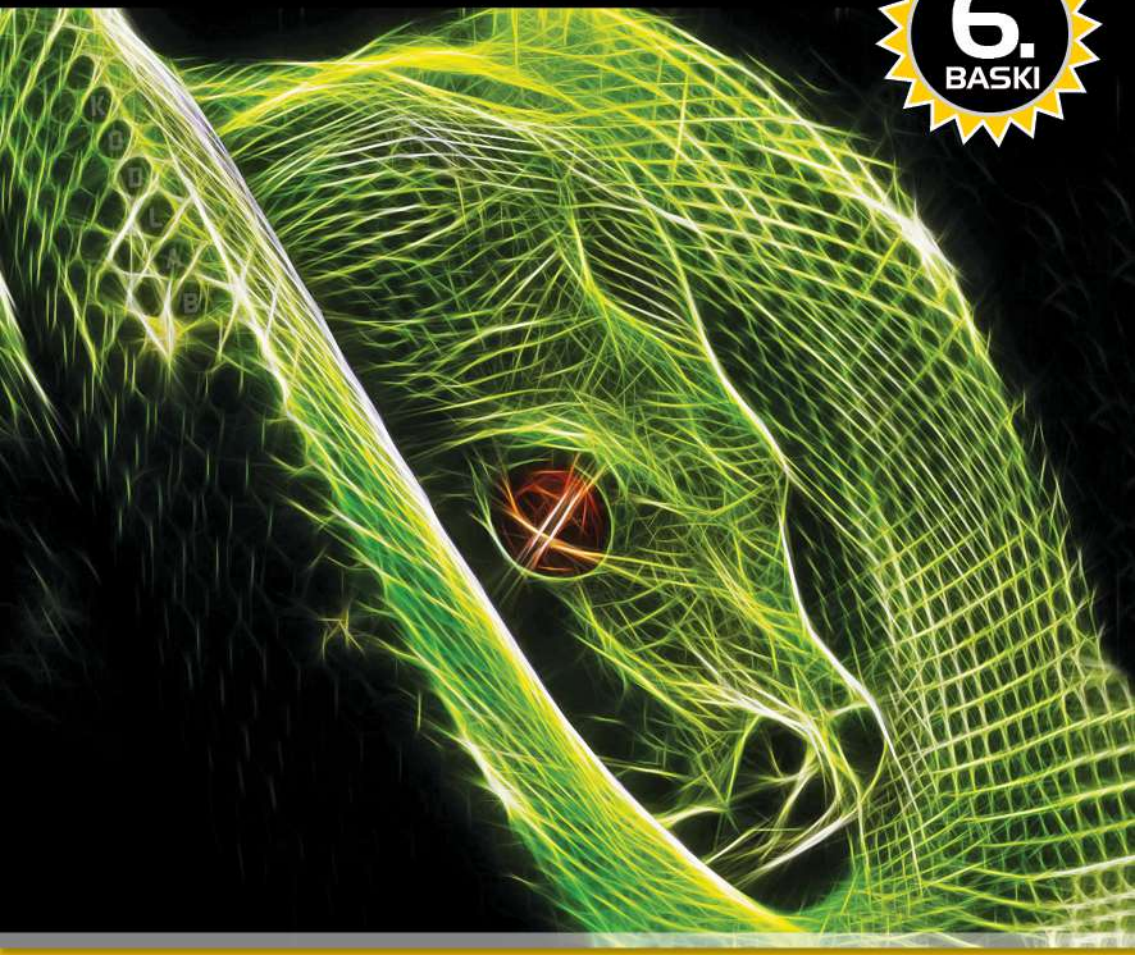
Fırat ÖZGÜL



6. BASKI

HER YÖNÜYLE PYTHON

KODLAB®
www.kodlab.com

HER YÖNÜYLE
PYTHON

Fırat Özgül

KODLAB®
Yayın Dağıtım Yazılım ve Eğitim
Hizmetleri San. ve Tic. Ltd. Şti.

KODLAB® 109

HER YÖNÜYLE PYTHON

FIRAT ÖZGÜL

ISBN 978-605-5201-26-5
Yayıncılık Sertifika No: 13206

1. Baskı: Temmuz 2013
2. Baskı: Ağustos 2014
3. Baskı: Şubat 2016
4. Baskı: Şubat 2017
5. Baskı: Ağustos 2017
6. Baskı: Ekim 2017

Yayına Hazırlık
Hüseyin Üstünel
Barış Elitoğ

Grafik Tasarım
Mustafa Said Şahin
Eray Küçük

Sosyal Medya Sorumlusu
Mahmut İpek

Satış ve Pazarlama
Can Üstünel
Yavuz Şengönül

Baskı: İnkılâp Kitabevi Yayın San. Tic. A.Ş. Tel: (0212) 496 11 81
Çobançeşme Mah. Sanayi Cad. Altay Sk. No:8 Yenibosna/İSTANBUL
Sertifika No: 10614

Bu kitabın bütün yayın hakları Kodlab Yayın Dağıtım Yazılım ve Eğitim Hizmetleri San. ve Tic. Ltd. Şti.'ne aittir. Yayınevimizin yazılı izni olmaksızın kısmen veya tamamen alıntı yapılamaz, kopya çekilemez, çoğaltılamaz ve yayınlanamaz.

KODLAB Yayın Dağıtım Yazılım ve Eğitim Hizmetleri San. ve Tic. Ltd. Şti.
15 Temmuz Mah. 1481. Sok. No: 44/A Bağcılar / İSTANBUL

tel: 0(212) 514 55 66 **web:** www.kodlab.com
fax: 0(212) 514 66 61 **e-posta:** bilgi@kodlab.com

FIRAT ÖZGÜL

1980 yılında Adana'da doğdu. Boğaziçi Üniversitesi Mütercim Tercümanlık Bölümü'nü bitiren yazarımız, özellikle özgür yazılım üzerine yaptığı çalışmalarla tanınmaktadır.

Yazarımız, Python programlama dili için bir Türkçe kaynak oluşturma projesi olan istihza.com'daki yazılarına devam etmektedir. Ayrıca yazarımızın KODLAB'tan daha önce PYTHON isimli kitabı çıkmıştır.

ÖNSÖZ

Python, 1990 yılından bu yana büyük bir topluluk tarafından geliştirilen kıvrak ve dinamik bir dildir. Sözdiziminin sade olması, kolay öğrenilmesi ve program geliştirme sürecini son derece hızlandırması ile tanınan bu dil; **Windows**, **GNU/Linux** ve **MacOSX** gibi pek çok farklı platform üzerinde çalışabilmektedir. Dolayısıyla tek bir platform üzerinde geliştirdiğiniz bir Python uygulaması, üzerinde hiç bir değişiklik yapmaya gerek olmadan veya küçük değişikliklerle başka platformlar üzerinde de çalışabilecektir.

Son yıllarda ülkemizde de ilgi uyandırmaya başlayan bu dil, yavaş yavaş üniversite müfredatlarında yerini almaktadır. Her ne kadar ülkemizde henüz yeterli ilgiyi görmese de, bugün Google'dan NASA'ya, YouTube'tan Hewlett Packard'a kadar pek çok kurum ve şirkette Python programlama dilinden yararlanılmaktadır.

Elinizdeki kitap, Python programlama diline merak duyan okurlara bu dilin temellerini sağlam ve hızlı bir şekilde öğretmeyi amaçlamaktadır. Bu kitaptan yararlanabilmek için önceden bir programlama dili biliyor olmanız gerekmez. Bu kitap, okurun programlama konusunda hiç bir bilgisi olmadığı varsayılarak ve herkesin anlayabileceği bir üslupla hazırlanmıştır. Python'un **2.7.11** sürümünü temel alan bu kitap üç bölümden oluşmaktadır.

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.

Özel Konular bölümü; Python programlama diline ilişkin bir konunun özel bir yönünü inceleyen makalelere ayrılmıştır. Bu bölümde Math modülünün 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'un 2.x sürümlerinden 3.x sürümlerine geçişte size kılavuz olacak bir makale de bulacaksınız.

Arayüz Programlama bölümü ise; Python programlama dilinin resmi arayüz takımı olan Tkinter'e ayrılmıştır. Bu bölümde Tkinter arayüz takımı üzerine ayrıntılı bir bilgi edinecek ve konsol uygulamalarının yanı sıra arayüze sahip modern uygulamalar geliştirebilecek düzeye geleceksiniz.

Fırat Özgül

Geceler boyu klavye tıkırtılarına sabırla katlanan eşim Hazan'a ve ikiz oğullarımız Deniz ile Ege'ye ithafen....

TEŞEKKÜR

Bu kitabın eksiksiz, doğru ve okunaklı olması için vakit ayırıp bana destek veren bütün istihza.com okurlarına, Türkiye'deki özgür yazılım topluluklarına ve "Math Modülü" adlı makaleye katkılarından ötürü Sayın Kürşat Örsel'e teşekkürlerimi sunarım.

İÇİNDEKİLER

KISIM I: TEMEL KONULAR **1**

1 TEMEL BİLGİLER **3**
Python Hakkında 3
Python'u Nereden Bulabilirim? 4
GNU/Linux 4
Microsoft Windows 5
Python Nasıl Çalıştırılır? 5
GNU/Linux'ta Python'u Çalıştırmak 6
Windows'ta Python'u Çalıştırmak 7
print Komutu 8
Python'da Sayılar ve Aritmetik İşlemler 10
Değişkenler 14
Metin Düzenleyici Kullanılarak Python Programı Nasıl Yazılır? 16
GNU/Linux Sistemi 17
Windows Sistemi 23
Türkçe Karakter Sorunu 24
Kullanıcıyla İletişim: Veri Alış-Verişi 30
raw_input() fonksiyonu 30
input() fonksiyonu 33
Güvenlik Açısından input() ve raw_input() 37
Kaçış Dizileri 39
Dönüştürme İşlemleri 41

2 PYTHON'DA KOŞULA BAĞLI DURUMLAR **45**
if 45
else 49
elif 50
Python'da Girintileme Sistemi 53

3 PYTHON'DA DÖNGÜLER **57**
while Döngüsü 58

for Döngüsü 64
range() fonksiyonu 65
len() fonksiyonu 66
break deyimi 67
continue deyimi 68
in işleci 69

4 PYTHON'DA LİSTELER, DEMETLER VE SÖZLÜKLER 71
Listeler 71
append 72
insert 75
extend 75
remove 76
pop 76
index 77
sort 78
reverse 78
count 78
Demetler 81
Sözlükler 82
Sıralı Sözlükler 87

5 PYTHON'DA FONKSİYONLAR 91
Fonksiyonları Tanımlamak 92
Fonksiyonlarda Parametre Kullanımı 97
İsimli Argümanlar 103
Gömülü Fonksiyonlar (Built-in Functions) 109
global Deyimi 110
return Deyimi 114
pass Deyimi 117

6 MODÜLLER 119
Modülleri İçe Aktarma (Importing Modules) 120
os Modülü 126

name Niteliği 128
listdir Fonksiyonu 129
getcwd Fonksiyonu 130
chdir Fonksiyonu 132
mkdir() ve makedirs() Fonksiyonları 133
rmdir() ve removedirs() Fonksiyonları 134
ossep Niteliği 136

7 DOSYA İŞLEMLERİ 141
Dosya Oluşturmak 141
Dosyaya Yazmak 145
Dosyayı Okumak 147
Dosya Silmek 151
Dosyaya Rastgele Satır Eklemek 152
Dosyadan Rastgele Satır Silmek 155

8 HATA YAKALAMA 157
try... except... 159
pass Deyimi 163

9 KARAKTER DİZİLERİN METOTLARI 165
Kullanılabilir Metotları Listelemek 166
capitalize metodu 168
upper metodu 169
lower metodu 169
swapcase metodu 169
title metodu 170
center metodu 170
ljust metodu 171
rjust metodu 171
zfill metodu 171
replace metodu 172
startswith metodu 173
endswith metodu 174

count metodu 175
isalpha metodu 175
isdigit metodu 176
isalnum metodu 176
islower metodu 177
isupper metodu 178
istitle metodu 178
isspace metodu 178
expandtabs metodu 179
find metodu 179
rfind metodu 181
index metodu 181
rindex metodu 182
join metodu 183
translate metodu 184
partition metodu 187
rpartition metodu 187
strip metodu 188
rstrip metodu 188
lstrip metodu 189
splitlines metodu 189
split metodu 189
rsplit metodu 191
Metotlarda Türkçe Karakter Sorunu 192

