write()
  - sys, 369
- str(), 219
- strftime()
  - time, 393
- Strings
  - Karakter Dizileri, 11, 218
- strip()
  - Karakter Dizileri, 252
- strxfrm, 101

- sum(), 104, 145
- swapcase()
  - Karakter Dizileri, 231
- symmetric\_difference()
  - Küme, 137
- symmetric\_difference\_update()
  - Küme, 137
- sys, 361
  - argv, 363
  - exit, 366
  - getdefaultencoding(), 367
  - path, 198, 367
  - platform, 368
  - stdout, 369
  - stdout.write(), 369
  - version\_info, 373
- sys.stdout
  - standart çıktı konumu, 372
- system
  - os, 182
- Türkçe
  - karakter problemi, 41
  - karakterler, 13, 38
- tırnak işaretleri
  - üç, 13
  - çift, 12
  - tek, 13
- TAB, 52
- Tab
  - Sekme, 20
- Tamsayılar
  - Sayılar, 18
- Tarih
  - aritmetik işlem, 390
  - biçimlendirme, 388
  - bugünün tarihi, 385
  - haftanın hangi günü, 386
- TCMB, 390
- time, 390
  - localtime(), 395
  - sleep(), 391
  - strftime(), 393
- title
  - pyPdf, 194
- title()
  - Karakter Dizileri, 230
- Tkinter, 392
- translate()
  - Karakter Dizileri, 250
- try... except, 87
- Tuple
  - Demet, 109
- type(), 219



- fonksiyonu, 110
- Ubuntu, 3
  - ubuntu.com, 4
  - ubuntu.org.tr, 4
- Uncomment
  - Yorumdan Kurtarma, 60
- union()
  - Küme, 136
- update()
  - Küme, 136
- upper()
  - Karakter Dizileri, 231
- USERPROFILE, 35
- utf-8, 39
- Uzun sayılar
  - Sayılar, 18
- Value
  - Değer, 114
- ValueError, 86
- values(), 120
- VirtualBox, 8
- vlc.exe, 42
- Warty Warthog, 109
- which, 30
- while, 76
- with, 217
- xdg-open, 183
- xorg.conf, 30
- Yeni satır, 20
- YOL
  - PATH, 6, 30
- Yorum
  - Comment, 59
- Yorum İçine Alma
  - Comment Out, 60
- Yorumdan Kurtarma
  - Uncomment, 60
- Yorumlayıcı
  - Interpreter, 9
- YouTube, 2
- ZeroDivisionError, 88
- zfill()
  - Karakter Dizileri, 238