10 DÜZENLİ İFADELER (REGULAR EXPRESSIONS) 197
Düzenli İfadelerin Metotları 198
match() Metodu 199
search() Metodu 204
findall() Metodu 206
Metakarakterler 208
[ ] (Köşeli Parantez) 211
. (Nokta) 213
* (Yıldız) 215

+ (Artı) 218
? (Soru İşareti) 220
{ } (Küme Parantezi) 222
^ (Şapka) 222
$ (Dolar) 226
\ (Ters Bölü) 227
| (Dik Çizgi) 228
( ) (Parantez) 229
Eşleşme Nesnelerinin Metotları 232
group() metodu 232
groups() metodu 233
Özel Diziler 234
Ondalık Sayıların Yerini Tutan Özel Dizi: \d 235
Alfanümerik Karakterlerin Yerini Tutan Özel Dizi: \w 235
Düzenli İfadelerin Derlenmesi 237
compile() metodu 237
compile() ile Derleme Seçenekleri 239
Düzenli İfadelerle Metin/Karakter Dizisi Değiştirme İşlemleri 242
sub() metodu 242
subn() metodu 247
Sonuç 247

11 NESNE TABANLI PROGRAMLAMA OOP (NTP) 249
Neden Nesne Tabanlı Programlama? 249
Sınıflar 251
Sınıf Tanımlamak 251
Sınıfları Çalıştırmak 252
Örnekleme (Instantiation) 254
Çöp Toplama (Garbage Collection) 255
Niteliklere Değinme (Attribute References) 256
__init__ Nedir? 260
self Nedir? 263
Miras Alma (Inheritance) 272
Eski ve Yeni Sınıflar 282
Sonuç 284

12 ASCII, UNICODE VE PYTHON **285**
Giriş 285
ASCII 286
Unicode 294
Python'da Unicode Desteği 299
Python Betiklerinde Unicode Desteği 300
unicode() Fonksiyonu 302
encode() ve decode() Metotları 306
unicodedata Modülü 309
codecs Modülü 310

12 BİÇİM DÜZENLEYİCİLER **311**
Biçim Düzenlemede Kullanılan Karakterler 313
d Harfi 313
i Harfi 314
o Harfi 314
x Harfi 315
X Harfi 315
f Harfi 315
c Harfi 316
İleri Düzeyde Karakter Dizisi Biçimlendirme 318
Karakter Dizisi Biçimlendirmede 319
Sözlükleri Kullanmak 319
Sayılarda Hassas Biçimlendirme 320
Sayıların Soluna Sıfır Eklemek 321
Karakter Dizilerini Hizalamak 323
Karakter Dizilerini Hem Hizalamak 325
Hem de Sola Sıfır Eklemek 325
format() Metodu ile Biçimlendirme 326

KISIM II: ÖZEL KONULAR **329**

1 MATH MODÜLÜ **331**
Üslü İfadeler (pow) 332

PI Sayısı (pi) 332
Karekök (sqrt) 333
Euler Sabiti (e) 333
exp() Fonksiyonu 334
Logaritma (log) 334
log10() Fonksiyonu 334
degrees() Fonksiyonu 335
radians() Fonksiyonu 335
Kosinüs (cos) 335
Sinüs (sin) 336
Tanjant (tan) 337

2 PYTHON'DA ID() FONKSİYONU, IS İŞLECİ VE ÖNBELLEKLEME MEKANİZMASI 339

3 WINDOWS'TA PYTHON'U YOL'A (PATH) EKLEMEK 347

4 FARKLI PYTHON SÜRÜMLERİ 353

KISIM III: ARAYÜZ 361

1 GRAFİK ARAYÜZ TASARIMI / TEMEL BİLGİLER 363
Pencere Oluşturmak 364
Pencere Başlığı 372
Renkler 375
fg Seçeneği 375
bg Seçeneği 378
Yazı Tipleri (Fonts) 379
Metin Biçimlendirme 380
İmleçler 380
Pencere Boyutu 381
Tekrar 383

2 PENCERE ARAÇLARI (WIDGETS):1.BÖLÜM 387
Label Pencere Aracı 387
Button Pencere Aracı 388
Entry Pencere Aracı 394
Frame() 397

3 GEOMETRİ YÖNETİCİLERİ 401
pack() Geometri Yöneticisi 403
grid() Geometri Yöneticisi 415
place() Geometri Yöneticisi 422

4 PENCERE ARAÇLARI (WIDGETS):2.BÖLÜM 423
Checkbutton Pencere Aracı 423
Toplevel Pencere Aracı 430
Listbox Pencere Aracı 432
Menu Pencere Aracı 442
Text Pencere Aracı 446
Scrollbar Pencere Aracı 451

5 TKINTER UYGULAMALARINI GÜZELLEŞTİRMEK 457
Pencere Araçlarına Simge Eklemek 461
Pencere Araçlarına İpucu Metni (Tooltip) Eklemek 465

6 NASIL YAPILIR? 469
Tkınter'de Fare ve Klavye Hareketleri (Events and Bindings) 469
Listbox Öğelerine Görev Atamak 481
Pencereleri Başlıksız Hale Getirmek 483
Pencere / Ekran Koordinatları ve Boyutları 486
Programı Tam Ekran olarak Çalıştırmak 493
Ekranı Ortalamak 494
Pencereleri Her Zaman En Üstte Tutmak 495

7 STANDART BİLGİ PENCERELERİ (STANDARD DIALOGS) 499
Hata Mesajı Gösteren Pencere 500
Bilgi Mesajı Gösteren Pencere 508
Uyarı Mesajı Gösteren Pencere 510

KISIM I

TEMEL KONULAR

TEMEL BİLGİLER 1

Python Hakkında

Python, Guido Van Rossum adlı Hollandalı bir programcı tarafından yazılmış bir programlama dilidir. Geliştirilmesine 1990 yılında başlanan Python; C ve C++ gibi programlama dillerine kıyasla;

- daha kolay öğrenilir,
- program geliştirme sürecini kısaltır,
- bu programlama dillerinin aksine ayrı bir derleyici programa ihtiyaç duymaz,
- hem daha okunaklıdır, hem de daha temiz bir sözdizimine sahiptir.

Python'un bu ve buna benzer özellikleri sayesinde, dünya çapında ün sahibi büyük kuruluşlar (Google, Yahoo! ve Dropbox gibi) bünyelerinde her zaman Python programcılarına ihtiyaç duyuyor. Mesela; pek çok büyük şirketin Python bilen programcılara iş olanağı sunduğunu, Python'un baş geliştiricisi Guido Van Rossum'un 2005 ile 2012 yılları arasında Google'da çalıştığını, 2012 yılının sonlarına doğru ise Dropbox şirketine geçtiğini söylersek, bu programlama dilinin önemi ve geçerliliği herhalde daha belirgin bir şekilde ortaya çıkacaktır.

Bu arada, her ne kadar Python programlama dili ile ilgili çoğu görsel malzemenin üzerinde bir yılan resmi görsek de, Python kelimesi aslında çoğu kişinin zannettiğinin aksine 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.

Python'u Nereden Bulabilirim?

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

Python Windows ve GNU/Linux işletim sistemlerine kurulma açısından farklılıklar gösterir. Biz burada Python'un hem GNU/Linux'a hem de Windows'a nasıl kurulacağını ayrı ayrı inceleyeceğiz. Öncelikle GNU/Linux'tan başlayalım...

GNU/Linux

Python hemen hemen bütün GNU/Linux dağıtımlarında kurulu olarak geliyor. Mesela; Ubuntu'da Python'un kurulu olduğunu biliyoruz; o yüzden Ubuntu kullanıyorsanız, Python'u kurmanıza gerek yok. Ama eğer herhangi bir şekilde Python'u kurmanız gerekirse;

- Öncelikle şu adresi ziyaret edin: *https://www.python.org/downloads/*
- Bu adreste, üzerinde **"Download Python 2.x.y"** yazan sarı düğmeye tıklayarak, kaynak kodları içeren dosyayı bilgisayarınıza indirin. Ancak burada, yanlışlıkla üzerinde **"Download Python 3.x.y"** yazan düğmeye **tıklamamaya** dikkat ediyoruz. Eğer o düğmeye tıklarsak Python'un 3.x sürümünü indirmiş oluruz. Unutmayın, bu kitapta Python3'ü değil, Python2'yi ele alacağız.
- Daha sonra bu sıkıştırılmış dosyayı açın ve açılan dosyanın içine girip, orada sırasıyla aşağıdaki komutları verin.

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

Böylelikle Python'un farklı bir sürümünü sistemimize kurmuş olduk. 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 bulunabilir.

Bu noktada bir uyarı yapmadan geçmeyelim. Python özellikle bazı GNU/Linux dağıtımlarında pek çok sistem aracıyla sıkı sıkıya bağlantılıdır.

Yani Python, kullandığınız dağıtımın belkemiği durumunda olabilir. Bu yüzden Python'u kaynaktan kurmak bazı riskler taşıyabilir. Eğer yukarıda anlatıldığı şekilde, sisteminize kaynaktan Python kuracaksanız, karşı karşıya olduğunuz risklerin farkında olmalısınız. Ayrıca GNU/Linux üzerine kaynaktan program kurmak konusunda tecrübeli değilseniz; kaynaktan kuracağınız programların bağımlılıklarının neler olduğunu, bunları nasıl kuracağınızı, programı kurduktan sonra nasıl çalıştıracağınızı bilmiyorsanız 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'un en yeni sürümlerini rahatlıkla bulabilirsiniz. Ubuntu'nun resmi sitesine *www.ubuntu.com* adresinden, yerel Türkiye sitesine ise *www.ubuntu-tr.net* adresinden ulaşabilirsiniz. Gelelim Windows kullanıcılarının Python'u nereden bulacağına...

Microsoft Windows

Python programlama dilini Windows'a kurabilmek için öncelikle https://www.python.org/downloads/ adresini ziyaret ediyoruz. Burada, üzerinde 'Download Python 3.x.y' ve 'Download Python 2.x.y' yazan, yan yana iki sarı düğme göreceksiniz. 3.x.y numaralarını içeren düğme Python'un 3.x sürümünü, 2.x.y numaralarını içeren düğme ise Python'un 2.x sürümünü indirir. Biz bu kitapta Python'un 2.x sürümünü ele alacağımız için, 2.x.y numaralarını içeren düğmeye tıklamaya dikkat ediyoruz.

Python'u indirdikten sonra, programı kurarken, kurulumun bir aşamasında karşınıza 'Customize Python 2.x.y' başlıklı bir sayfa çıkacak. Bu sayfadaki listenin en altında, sol tarafında bir çarpı işareti olan ve 'Add python.exe to Path' ifadelerini içeren bir seçenek göreceksiniz. Burada çarpı işaretine tıklayarak 'Entire feature will be installed on local hard drive' öğresini seçin ve kuruluma devam edin.

Python Nasıl Çalıştırılır?

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

GNU/Linux'ta Python'u Çalıştırmak

Eğer GNU/Linux işletim sistemi üzerinde KDE kullanıyorsak, Python programını çalıştırmak için **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
```

UNITY masaüstü ortamında ise **CTRL+ALT+T** tuşlarına basarak doğrudan komut satırına ulaşabiliriz.

Bu şekilde komut satırına ulaştıktan sonra;

```
python
```

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

```
Python 2.7.11 (default, Dec 6 2015, 12:11:55)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-54)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

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.

İkincisi, eğer ilk maddede söylenenlerin geçerli olmadığından eminseniz, çok düşük bir ihtimal olmakla birlikte Python sisteminizde kurulu değil demektir. Yalnız GNU/Linux sistemlerinde Python'un kurulu olmama ihtimalinin imkânsıza yakın olduğunu söyleyeyim. O yüzden sisteminizde Python'un kurulu olmadığına kesinkes karar vermeden önce, durumunuzun birinci madde kapsamına girmediğinden emin olmalısınız.

Eğer Python'un, kullandığınız GNU/Linux dağıtımında kurulu olmadığına (ve dağıtımınızın depolarında Python için bir paket bulunmadığına) eminseniz, *https://www.python.org/downloads* adresinden **Python 2.x.y**'nin kaynak kodlarını 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'un 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.

Gelelim Microsoft Windows kullanıcılarına...

Windows'ta Python'u Çalıştırmak

Python'u yukarıda verdiğimiz indirme adresinden indirip bilgisayarlarına kurmuş olan Windows kullanıcıları, Python komut satırına **Başlat / Programlar / Python 2.7 / Python (Command Line)** yolunu takip ederek ulaşabilirler.

Ayrıca **Başlat / Çalıştır** yolunu takip ederek, cmd komutuyla ulaştığınız MS-DOS ekranında şu komutu verdiğinizde de karşınıza Python'un komut satırı gelecektir (Kullanışlılık açısından, Python komut satırına **Başlat / Çalıştır** yerine, bu şekilde MS-DOS aracılığıyla ulaşmanızı tavsiye ederim).

```
python
```

Eğer '**`python`**' komutu verdiğinizde **Python** başlamıyorsa, ya '**python**' kelimesini yanlış yazıyorsunuzdur ya da Python'u kurarken, yukarıda bahsettiğimiz şekilde **'Add python.exe to Path'** seçeneğini işaretlememişsinizdir. Bu seçeneği işaretlemeyi unuttuysanız, kurulum programını tekrar çalıştırıp ilk ekranda **'Change Python 2.7.x'** seçeneğini işaretleyerek **'Add python.exe to Path'** işlemini, yukarıda anlattığımız şekilde gerçekleştirebilirsiniz.

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 2.7.11 (default, Dec 6 2015, 19:54:46)
[MSC v.1500 32 bit (Intel)] on win32
```

```
Type "help", "copyright", "credits" or "license"
for more information.
>>>
```

Bu ekranda kullandığımız Python sürümünün **2.7.11** olduğunu görüyoruz. Buradaki **>>>** işareti Python'un 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.

Python'u nasıl çalıştıracağımızı öğrendiğimize göre artık ilk programımızı yazabiliriz. İşe çok bilindik, basit bir komutla başlayalım.

PRINT KOMUTU

Bir önceki bölümde Python'un komut satırına nasıl ulaşacağımızı görmüştük. (Bu komut satırına Pythonca'da *"Etkileşimli Kabuk"* veya *"Yorumlayıcı"* adı verilir.) Şimdi yukarıda anlattığımız yöntemlerden herhangi birini kullanarak Python'un etkileşimli kabuğunu açalım ve şuna benzer bir ekranla karşılaşalım:

```
Python 2.7.11 (default, Dec 6 2015, 12:11:55)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-54)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Komutlarımızı bu **>>>** işaretinden hemen sonra, hiç boşluk bırakmadan yazmaya başlayacağımızı daha önce söylemiştik.

Bu bölümde inceleyeceğimiz ilk komutumuzun adı **`print`**.

`print` adlı komut, ekrana çıktı vermemizi sağlar. Mesela bu komutu tek başına kullanmayı deneyelim:

```
>>> print
```

yazıp hemen **ENTER** tuşuna basalım. (Bu arada yukarıdaki komutta gördüğümüz **>>>** işaretini kendimiz yazmayacağız. Bize düşen sadece **`print`** yazmak.)

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 "Ben Python, Monty Python!"
```

Bu satırı yazıp **ENTER** tuşuna bastıktan sonra ekranda *"Ben Python, Monty Python!"* çıktısını görmemiz gerekiyor.

`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.

Gördüğünüz gibi **`print`** komutunun ardından gelen *"Ben Python, Monty Python!"* ifadesini çift tırnak içinde belirtiyoruz.

Python programlama dilinde tırnak içinde gösterilen bu tür değerlere **karakter dizisi** (*string*) adı verilir. Karakter dizilerinin ayırt edici özelliği, tırnak işaretleri içinde tanımlanmış olmalarıdır. Eğer burada tırnak işaretlerini 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.

Burada biz istersek çift tırnak yerine tek tırnak (**'**) da kullanabiliriz:

```
>>> print 'Ben Python, Monty Python!'
```

Hatta istersek üç tırnak da kullanabiliriz:

```
>>> print '''Ben Python, Monty Python! '''
```

Karakter dizilerini tanımlarken hangi tırnak çeşidini kullandığımızın önemi yok. Önemli olan, karakter dizisini hangi tırnak çeşidiyle tanımlamaya başlamışsak, o tırnak çeşidiyle kapatmamızdır.

Ancak karakter dizilerini tanımlarken, karakter dizisi içindeki başka kesme ve tırnak işaretlerine karşı dikkatli olmalıyız.

Örneğin;

```
>>> print "Linux'un faydaları"
```

Bu komut bize hatasız bir şekilde *"Linux'un faydaları"* çıktısını verir.

Ancak aynı işlemi tek tırnakla yapmaya çalışırsak şöyle bir hata mesajı alırız:

```
File "<stdin>", line 1
  print 'Linux'un faydaları'
              ^
SyntaxError: invalid syntax
```

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

```
>>> print 'Linux\'un faydaları'
```

Aynı şekilde şu kodlar da hata verecektir:

```
 >>> print "Ahmet, "Adana'ya gidiyorum," dedi."
```

Buradaki hatanın sebebi de, karakter dizisini başlatıp bitiren tırnaklarla, Ahmet'in sözünü aktarmamızı sağlayan tırnak işaretlerinin birbirine karışmasıdır.

Bu hatayı da şu şekilde önleyebiliriz:

```
 >>> print "Ahmet, \"Adana'ya gidiyorum,\" dedi."
```

Buradaki **** işaretleri olası bir hatadan kaçmamızı sağlar. Bu tür ifadelere Python dilinde **kaçış dizileri** (*escape sequences*) adı verilir. Kaçış dizilerinden biraz sonra bahsedeceğiz. Şimdilik bu kaçış dizisi kavramına takılmadan yolumuza devam edelim.

Python'da **`print`** komutunun nasıl kullanıldığını gördüğümüze göre artık Python'un başka bir özelliğini anlatmaya başlayabiliriz.

Python'da Sayılar ve Aritmetik İşlemler

Python'da henüz dört dörtlük bir program yazamasak da en azından şimdilik onu basit bir hesap makinesi niyetine kullanabiliriz.

Örneğin:

```
>>> 2 + 5
7
```

... veya:

```
>>> 5 - 2
3
```

... ya da:

```
>>> 2 * 5
10
```

... hatta:

```
>>> 6 / 2
3
```

Gördüğünüz gibi, aritmetik işlemler için şu işleçleri (*operator*) kullanıyoruz:

Toplama için: **+** işareti

Çıkarma için: **-** işareti

Çarpma için: ***** işareti

Bölme için: **/** işareti

Bunların dışında, işimize yarayacak birkaç işleç daha öğrenelim:

```
>>> 2 ** 2
4
>>> 2 ** 3
8
>>> 6 % 3
0
>>> 9 % 2
1
```

Burada gördüğümüz ****** işleci kuvvet hesaplama işlemleri için kullanılır. Mesela; yukarıdaki iki örnekte sırasıyla 2 sayısının 2. ve 3. kuvvetlerini hesapladık.

Tahmin edebileceğiniz gibi ****** işlecini kullanarak bir sayının karesini ve karekökünü de rahatlıkla hesaplayabilirsiniz:

```
>>> 12 ** 2
144
```

Bir sayının 2. kuvveti o sayının karesidir. Aynı şekilde bir sayının 0.5. kuvveti de o sayının karekökü dür:

```
>>> 144 ** 0.5
12.0
```

Yukarıda gördüğümüz % işleci ise bölme işleminden kalan sayıyı gösterir. Örneğin; 6 sayısını 3'e böldüğümüzde, bu sayı 3'e tam bölündüğü için kalan 0'dır. Bir sonraki örnekte gördüğümüz 9 sayısı ise 2'ye bölündüğünde kalan 1 olduğu için 9 % 2 işleminin değeri 1 olacaktır. Bu işleci kullanarak tek ve çift sayıları tespit edebilirsiniz. Eğer bir sayı 2'ye bölündüğünde kalan 0 ise o sayı çifttir, aksi halde o sayı tektir:

```
>>> 10 % 2
0
```

10 sayısı 2'ye bölündüğünde kalan 0. Dolayısıyla 10, bir çift sayıdır.

```
>>> 5 % 2
1
```

5 sayısı ise 2'ye bölündüğünde kalan 0 değildir. Dolayısıyla 5, bir tek sayıdır.

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:

```
>>> 25 + 50
75
```

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

```
>>> "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?

```
>>> "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:

```
>>> "25" + "50"
2550
```

Uzun lafın kısası, 25 ifadesi ile *"Ben Python, Monty Python!"* ifadesi arasında Python açısından hiç bir fark yoktur. Bunların ikisi de karakter dizisi sınıfına girer. Ancak tırnak işareti olmayan 25 ile *"Ben Python, Monty Python!"* ifadeleri Python dilinde ayrı anlamlar taşır. Çünkü bunlardan biri **tamsayı** (*integer*) öteki ise **karakter dizisi**dir (*string*).

Şimdi matematik işlemlerine geri dönelim. Öncelikle şu komutun çıktısını inceleyelim:

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

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

```
>>> 5.0 / 2
2.5
```

Gördüğünüz gibi bölme işlemini oluşturan bileşenlerden birinin yanına `.0` koyulursa sorun çözülüyor. Böylelikle Python bizim sonucu tamsayı yerine **kayan noktalı** (*floating point*) sayı cinsinden görmek istediğimizi anlıyor. Bu *".0"* ifadesini istediğimiz sayının önüne koyabiliriz. Birkaç örnek görelim:

```
>>> 5 / 2.0
2.5
>>> 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:

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

işleminin sonucu 8 olacaktır. Tabii biz istersek parantezler yardımıyla Python'un 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
```



Burada __ işaretini iki kez art arda klavyedeki alt çizgi tuşuna basarak yapabilirsiniz.

Artık bir sonraki Python oturumuna kadar bütün işlemlerinizin sonucu kayan noktalı sayı cinsinden gösterilecektir.

Buraya kadar Python'da 3 tane **veri tipi** (*data type*) olduğunu gördük. Bunlar:

- Karakter dizileri (*strings*)
- Tamsayılar (*integers*)
- Kayan noktalı sayılar (*floating point numbers*)

Python'da bunların dışında başka veri tipleri de bulunur. Ama biz şimdilik veri tiplerine ara verip değişkenler konusuna değinelim biraz.

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 n komutunu verirsek ekrana yazdırılacak veri 5 olacaktır.

Yani:

```
>>> n = 5
>>> 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
>>> n
1530
```

Şu örneklere bir göz atalım:

```
>>> a = 5
>>> b = 3
>>> 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"
```

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" %(a, b, a*b)
```

Burada, parantez içinde göstereceğimiz her bir öğe için karakter dizisi içine **%s** işaretini ekliyoruz. Karakter dizisini yazdıktan sonra da **%** işaretinin ardından parantez içinde bu işaretlere karşılık gelen değerleri teker teker tanımlıyoruz. Buna göre birinci değerimiz **a** (yani 5), ikincisi **b** (yani 3), üçüncüsü ise bunların çarpımı (yani 5 * 3)…

Bu yapıyı daha iyi anlayabilmek için bir iki örnek daha verelim:

```
>>> print "%s ve %s iyi bir ikilidir." %("Python", "Django")
>>> print "%s sayısının karesi %s sayısıdır." %(12, 12**2)
>>> print "Adım %s, soyadım %s" %("Fırat", "Özgül")
```

Gördüğünüz gibi, **%s** işaretleri ile hem değişkenleri hem de doğrudan değerleri kullanabiliyoruz. Ayrıca bu işaretler, bir karakter dizisi içine çeşitli değerleri kolaylıkla yerleştirmemizi de sağlıyor.

Metin Düzenleyici Kullanılarak Python Programı Nasıl Yazılır?

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. Ancak içine yazılan kodları ayırt edebilen, bunları farklı renklerde gösterebilen bir metin düzenleyici ile yola çıkmak her bakımdan hayatınızı kolaylaştıracaktır.

Biz bu bölümde metin düzenleyici kullanılarak Python programlarının nasıl yazılacağını GNU/Linux ve Windows işletim sistemleri için ayrı ayrı inceleyeceğiz. Öncelikle GNU/Linux'tan başlayalım.

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 veya UNITY 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.

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

```
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
```

GNU/Linux sistemlerinde komut satırını ilk başlattığınızda içinde bulunduğunuz dizin ev dizininiz olacaktır (*"/home/kullanıcı_adı"*). 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 /home/kullanıcı_adı/Desktop
```

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
```

Bu arada, kodları Python'un etkileşimli kabuğuna yazmakla metin düzenleyici içine yazmak arasındaki önemli bir fark, **`print`** komutuyla ilgilidir. Etkileşimli kabukta **`print`** komutunu kullansak da kullanmasak da (şeklen farklılıklar olmakla birlikte) değerler ekrana basılacaktır. Ama metin düzenleyicide başına **`print`** koymadığımız ifadeler ekrana çıktı olarak verilmez. Bu durumu daha iyi anlayabilmek için kendi kendinize denemeler yapmanızı tavsiye ederim.

Gördüğünüz gibi, **`python deneme.py`** komutuyla programlarımızı çalıştırabiliyoruz. Peki, ama acaba Python programlarını başa **`python`** komutu eklemeden çalıştırmanın bir yolu var mı? Elbette var, ancak bunun için öncelikle bazı işlemler yapmamız gerekiyor.

Başa **`python`** komutu getirmeden programımızı çalıştırabilmek için öncelikle programımızın ilk satırına şu kodu ekliyoruz:

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

Yani programımız tam olarak şöyle görünecek:

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

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

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

Peki, programımızın en başına yerleştirdiğimiz bu komut ne işe yarıyor?

Bu komut, biraz sonra yazacağımız kodların birer Python kodu olduğunu gösteriyor ve Python'un çalıştırılabilir dosyasının nerede olduğunu bulmak konusunda işletim sistemimize yardımcı oluyor. Bu konuyu birazdan daha ayrıntılı olarak inceleyeceğiz.

Kullandığımız metin düzenleyicinin ilk satırına yukarıda verilen ifadeyi yerleştirdikten sonra programımıza çalıştırma yetkisi vermeliyiz. Bunu şu komut yardımıyla yapıyoruz:

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

Bu komutla **deneme.py** adlı dosyayı **çalıştırılabilir** (*executable*) bir dosya haline getirmiş oluyoruz.

Artık komut satırında şu komutu vererek programımızı çalıştırabiliriz:

```
./deneme.py
```

Gelin isterseniz yukarıda yaptığımız işlemleri biraz mercek altına alalım.

Biz bir Python programını **`python program_adı.py`** komutuyla çalıştırdığımızda, baştaki **`python`** ifadesi sayesinde bu programı Python programlama dilinin çalıştıracağı anlaşılıyor. Böylece işletim sistemimiz programımızı doğru bir şekilde çalıştırabiliyor. Ancak başa **`python`** ifadesini getirmediğimizde programımızın hangi programlama dili tarafından çalıştırılacağı anlaşılmıyor. İşte programımız içine eklediğimiz **`#!/usr/bin/env python`** satırı programımızı hangi programlama dilinin çalıştıracağını ve bu programlama dilinin işletim sistemi içinde hangi konumda bulunduğunu (**`env`** adlı bir betik aracılığıyla) işletim sistemimize haber veriyor.

İsterseniz neler olup bittiğini daha iyi anlayabilmek için basit bir deneme çalışması yapalım.

Metin düzenleyici içine sadece şu kodları yazalım:

```
a = "elma"
b = "armut"
c = "muz"
print "bir", a, "bir", b, "bir de", c, "almak istiyorum"
```

Dosyamızı **deneme.py** adıyla kaydedelim.

Şimdi programımızı çalıştırmayı deneyelim:

```
./deneme.py
```

Bu komutu verdiğimizde şu hata mesajını alacağız:

```
-bash: ./deneme.py: Permission denied
```

Bu hatanın sebebi, programımıza henüz çalıştırma yetkisi vermemiş olmamız. O yüzden öncelikle programımıza çalıştırma yetkisi verelim:

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

Şimdi tekrar deneyelim:

```
./deneme.py
```

Bu defa da şu hata mesajını aldık:

```
./deneme.py: line 1: a: command not found
./deneme.py: line 2: b: command not found
./deneme.py: line 3: c: command not found
./deneme.py: line 5: print: command not found
```

İşte bu hatanın sebebi, işletim sistemimizin programımızı nasıl çalıştırması gerektiğini bilmemesidir.

Şimdi program dosyamızı açıp ilk satıra şu ibareyi ekleyelim:

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

Artık programımızı çalıştırabiliriz:

```
./deneme.py
```

Bu 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?

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/kerberos/sbin:/usr/kerberos/bin:
/usr/local/sbin:/usr/local/bin:/sbin:
/bin:/usr/sbin:/usr/bin:/root/bin
```

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.

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 görecektir:

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

Şimdi konsol ekranında:

```
deneme
```

yazınca, programımızın çalıştığını görmemiz lazım.

Gelelim Windows kullanıcılarına...

Windows Sistemi

Windows kullanıcıları **IDLE** adlı metin düzenleyici ile çalışabilirler. **IDLE**'a ulaşmak için **Başlat / Programlar / Python / IDLE (Python GUI)** yolunu takip ediyoruz. IDLE'ı çalıştırdığımızda gördüğümüz **>>>** işaretinden de anlayabileceğimiz gibi, bu ekran aslında Python'un etkileşimli kabuğunun ta kendisidir... Dilerseniz, etkileşimli kabukta yaptığınız 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 File** düğmesine tıklayarak boş bir sayfa açıyoruz.

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.

Eğer programınızı IDLE üzerinden değil de, doğrudan MS-DOS komut satırını kullanarak çalıştırmak isterseniz şu işlemleri yapın:

Başlat / Çalıştır yolunu takip edip, açılan pencereye `cmd` yazın ve **ENTER** tuşuna basın.

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
```

Eğer her şey yolunda gitmişse şu çıktıyı almamız lazım:

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

Böylece, Python programlama dilinde ilerlememizi kolaylaştıracak en temel bilgileri edinmiş olduk. Bu temel bilgileri cebimize koyduğumuza göre, artık gönül rahatlığıyla ileriye bakabiliriz. Bir sonraki bölümde programlama maceramızı bir adım öteye taşıyacak bir konuya değineceğiz. Bu konunun sonunda artık Python programlama dilini kullanarak az çok işe yarar şeyler üretebileceğiz. O halde lafı hiç dolandırmadan yolumuza devam edelim.

Türkçe Karakter Sorunu

Bu bölümde çok önemli bir sorundan bahsedeceğiz. Hatta belki de buraya kadar işlediğimiz konularla ilgili kendi kendinize alıştırma amaçlı örnek kodlar yazarken, birazdan bahsedeceğimiz bu sorunla zaten karşılaşmış bile olabilirsiniz. Örneğin; bir metin düzenleyiciye şöyle bir kod yazdığınızda:

```
print "Python zevkli bir programlama dilidir."
```

Programınız gayet güzel çalışarak şu çıktıyı veriyordur:

```
Python zevkli bir programlama dilidir.
```

Ama şöyle bir kod yazdığınızda ise:

```
print "Python Guido Van Rossum'un geliştirdiği bir dildir."
```

Programınız şöyle bir hata mesajı veriyordur:

```
File "deneme.py", line 1
SyntaxError: Non-ASCII character '\x92' in file deneme.py
on line 1, but no encoding declared; see
http://www.python.org/peps/pep-0263.html for details
```

Sizin de tahmin etmiş olabileceğiniz gibi, ilk kodun çalışıp, ikinci kodun çalışmamasının sebebi ikinci kodda yer alan Türkçe karakterlerdir. Peki, ama neden?

Dikkat ettiyseniz bugün günlük yaşamımızda sıklıkla kullandığımız pek çok sistem, bizden Türkçe karakterler girmememizi ister. Örneğin; Gmail'den bir e-posta hesabı alabilmek için belirlememiz gereken parolada Türkçe karakterler yer alamaz. Bunun sebebi; çok kaba bir şekilde ifade etmek gerekirse, bilgisayarların aslında yalnızca İngilizceye özgü harfleri göstermek üzere tasarlanmış olmasıdır. Esasında bütün bilgisayarların güvenilir bir şekilde görüntüleyebileceği karakterler *http://belgeler.istihza.com/py3/_images/asciifull.png* adresinde göreceğiniz karakter kümesindekilerden ibarettir. Bu adreste gördüğünüz karakter tablosuna **ASCII Tablosu** (veya ASCII standardı ya da ASCII dil kodlaması) adı verilir. Biz ilerleyen derslerde ASCII kavramından ayrıntılı olarak bahsedeceğiz. Şimdilik sadece bu tablonun bütün bilgisayarlarda güvenilir bir şekilde görüntülenebilecek harflerin, simgelerin, işaretlerin tamamını temsil ettiğini bilelim yeter.

İşte Python programlama dili de ASCII standardı üzerine kurulmuş bir sistemdir. Bu durumu ispatlamak için şu kodları kullanabilirsiniz:

```
>>> import sys
>>> sys.getdefaultencoding()

'ascii'
```

Gördüğünüz gibi, buradan aldığımız çıktı *'ascii'*. Bu çıktı bize Python programlama dilinin öntanımlı dil kodlamasının *'ASCII'* olduğunu söylüyor.

Hatta dikkatli baktıysanız, *"Python Guido Van Rossum'un geliştirdiği bir dildir."* ifadesini içeren programımızı çalıştırmak istediğimizde aldığımız hata mesajında da ASCII kavramını görmüş olmalısınız:

```
File "deneme.py", line 1
SyntaxError: Non-ASCII character '\x92' in file deneme.py
on line 1, but no encoding declared; see
http://www.python.org/peps/pep-0263.html for details
```

Burada Python bize tam olarak şunu söylüyor:

```
"deneme.py" dosyası, 1. satır
SözdizimiHatası: deneme.py dosyasının 1. satırında
ASCII olmayan, '\x92' adlı bir karakter var, ama
herhangi bir dil kodlaması belirtilmemiş. Ayrıntılar
için http://www.python.org/peps/pep-0263.html adresine bakınız.
```

Dediğimiz gibi, Python ASCII sistemi üzerine kurulduğu için, bu sistemde yer almayan karakterleri gösteremiyor. Dolayısıyla Türkçe karakterler de bu karakter kümesi içinde yer almadığı için, bu sistem ile gösterilemiyor.



Siz şimdilik ASCII, dil kodlaması, standart, karakter kümesi gibi kavramlara takılmayın. İlerde bu kavramları ayrıntılı bir şekilde inceleyeceğiz. Şu anda sadece bu sistemin sonuçları üzerine odaklanmamız yeterli olacaktır.

Peki, biz bu durumda Türkçe karakterleri programlarımızda kullanamayacak mıyız?

Elbette kullanacağız. Ama bunu yaparken dikkat etmemiz gereken bazı kurallar var. İşte bu bölümde bu kuralların neler olduğunu anlatacağız.

Yazdığımız programlarda Türkçe karakterleri kullanabilmek için yapmamız gereken ilk işlem, kodlarımızı yazmak için kullandığımız metin düzenleyicinin **dil kodlaması** (*encoding*) ayarlarını düzgün yapmaktır. Ben size dil kodlaması olarak GNU/Linux'ta utf-8'i, Windows'ta ise **cp1254**'ü (Windows-1254) seçmenizi tavsiye ederim. Dediğimiz gibi, Python'un öntanımlı dil kodlaması olan ASCII Türkçe karakterleri gösteremez, ama utf-8 ve cp1254 Türkçe karakterleri düzgün bir şekilde görüntüleyebilme kabiliyetine sahip iki dil kodlamasıdır.

Metin düzenleyicimiz için uygun bir dil kodlaması seçip gerekli ayarlamaları yapıyoruz. Elbette etrafta yüzlerce metin düzenleyici olduğu için bu ayarların her metin düzenleyicide nasıl yapılacağını tek tek göstermemiz mümkün değil. Ancak iyi bir metin düzenleyicide bu ayar bulunur. Tek yapmanız gereken, bu ayarın, kullandığınız metin düzenleyicide nereden yapıldığını bulmak. Eğer kullandığınız metin düzenleyiciyi ayarlamakta zorlanıyorsanız, *www.istihza.com/forum* adresinde sıkıntınızı dile getirebilirsiniz.

Dil kodlamasını ayarladıktan sonra, kodlarımızın ilk satırına şu ifadelerden birini yazıyoruz:

GNU/Linux için:

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

Windows için:

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



Aslında her iki işletim sisteminde de `# -*- coding: utf-8 -*-` satırı aracılığıyla utf-8 adlı dil kodlamasını kullanabilirsiniz. Ancak cp1254 adlı dil kodlaması, Windows'ta Türkçe için biraz daha uygun görünüyor. Bazı durumlarda, özellikle MS-DOS komut satırının utf-8 ile kodlanmış programlardaki Türkçe karakterleri düzgün gösteremediğine tanık olabilirsiniz.

Ardından Türkçe karakter içeren program kodlarımızı ekleyebiliriz. Örneğin:

GNU/Linux'ta:

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

print "Merhaba zalim dünya!"
```

Windows'ta:

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

print "Merhaba zalim dünya!"
```

Gördüğünüz gibi bu kodlar arasındaki tek fark, kullanılan dil kodlaması: GNU/Linux için utf-8, Windows için ise cp1254 adlı dil kodlamasını kullandık (Tabii bir de Windows'ta herhangi bir işlevi olmayan `#!/usr/bin/env python` satırını Windows'taki kodlardan çıkardık).

Bu arada, zaten eğer IDLE üzerinde çalışıyorsanız programınızı herhangi bir dil kodlaması belirtmeden kaydetmeye çalıştığınızda şöyle bir uyarı penceresiyle karşılaşabilirsiniz:



Burada IDLE, dil kodlamasını belirtmeniz konusunda sizi uyarıyor. Eğer bu ekranda Edit My File düğmesine basacak olursanız, IDLE programınızın ilk satırına sizin yerinize `# -*- coding: cp1254 -*-` komutunu ekleyecektir...

Bu kodları bu şekilde yazıp, GNU/Linux dağıtımlarında çalıştırdığımızda, Türkçe karakter içeren programlarımız düzgün bir şekilde çalışır. Ancak Windows için yapmamız gereken bir işlem daha var.

Kodlarımıza eklediğimiz yukarıdaki satır, program dosyalarımıza Türkçe karakterleri girebilmemizi sağlıyor. Ancak programlarımızda Türkçe karakterleri gösterebilmek için önemli bir şart daha var: Programlarımızı çalıştırırken kullandığımız komut ekranının da Türkçe karakterleri gösterebiliyor olması lazım. GNU/Linux işletim sistemlerinde komut ekranı Türkçe karakterleri gösterebilir. Ancak kullandığınız Windows sürümünün komut ekranında (MS-DOS) öntanımlı olarak Türkçe karakterleri gösteremeyen bir yazı tipi (ve dil kodlaması) tanımlanmış olabilir. Eğer durum böyleyse, öncelikle MS-DOS'un öntanımlı yazı tipini (ve dil kodlamasını) değiştirmemiz gerekir.

Gelin isterseniz bahsettiğimiz bu durumu somutlaştıran bir örnek verelim.

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

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

santigrat = 22
fahrenhayt = santigrat * (9.0/5.0) + 32

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

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

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

Gördüğünüz gibi Türkçe karakterler bozuk görünüyor.

Bu durumu düzeltmek için şu adımları takip ediyoruz:

- Önce MS-DOS ekranını açıyoruz.
- Ardından şu komutu vererek dil kodlamasını **cp1254** olarak değiştiriyoruz:

```
chcp 1254
```

- Daha sonra pencere başlığına sağ tıklayıp **Özellikler**'e giriyoruz.
- **Yazı Tipi** sekmesi içinde yazı tipini **Lucida console** (veya varsa **Consolas**) olarak değiştirerek Türkçe karakterleri gösterebilen bir yazı tipi seçiyoruz.
- **Tamam**'a basıyoruz.
- Eğer karşımıza **Özellikleri Uygula** başlıklı bir pencere çıkarsa, burada **Özellikleri, aynı başlıkla ileride oluşturulacak pencereler için kaydet** seçeneğini işaretliyoruz.



Eğer yazı tipi ayarlama işleminin nasıl yapılacağını görüntülü olarak da öğrenmek isterseniz *http://media.istihza.com/videos/ms-dos.swf* adresindeki videomuzu izleyebilirsiniz.

Bu işlemleri yaparken takip etmemiz gereken temel prensip şu: Programımızda hangi dil kodlamasını kullanıyorsak, metin düzenleyici ve komut ekranında da aynı dil kodlamasını ayarlamamız gerekir. Mesela yukarıdaki örnekte, programımız için cp1254 adlı dil kodlamasını seçtik. Aynı şekilde, kullandığımız metin

düzenleyicinin dil kodlaması ayarlarında da cp1254'ü tercih ettik. Bütün bunlara uygun olarak, MS-DOS komut satırının dil kodlamasını da 1254 olarak ayarladık. Ayrıca MS-DOS için, Türkçe karakterleri gösterebilen bir yazı tipi belirlemeyi de unutmadık.

Esasında bu konuyla ilgili olarak söylenecek çok şey var. Çünkü aslında bu konu burada anlattığımızdan biraz daha karmaşıktır. Ancak bu aşamada kafa karışıklığına yol açmamak için bu konuyu burada kesip yeni bir konuya geçiyoruz. Gerekli ayrıntılardan ilerleyen derslerimizde söz edeceğiz.

Kullanıcıyla İletişim: Veri Alış-Verişi

Bu bölümde Python'daki çok önemli bir konudan söz edeceğiz. 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.

Python'da kullanıcıdan birtakım veriler alabilmek, yani kullanıcıyla iletişime geçebilmek için iki tane fonksiyondan faydalanılır. (Bu fonksiyon kelimesine takılmayın. Birkaç bölüm sonra fonksiyonun ne demek olduğunu inceleyeceğiz) .

Bunlardan öncelikle ilkine bakalım.

raw_input() fonksiyonu

`raw_input()` fonksiyonu kullanıcılarımızın veri girmesine imkân tanır. İsterseniz bu fonksiyonu tarif etmeye çalışmak yerine hemen bununla ilgili bir örnek verelim. Öncelikle boş bir metin belgesi açalım. Eğer GNU/Linux kullanıyorsak her zaman yaptığımız gibi, ilk satırımızı ekleyelim belgeye (Windows'ta bu satırı yazmanızın bir zararı veya gereği yok):

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

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!"
```



Ben bu kitaptaki kodlarda dil kodlaması olarak utf-8'i tercih edeceğim. Eğer kullandığınız işletim sistemi başka bir dil kodlaması kullanmanızı gerektiriyorsa, utf-8 yerine o dil kodlamasını yazmalısınız (örneğin; cp1254).

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
```

İsterseniz şimdi yazdığımız bu programı biraz geliştirelim. Mesela; programımız şu işlemleri yapsın:

- Program ilk çalıştırıldığında kullanıcıya parola sorsun,
- Kullanıcı parolasını girdikten sonra programımız kullanıcıya teşekkür etsin,
- Bir sonraki satırda kullanıcı tarafından girilen bu parola ekrana yazdırılsın,
- 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("Lütfen parolanızı girin:")
```

Şimdi kullanıcıya teşekkür ediyoruz:

```
print "Teşekkürler!"
```

Kullanıcı tarafından girilen parolayı ekrana yazdırmak için şu satırı ekliyoruz:

```
print "Girdiğiniz parola: ", parola
```

Biraz önce **raw_input()** fonksiyonunu neden bir değişkene atadığımızı anladınız sanırım. Bu sayede doğrudan **parola** değişkenini çağırarak kullanıcının yazdığı şifreyi ekrana dökebiliyoruz.

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("Lütfen parolanızı girin:")

print "Teşekkürler!"
print "Girdiğiniz parola: ", 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)
```

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

INPUT() FONKSİYONU

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 sayılardan bahsederken de karşılaşmıştık. O zaman şu örnekleri vermiştik:

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

İşte **`raw_input()`** ile **`input()`** arasındaki fark, yukarıdaki iki örnek arasındaki farka benzer. Yukarıdaki örneklerde, Python'un sayılara ve karakter dizilerine nasıl davrandığını görüyoruz. Eğer Python iki adet sayıyla karşılaşırsa, bu sayıları birbiriyle **topluyor**. İki adet karakter dizisiyle karşılaşırsa, bu karakter dizilerini **birleştiriyor**. Bir sayı, bir de karakter dizisi ile karşılaşırsa (sayılarla karakter dizileri arasında herhangi bir birleştirme işlemi veya aritmetik işlem yapılamayacağı için) **hata veriyor**:

```
>>> "34" + 34
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

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 -*-
soru = "Bir sayı girin, o sayının karesini söyleyeyim:"
sayi = raw_input(soru)
print "girdiğiniz sayı: %s" %(sayi)
print "girdiğiniz sayının karesi: %s" % (sayi ** 2)
```

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

```
Traceback (most recent call last):
  File "deneme.py", line 8, in <module>
    print "girdiğiniz sayının karesi: %s" % (sayi ** 2)
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

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 örneklere bakabilirsiniz:

```
>>> a = '12'
>>> b = '2'
>>> a ** b

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'str'
```

Gördüğünüz gibi biraz öncekiyle benzer bir hata 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 = 2
>>> 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 -*-

soru = "Bir sayı girin, o sayının karesini söyleyeyim:"
sayi = input(soru)
```

```
print "girdiğiniz sayı: %s" %(sayi)
print "girdiğiniz sayının karesi: %s" % (sayi ** 2)
```

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 -*-

isim = raw_input("isminiz: ")
soyisim = raw_input("soyisminiz: ")

print isim, soyisim
```

Eğer bu kodları **input()** fonksiyonuyla yazmayı denediyseniz, Python'un 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'un **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'un 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 bölümde verdiğimiz fahrenhayt'a dönüştürme programını, yeni öğrendiğimiz bilgiler yardımıyla biraz geliştirelim:

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

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

fahrenhayt = santigrat * (9.0/5.0) + 32

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

Böylece elimizde gayet şık bir derece dönüştürme programı olmuş oldu. Günün birinde santigrat dereceleri fayhrenhayt'a dönüştürmeniz gerekirse yukarıdaki programı kullanabilirsiniz...

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 alışverişine 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.

Kaçış Dizileri

Hatırlarsanız bu bölümün ilk kısımlarında şöyle bir örnek vermiştik:

```
>>> print 'Linux\'un faydaları'
```

Burada kullandığımız \ işaretine dikkat edin. Bu işaretin yukarıdaki görevi, tanımlamaya tek tırnakla başladığımız karakter dizisi içinde yine bir tek tırnak işareti kullanabilmemizi sağlamaktır. Bu örneği verdiğimizde, bu \ işaretine teknik olarak kaçış dizisi adı verildiğini söylemiştik. Peki, **kaçış dizisi** tam olarak ne anlama gelir?

Kaçış dizileri, kendilerinden hemen sonra gelen karakterlerle birleşerek, bu karakterin farklı veya özel bir anlam kazanmasını sağlayan birtakım işaretlerdir. Örneğin; Python programlama dilinde tek tırnak ve çift tırnak işaretleri karakter dizilerini tanımlamak için kullanılır. Eğer siz yazdığınız bir kodda tek veya çift tırnak işaretlerini farklı bir amaçla kullanacaksanız, kaçış dizilerinden yararlanmanız gerekir. Birkaç örnek verelim:

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

Eğer bu örneği bu şekilde yazıp çalıştırırsanız Python size bir hata mesajı gösterecektir. Çünkü biz burada tek tırnak işaretini '*İstanbul*' kelimesini '*un*' ekinden ayırmak amacıyla, kesme işareti anlamında kullandık. Python ise bu tırnak işaretini karakter dizisi tanımını bitiren işaret zannetti. Eğer yukarıdaki kodun çalışmasını istiyorsak \ adlı kaçış dizisinden yararlanmalıyız:

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

Aynı şekilde çift tırnak işareti de Python programlama dilinde karakter dizilerini tanımlamak için kullanılır. Eğer biz bu işareti başka bir amaçla kullanacaksak, Python'a bu isteğimizi bildirmeliyiz. Bu bildirme işini kaçış dizileri yardımıyla yapabiliriz:

```
>>> print "Ahmet, "Yarın ben Adana'ya gidiyorum," dedi."
```

Eğer bu kodları bu şekilde yazıp çalıştırırsak, Python'un bize bir hata mesajı göstereceğini biliyorsunuz. Çünkü Python'un, bu kodlar içindeki çift tırnak işaretlerinin hangisinin karakter dizisini başlatıp bitiren tırnaklar olduğunu, hangisinin ise, Ahmet'in sözlerini aktarmamıza yarayan tırnak işaretleri olduğunu ayırt etmesi mümkün değil. Bu konuda bizim Python'a yardımcı olmamız gerekiyor. Dikkatlice bakın:

```
>>> print "Ahmet, \"Yarın ben Adana'ya gidiyorum,\" dedi."
```

Gördüğünüz gibi, farklı bir amaçla kullandığımız her bir tırnak işaretini, kaçış dizileri yardımıyla Python'un gözünden **kaçırdık**.

Yukarıdaki örneklerden gördüğünüz gibi, \ işareti, tek ve çift tırnak işaretleri ile birleşerek, bu işaretleri karakter dizileri içinde farklı anlamlarda kullanmamızı sağlıyor. \ adlı kaçış dizisi başka karakterlerle de birleşebilir. Örneğin; bu kaçış dizisini n harfiyle birleştirdiğimizde, **yeni satır** adlı bir kaçış dizisini elde ederiz. Dikkatlice bakın:

```
>>> print "Birinci satır\nİkinci satır\nÜçüncü satır"
```

Bu kodları çalıştırdığımızda şöyle bir çıktı elde ediyoruz:

```
Birinci satır
İkinci satır
Üçüncü satır
```

Gördüğünüz gibi, Python **\n** işaretleri ile karşılaştığı her noktada yeni bir satıra geçti. İşte bunu sağlayan şey, \ işaretinin **n** karakteri ile birleşerek oluşturduğu **yeni satır** adlı kaçış dizisidir. Bildiğiniz gibi, **n** karakterinin görevi aslında ekrana **n** harfini basmaktır. Ama bu karakter \ işareti ile birleştiğinde özel bir anlam kazanarak **yeni satır** adını verdiğimiz kaçış dizisini oluşturuyor.

n adlı kaçış dizisi Python'da en fazla kullanacağınız kaçış dizilerinden biri, belki de birincisidir. O yüzden bu kaçış dizisini iyi öğrenmelisiniz.

\ kaçış dizisinin, birleştiğinde özel bir anlam kazanmasını sağladığı bir başka karakter de t karakteridir. Bildiğiniz gibi, **t** karakterinin görevi normalde ekrana **t** harfini basmaktır. Ama eğer biz bu karakteri \ işareti ile birleştirirsek ortaya **sekme** adlı bir kaçış dizisi çıkar.

Hemen bir örnek verelim:

```
>>> print "uzak...\tçok uzak"
uzak... çok uzak
```

Gördüğünüz gibi, **\t** adlı kaçış dizisi sanki **TAB** (*sekme*) tuşuna basmışız gibi bir etki ortaya çıkardı. Eğer bu etkiyi göremediyseniz birkaç tane **\t** kaçış dizisini art arda kullanabilirsiniz:

```
>>> print "uzak...\t\tçok uzak"
uzak...         çok uzak
```

Python'daki kaçış dizileri yukarıdakilerden ibaret değildir, ama bunlar arasında en önemlileri yukarıda verdiklerimizdir. Dolayısıyla özellikle ilk aşamada burada verdiğimiz kaçış dizilerini öğrenmeniz işlerinizin büyük bölümünü halletmenizi sağlayacaktır. Ayrıca ilerleyen derslerde, sırası geldikçe öteki kaçış dizilerinden de söz edeceğimizi belirtelim.

Dönüştürme İşlemleri

Programcılık maceranız sırasında, verileri birbirine dönüştürmeniz gereken durumlarla sıkça karşılaşacaksınız. 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.

Hatırlarsanız, **input()** ve **raw_input()** fonksiyonlarını incelerken, **input()** fonksiyonuyla yapabileceğimiz her şeyi aslında **raw_input()** fonksiyonuyla da yapabileceğimizi söylemiştik. İşte **int()** adlı bir fonksiyon, 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 tümden **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.

Gelin isterseniz bir örnek daha vererek ne ile karşı karşıya olduğumuzu daha iyi anlamaya çalışalım:

```
#!/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 kaçıncı kuvvetini "
                       "hesaplamak istediğinizi söyleyin: "
                       %sayi))

print "%s sayısının %s. kuvveti %s olur." %(sayi, kuvvet,
                                            sayi ** kuvvet)
```

Burada yazdığımız kodların nasıl işlediğine dikkat etmenin yanısıra, 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'
```

Böylece 23 sayısını, bir karakter dizisi olan 23'e dönüştürmüş olduk.

Gördüğünüz gibi, **`int()`** ve **`str()`** adlı fonksiyonlar yardımıyla karakter dizileri ve tamsayılar arasında dönüştürme işlemi yapabiliyoruz. Eğer bir sayıyı veya sayı değerli karakter dizisini kayan noktalı sayıya dönüştürmek istersek de **`float()`** adlı fonksiyondan yararlanacağız:

```
>>> a = 23
>>> float(a)

23.0

>>> float("34")
34.0
```

Python programlama dilinde **Temel Bilgiler** konusunu tamamladığımıza göre, artık **Python'da Koşula Bağlı Durumlar** adlı önemli bir konuyu incelemeye geçebiliriz.

PYTHON'DA KOŞULA BAĞLI DURUMLAR

2

Python'da en önemli konulardan biri de 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 Ubuntu'da konsol ekranından güncelleme işlemi yapacaksınız. **`sudo apt-get upgrade`** komutunu verdiğiniz zaman, güncellemelerin listesi size bildirilecek, bu güncellemeleri yapmak isteyip istemediğiniz size sorulacaktır. Eğer evet cevabı verirseniz güncelleme işlemi başlayacaktır. Yok, eğer hayır cevabı verirseniz 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`**, **`else`** ve **`elif`**

IF

`If` sözü İngilizce'de "*eğer*" anlamına geliyor. Dolayısıyla, adından da anlaşılabileceği 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..."

Bu arada, 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 ve kodların içine nasıl açıklama ekleneceğinden bahsedelim kısaca.

Öncelikle girintilemeden bahsedelim. 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. 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'un kafasının karışmasına yol açmayın. Python'daki girintileme sistemi konusuna biraz sonra tekrar döneceğiz. Ama şimdi bunu bir kenara bırakıp Python'da kodlar içine nasıl açıklama/yorum eklenir, biraz da bundan bahsedelim:

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 maksadıyla, 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...
```

Sizin daha mantıklı açıklamalar yazacağınızı ümit ederek konumuza geri dönüyoruz...

Şimdi yukarıda verdiğimiz yarım programı tamamlamaya çalışalım. Hemen boş bir metin 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 -*-

a = 23
b = 23
```

Yukarıda **a** ve **b** adında iki tane değişken tanımladık. Bu iki değişkenin de değeri **23**.

Programımızı yazmaya devam edelim:

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

a = 23
b = 23

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

Bu şekilde programımızı tamamlamış olduk. Bu programın pek önemli bir iş yaptığı söylenemez. Yaptığı tek şey, a ile b değişkenlerinin değerine bakıp, eğer bunlar birbirleriyle aynıysa ekrana "*a ile b birbirine eşittir*" 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."

ELSE

else deyimi kısaca, **if** deyimiyle 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!"

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 Pythonca'ya çevirdik:

"Eğer isim değişkeninin değeri Ferhat ise, ekrana 'Ne güzel bir isim bu!' cümlesini yazdır. Yok, eğer isim değişkeninin değeri Ferhat değil de 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!"
```

ELIF

Eğer bir durumun gerçekleşmesi birden fazla koşula bağlıysa **elif** deyiminden faydalanıyoruz. Mesela:

```
#!/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"

else:
 print meyve, "bir meyve değildir!"
```

Burada şu Türkçe ifadeyi Pythonca'ya 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. Eğer kullanıcının yazdığı isim bunlardan hiçbiri değilse ekrana meyve değişkeninin değeri ve 'bir meyve değildir' çıktısı yazılsın."

Eğer bir durumun gerçekleşmesi birden fazla koşula bağlıysa birden fazla **if** deyimini art arda da kullanabiliriz. Örneğin:

```
#!/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** deyimini de kullanarak 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:

```
sayi 100'dür
```

Gördüğünüz gibi programımızı **elif** deyimini kullanarak yazarsak Python belirtilen koşulu karşılayan ilk sonucu ekrana yazdıracak ve orada duracaktır.

Buraya kadar Python'da pek çok şey öğrenmiş olduk. **if**, **elif**, **else** deyimlerini de öğrendiğimize göre 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 (mesela; bu programı **`input()`** yerine **`raw_input()`** ile nasıl yazabileceğinizi düşünürseniz), verdiğimiz bu basit örnek amacına ulaşmış demektir.

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("Merhaba 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 görünümlerinin 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 genişliğini** 4 olarak ayarlamanız, **Sekme yerine boşluk ekle** seçeneğinin yanındaki kutucuğu da işaretli hale getirmeniz 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

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 bulunmaktadır. Bu makaleyi okumanızı tavsiye ederim.

PYTHON'DA DÖNGÜLER

3

Hatırlarsanız bir önceki bölümün sonlarına doğru 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** (*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...

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 küçücük örneği 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. Şimdi bu koda bazı eklemeler yapalım:

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

a = 0

while a < 100:
 a = 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 Python'caya 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`'nın 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`'nın 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.

Burada ilerleyebilmek için ihtiyacımız olacak bazı işlem yardımcılarına veya başka bir ifadeyle **işleçlere** (*operators*) değinelim:

Şimdiye kadar aslında bu işleçlerden birkaç tanesini gördük. Mesela:

+ işleci: Toplama işlemi yapmamızı sağlıyor.

- işleci: Çıkarma işlemi yapmamızı sağlıyor.

/ işleci: Bölme işlemi yapmamızı sağlıyor.

*** işleci:** Çarpma işlemi yapmamızı sağlıyor.

> işleci: "büyüktür" anlamına geliyor.

< işleci: "küçüktür" anlamına geliyor.

Bir de henüz görmediklerimiz, ama bilmemiz gerekenler var:

>= işleci: Büyük eşittir anlamına geliyor.

<= işleci: küçük eşittir anlamına geliyor.

!= işleci: Eşit değildir anlamına geliyor. (örn. "2 * 2 != 5")

and işleci: Ve anlamına geliyor.

or işleci: Veya anlamına geliyor.

not işleci: Değil anlamına geliyor.

true işleci: Doğru anlamına geliyor.

false işleci: Yanlış anlamına geliyor.

Bu işleçleri şu anda ezberlemenize gerek yok. Bunlar yalnızca size kılavuz olsun diye veriliyor. Yeri geldikçe bunları kullanacağımız için muhakkak aklınıza yerleşeceklerdir...

Şimdi konumuza geri dönebiliriz:

Bu konunun başında, bir önceki bölümde yazdığımız hesap makinesi programına değinmiştik. Şimdi bu programı görelim tekrar:

```
#!/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("Yapacağınız 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
```

Dediğimiz gibi, program bu haliyle her defasında yalnızca bir kez işlem yapmaya izin verecektir. Yani mesela toplama işlemi bittikten sonra program sona erecektir. Ama eğer biz bu programda şu ufacık değişikliği yaparsak işler değişir:

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

from __future__ import division

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

 soru = raw_input("Yapacağınız 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
```

Burada şu değişiklikleri yaptık:

İlk önce **`from __future__ import division`** satırı ile **`secenek1 = "(1) toplama"`** satırı arasına:

```
while True:
```

ifadesini ekledik. Bu sayede programımıza şu komutu vermiş olduk: *"Doğru olduğu müddetçe aşağıdaki komutları çalıştırmaya devam et!"* Zira yukarıda verdiğimiz işleç tablosundan da hatırlayacağınız gibi **`true`**, *"doğru"* anlamına geliyor.

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.

İkinci değişiklik ise **`while True:`** ifadesinin altında kalan bütün satırları bir seviye sağa kaydırmak oldu. Eğer Kwrite kullanıyorsanız, kaydıracağınız bölümü seçtikten sonra **CTRL+i** tuşlarına basarak bu kaydırma işlemini kolayca yapabilirsiniz. Bir seviye sola kaydırmak için ise **CTRL+SHIFT+i** tuşlarını kullanıyoruz.

Bir de şu örneğe 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.

Bu programı çalıştırdığımızda sorulan soruya "Python" cevabı vermezsek, program biz müdahale edene kadar ekrana "*Yanlış cevap!*" çıktısını vermeye devam edecektir. 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...

Aynı kodları bir de şu şekilde denerseniz **`if`** ile **`while`** arasındaki fark bariz bir biçimde ortaya çıkacaktır:

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

soru = raw_input("Python mu Ruby mi?")

if soru != "Python":
 print "Yanlış cevap!"
```

Şimdilik **`while`** döngüsüne ara verip bu konuda incelememiz gereken ikinci döngümüze geçiyoruz.

FOR DÖNGÜSÜ

Yukarıda while döngüsünü anlatırken yazdığımız şu kodu hatırlıyorsunuz:

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

a = 0

while a < 100:
 a = 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!"

`for` döngüsüyle ilgili şu örneğe de bir bakalım:

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

for kelimeler in "linux":
 print kelimeler
```

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.

RANGE() FONKSİYONU

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. Hatırlarsanız bu **`range()`** fonksiyonunu bir önceki bölümde örneklerimizden birinde de 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.

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"
```

LEN() FONKSİYONU

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 dökecektir.

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:

```
>>> sayi = 123456
>>> kardiz = str(sayi)
>>> len(kardiz)

6
```

123456 sayısını **str()** fonksiyonu yardımıyla bir karakter dizisine dönüştürdüğümüz için **len()** fonksiyonu görevini yerine getirecektir.

BREAK DEYİMİ

break deyimi, bir döngüyü 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 == kullanici_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 == kullanici_adi and soru2 == parola:
 print "Kullanıcı adı ve parolanız onaylandı."
```

Burada şu Türkçe ifadeyi Pythonca'ya çevirmiş olduk:

"Eğer soru1 değişkeninin değeri kullanici_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** işlecini yakından tanıyor olmalılar. **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*" derseniz, burada hem **a**'nın hem de **b**'nin doğru olması gerekir.

CONTİNUE DEYİMİ

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.

in işleci

for döngüsünü anlattığımız kısımda, bu döngünün sözdizimi içinde **`in`** adlı bir işleç dikkatinizi çekmiş olmalı:

```
>>> for i in falanca:
... ...
```

İşte bu bölümde bu işlecin tam olarak ne olduğundan ve ne işe yaradığından söz edeceğiz.

`in` kelimesi Türkçe'de '*içinde*' anlamına gelir. Python programlama dilindeki kullanımı da kelimenin bu anlamına oldukça yakındır. Eğer birkaç örnek verirsek, bu işlecin tam olarak ne işe yaradığını rahatlıkla anlayacaksınız:

```
>>> kardiz = 'istihza'
>>> 'i' in kardiz

True
```

Bu örnekte **`in`** işleci yardımıyla '*i*' adlı karakterin '*istihza*' adlı karakter dizisi içinde olup olmadığını sorguladık. Bu karakter '*istihza*' adlı karakter dizisi içinde geçtiği için de **`true`** cevabını aldık.

Bir de şuna bakalım:

```
>>> 'e' in kardiz
False
```

'*istihza*' adlı karakter dizisi içinde '*e*' karakteri olmadığı için bu defa da **false** çıktısı aldık.

Bu işlecin gerçek hayatta nasıl kullanılabileceğine dair bir örnek verelim:

```
# -*- coding: utf-8 -*-
cevap = raw_input("Sistemden çıkmak istediğinize emin misiniz? [E/H] ")
if 'E' in cevap or 'e' in cevap:
 print "Güle güle!"
else:
 print "Sistemden çıkılmadı!"
```

Bu kodlara göre, öncelikle kullanıcıya sistemden çıkmak isteyip istemediğini soruyoruz. Eğer kullanıcının verdiği cevap içinde '*E*' veya '*e*' karakterleri varsa kullanıcıyı sistemden çıkarıyoruz. Aksi halde kullanıcıyı sistemden çıkarmıyoruz.

Gördüğünüz gibi, bu kodlarda **or** ve **in** adlı işleçleri kullandık. **or** işleci kodlarımıza ya da anlamı verirken, **in** işleci **içinde** anlamı verdi. Dolayısıyla yukarıdaki kodları şu şekilde Türkçeleştirebiliriz:

```
Eğer cevap içinde 'E' varsa veya cevap içinde 'e' varsa:
    ekrana 'Güle güle!' ifadesini bas.
Aksi halde:
    ekrana 'Sistemden çıkılmadı!' ifadesini bas.
```

Bütün bu örneklerden de anlayacağınız gibi **in** işleci herhangi bir öğenin herhangi bir veri tipi içinde olup olmadığını sorgulamamızı sağlıyor. İlerleyen bölümlerde başka veri tiplerini de öğrendiğimizde bu işlecin ne kadar faydalı bir araç olduğunu siz de anlayacaksınız.

PYTHON'DA LİSTELER, DEMETLER VE SÖZLÜKLER

4

Bu bölümde Python'da dört yeni veri tipi daha öğreneceğiz. Öğreneceğimiz bu veri tipleri **liste** (*list*), **demet** (*tuple*), **sözlük** (*dictionary*) ve **sıralı sözlük** (*ordereddict*). Burada nasıl liste, demet, sözlük ve sıralı sözlük oluşturulacağını öğrenmenin yanısıra, bu veri tipleriyle neler yapabileceğimizi de görmeye çalışacağız.

Listeler

Dilerseniz listenin ne demek olduğunu tanımlamaya çalışmakla vakit kaybetmek yerine doğrudan konuya girelim. Böylece soyut kavramlarla kafa karıştırmadan ilerlememiz mümkün olabilir.

Listeleri kullanabilmek için yapacağımız ilk iş, listeyi tanımlamak olacak. Python'da herhangi bir liste oluşturmak için önce listemize bir ad vermemiz, ardından da köşeli parantezler içinde bu listenin öğelerini belirlememiz gerekiyor. Yani liste oluştururken dikkat etmemiz gereken iki temel nokta var. Birincisi tıpkı değişkenlere isim veriyormuşuz gibi listelerimize de isim vermemiz gerekiyor. Tabii listelerimizi isimlendirirken Türkçe karakterler kullanmayacağız. İkincisi, listemizi oluşturan öğeleri köşeli parantezler içinde yazacağız.

Şimdi hemen ilk listemizi tanımlayalım:

```
>>> liste = [“Hale”, “Jale”, “Lale”, 12, 23]
```

Daha önce de söylediğimiz gibi, burada dikkat etmemiz gereken nokta, liste öğelerini tanımlarken köşeli parantezler kullanıyor olmamız. Ayrıca liste içindeki karakter dizilerini her zamanki gibi tırnak içinde belirtmeyi unutmuyoruz. Tabii ki sayıları yazarken bu tırnak işaretlerini kullanmayacağız. Eğer sayılarda tırnak işareti kullanırsanız Python'un bu öğeleri nasıl algılayacağını biliyorsunuz. Bakalım bunları Python nasıl algılıyormuş?

Python komut satırında şu ifadeyi yazın:

```
>>> type("Hale")
```

Bu komutun çıktısı:

```
<type 'str'>
```

olacaktır. Yani "*Hale*" ifadesinin tipi **`str`** imiş. **`str`**, İngilizce'deki "*string*" kelimesinin kısaltması. Türkçe anlamı ise karakter dizisidir.

Şimdi aynı komutu şu şekilde deniyoruz:

```
>>> type(123)
```

Bu komut bize şu çıktıyı verecektir:

```
<type 'int'>
```

Demek ki 123 ifadesinin tipi **`int`** imiş. Bu **`int`** de İngilizce'deki "*integer*" kelimesinin kısaltması oluyor. Türkçe anlamı tamsayıdır.

Şimdi bu 123 ifadesini tırnak içinde yazalım:

```
>>> type("123")
<type 'str'>
```

Gördüğünüz gibi yazdığınız şey sayı da olsa, siz bunu tırnak içinde belirtirseniz, Python bunu sayı olarak algılamıyor.

Neyse biz konumuza dönelim.

Olması gerektiği şekilde listemizi tanımladık:

```
>>> liste = ["Hale", "Jale", "Lale", 12, 23]
```

Şimdi komut satırında:

```
>>> liste
```

yazdığımızda tanımladığımız "**liste**" adlı listenin öğeleri ekrana yazdırılacaktır.

Tanımladığımız bu listenin öğe sayısını, bir önceki bölümde öğrendiğimiz **len()** fonksiyonu yardımıyla elde edebiliriz:

```
len(liste)
5
```

Şimdi listeleri yönetmeyi; yani listeye öğe ekleme, listeden öğe çıkarma gibi işlemleri nasıl yapacağımızı öğreneceğiz. Bu işi Python'da bazı parçacıklar (ya da daha teknik bir dille söylemek gerekirse "metotlar"...) yardımıyla yapıyoruz.

İsterseniz gelin şimdi bu metotların neler olduğuna ve nasıl kullanıldıklarına bakalım.

APPEND

İlk metodumuz **append()**. Bu kelime Türkçe'de **eklemek**, **iliştirmek** anlamına geliyor.

Oluşturduğumuz listeye yeni bir öğe eklemek için **append()** metodundan faydalanıyoruz:

```
liste.append("Mehmet")
```

Dikkat edin, liste tanımlarken köşeli parantez kullanıyorduk. Listeleri yönetirken ise (yani metotları kullanarak ekleme, çıkarma vb. yaparken) normal parantezleri kullanıyoruz. Ayrıca gördüğünüz gibi, bu **append()** metodunu, liste isminin yanına koyduğumuz bir noktadan sonra yazıyoruz. **append()** metodu yardımıyla, oluşturduğumuz bir listenin en sonuna öğe ekleyebiliyoruz. Peki, bu metot yardımıyla birden fazla öğe ekleyebilir miyiz? Ne yazık ki, **append()** metodu bize listeye yalnızca tek bir öğe ekleme olanağı sunar.

Eğer biz ekleyeceğimiz bir öğeyi en sona değil de listenin belirli bir noktasına yerleştirmek istiyorsak, başka bir metottan faydalanıyoruz. Ama bu yeni metodu kullanmaya başlamadan önce Python'un liste öğelerini sıralama yönteminden bahsetmemiz gerekir. Python'un sıralama yöntemi ile ilgili olarak bilinmesi gereken en önemli kural şudur: Python, liste (ve öteki veri tipleri) içindeki öğeleri sıralarken, ilk öğeyi 0'dan başlatır.

Yani:

```
>>> liste = ["Hale", "Jale", "Lale", 12, 23, "Mehmet"]
```

biçiminde gördüğümüz listenin ilk öğesine "0'ıncı öğe" denir. Bu listedeki birinci öğe ise "Jale"dir.

Python'da bir listenin öğelerine erişmek için aşağıdaki yapıyı kullanıyoruz:

```
>>> liste[0]
```

Bu komutu yazdığımızda Python bize 0'ıncı öğenin "Hale" olduğunu söyleyecektir. Aynı şekilde;

```
>>> liste[2]
```

komutu ise bize 2. öğenin "*Lale*" olduğunu söyleyecektir. Ancak burada şuna dikkat etmemiz lazım: Python liste öğelerini numaralarken 0'dan başlasa da liste öğelerini sayarken 1'den başlar. Yani;

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

komutunu verdiğimizde elde edeceğimiz sayı 6 olacaktır. Çünkü listemizde 6 adet öğe bulunuyor.

Bu arada, bu sıralama yöntemi yalnızca listelere özgü değildir. Bu sistemi başka veri tipleri üzerine de uygulayabiliriz. Örneğin:

```
>>> kardiz = 'istihza.com'
>>> kardiz[0]
'i'
>>> kardiz[1]
's'
```

Python'un öğe sıralama mantığını öğrendiğimize göre, şimdi listenin en sonuna değil de kendi belirleyeceğimiz başka bir noktasına öğe eklememizi sağlayacak metodu görebiliriz.

INSERT

İşte bu **`insert()`** metodu yardımıyla listenin herhangi bir noktasına öğe ekleyebiliyoruz. Bu kelime Türkçe'de **yerleştirmek**, **sokmak** anlamına geliyor. **`insert`** metodu yardımıyla listenin 1. sırasına (Dikkat edin, 0'ıncı sıraya demiyoruz.) "*Ahmet*"i yerleştirebiliriz:

```
>>> liste.insert(1, "Ahmet")
```

Burada parantez içindeki ilk sayı, "*Ahmet*" öğesinin liste içinde yerleştirileceği sırayı gösteriyor. Listemizin son durumunu kontrol edelim:

```
>>> liste
```

Bu komutun çıktısı şöyle olur:

```
["Hale", "Ahmet", "Jale", "Lale", 12, 23, "Mehmet"]
```

Gördüğünüz gibi, **1** sayısı Python için **ilk** anlamına gelmiyor. Eğer listemizin en başına bir öğe eklemek istiyorsak şu komutu kullanacağız:

```
>>> liste.insert(0, "Veli")
```

Bu metot da tıpkı **`append()`** metodunda olduğu gibi listeye yalnızca bir adet öğe eklememize izin verir.

EXTEND

Bu kelime **genişletmek**, **uzatmak** anlamına geliyor. **`extend()`** metodu, oluşturduğumuz listeleri genişletmemizi veya uzatmamızı sağlar. Bu metodun işlevini anlatabilmenin en iyi yolu tabii ki örnekler üzerinde çalışmak. Şimdi yeni bir liste oluşturalım:

```
>>> yeni_liste = ["Simovic", "Prekazi", "Jardel", "Nouma"]
```

Şimdi de şu komutu verip ne elde ettiğimize bir bakalım:

```
>>> liste.extend(yeni_liste)
```

Gördüğünüz gibi, **`extend()`** metodu iki listenin öğelerini tek bir liste içinde birleştirmeye yarıyor. Ya da başka bir ifadeyle, bir listeyi genişletiyor, uzatıyor. **`extend()`** metoduyla yaptığımız işlemin aynısını **+** işlecini kullanarak şu şekilde de yapabiliriz:

```
>>> liste = liste + yeni_liste
```

Burada yaptığımız şey; **`liste`** ve **`yeni_liste`** adlı listelerin öğelerini bir araya toplayıp bütün öğeleri tekrar "*liste*" adlı listeye atamaktan ibarettir.

REMOVE

Liste oluşturmayı, **`append()`** ve **`insert()`** metotları yardımıyla listeye öğeler eklemeyi öğrendik. Peki, ya listemizden öğe çıkarmak istersek ne yapacağız? Python'da bu işi yapmamızı sağlayan iki tane metot var. Biz önce bunlardan ilki olan **`remove()`** metoduna bakacağız. Bu kelime Türkçe'de **çıkarmak**, **kaldırmak**, **silmek** anlamına geliyor.

Diyelim ki yukarıda tanımladığımız listeden "*Nouma*" öğesini çıkarmak/kaldırmak istiyoruz. O zaman şu komutu vermemiz gerekir:

```
>>> liste.remove("Nouma")
```

Eğer listede "*Nouma*" adlı birden fazla öğe varsa, Python listede bulduğu ilk "*Nouma*"yı çıkaracaktır.

POP

İngilizce'de pop kelimesi, **fırlamak**, **pırtlamak**, **aniden açılmak** gibi anlamlar taşıyor. Biz bu kelimeyi internette bir adrese tıkladığımızda aniden önümüze çıkan pop-up'lardan, yani açılır pencerelerden hatırlıyoruz. Python'da listeler ile birlikte kullandığımız **`pop()`** metodu ise listeden bir öğe silerken, bu sildiğimiz öğenin ekrana yazdırılmasını sağlıyor.

Şu komutu deneyelim:

```
>>> liste.pop()
```

Gördüğünüz gibi, Python bu **pop()** metodu yardımıyla listenin son öğesini çıkaracak, üstelik çıkardığı öğeyi ekrana yazdıracaktır. Eğer bu komutu şöyle verirsek ne olur?

```
>>> liste.pop(0)
```

Bu komut ise listedeki ilk, yani 0. öğeyi çıkarır ve çıkardığı öğeyi ekrana yazdırır. Anladığınız gibi **pop()** ile **remove()** arasındaki en temel fark; **pop()** metodunun silinen öğeyi ekrana yazdırması, **remove()** metodunun ise yazdırmamasıdır. Ayrıca **pop()** metodunda isim belirterek listeden silme işlemi yapamazsınız. Mutlaka silinecek öğenin liste içindeki sırasını vermelisiniz. **remove()** metodunda da bu durumun tam tersi söz konusudur. Yani **remove()** metodunda da sıra belirtemezsiniz; isim vermeniz gerekir...

Şimdiye kadar:

- bir listenin en sonuna nasıl öğe ekleyeceğimizi (**append()**),
- listenin herhangi bir yerine nasıl öğe ekleyeceğimizi (**insert()**),
- listeden isim vererek nasıl öğe çıkaracağımızı (**remove()**),
- listeden sayı vererek nasıl öğe çıkaracağımızı (**pop()**)

öğrendik.

Buraya kadar öğrendiğimiz metotlar listenin boyutunda değişiklikler yapmamızı sağlıyordu. Şimdi öğreneceğimiz metotlar ise listelerin boyutlarında herhangi bir değişiklik yapmıyor; yalnızca öğelerin yerlerini değiştiriyor veya bize liste hakkında ufak tefek bazı bilgiler veriyorlar.

İNDEX

Diyelim ki listedeki "*Jardel*" öğesinin listenin kaçıncı sırasında olduğunu merak ediyorsunuz. İşte bu **index()** metodu sizin aradığınız şey! Bunu şöyle kullanıyoruz:

```
>>> liste.index("Jardel")
```

Bu komut, "*Jardel*" öğesinin liste içinde kaçıncı sırada olduğunu gösterecektir bize...

SORT

Bazen listemizdeki öğeleri alfabe sırasına dizmek isteriz. İşte böyle bir durumda kullanacağımız metodun adı **`sort()`**:

```
>>> liste.sort()
```

REVERSE

Bu metot listedeki öğelerin sırasını ters yüz eder. Şöyle ki:

```
>>> liste.reverse()
```

Bu komutu üst üste iki kez verirseniz listeniz ilk haline dönecektir. Yani bu komut aslında **`sort()`** metodunun yaptığı gibi alfabe sırasını dikkate almaz. Listenizdeki öğelerin sırasını ters çevirmekle yetinir.

COUNT

Listelerle birlikte kullanabileceğimiz başka bir metot da budur. Görevi ise liste içinde bir öğenin kaç kez geçtiğini söylemektir:

```
>>> liste.count("Prekazi")
```

Buraya kadar listeleri nasıl yöneteceğimizi; yani:

- Nasıl liste oluşturacağımızı **`- - liste = []`**
- bu listeye nasıl yeni öğeler ekleyeceğimizi **`- - liste.append(), liste.insert()`**
- listemizi nasıl genişleteceğimizi **`- - liste.extend()`**
- eklediğimiz öğeleri nasıl çıkaracağımızı **`- - liste.remove(),liste.pop()`**
- liste içindeki öğelerin sırasını bulmayı **`- - liste.index()`**
- öğeleri abc sırasına dizmeyi **`- - liste.sort()`**
- öğelerin sırasını ters çevirmeyi **`- - liste.reverse()`**
- listedeki öğelerin liste içinde kaç kez geçtiğini bulmayı **`- - liste.count()`**

öğrendik...

Bunların yanısıra Python'un liste öğelerini kendi içinde sıralama mantığını da öğrendik. Buna göre unutmamamız gereken şey; Python'un liste öğelerini saymaya 0'dan başladığı. İsterseniz bu mantık üzerine bazı çalışmalar yapalım. Örneğin şunlara bir bakalım:

```
>>> liste[0]
```

Bu komut listenin ilk, yani 0. öğesini ekrana yazdıracaktır. Dikkat edin, yine köşeli parantez kullandık.

Peki, listedeki son öğeyi çağırmak istersek ne yapacağız? Eğer listemizde kaç tane öğe olduğunu bilmiyorsak ve **`len()`** komutuyla bunu öğrenmeyecek kadar tembelsek şu komutu kullanacağız:

```
>>> liste[-1]
```

Tabii ki siz **`len(liste)`** komutu verip önce listenin uzunluğunu da öğrenebilirsiniz. Buna göre, Python saymaya 0'dan başladığı için, çıkan sayının bir eksiği listenin son öğesinin sırasını verecektir. Yani eğer **`len(liste)`** komutunun çıktısı 5 ise, listedeki son öğeyi:

```
>>> liste[4]
```

... komutuyla da çağırabilirsiniz...

Olur ya, eğer kulağınızı tersten göstermek isterseniz **`len(liste)`** komutuyla bulduğunuz sayıyı eksiye dönüştürüp listenin ilk öğesini de çağırabilirsiniz. Yani, eğer **`len(liste)`** komutunun çıktısı 5 ise:

```
>>> liste[-5]
```

komutu size ilk öğeyi verecektir, tıpkı **`liste[0]`** komutunun yaptığı gibi...

Python bize bu mantık üzerinden başka olanaklar da tanıyor. Mesela tanımladığımız bir listedeki öğelerin tamamını değil de yalnızca 2. ve 3. öğeleri görmek istersek, şu komuttan faydalanıyoruz (saymaya 0'dan başlıyoruz):

```
>>> liste[2:4]
```

Gördüğünüz gibi, yukarıdaki komutta birinci sayı dâhil, ikinci sayı hariç olacak şekilde bu ikisi arasındaki öğeler listelenecektir. Yani **`liste[2:4]`** komutu listedeki 2. ve 3. öğeleri yazdıracaktır.

Eğer **:** işaretinden önce veya sonra herhangi bir sayı belirlemezseniz, Python varsayılan olarak oraya ilk veya son öğeyi koyacaktır:

```
>>> liste[:3]
```

komutu şu komutla aynıdır:

```
>>> liste[0:3]
```

Aynı şekilde;

```
>>> liste[0:]
```

komutu da şu komutla aynıdır (Listenin 5 öğeli olduğunu varsayarsak):

```
>>> liste[0:5]
```

Bu yöntemlerle listeye yeni öğe yerleştirmek, listeden öğe silmek vb. de mümkündür. Yani yukarıda metotlar yardımıyla yaptığımız işlemleri başka bir şekilde de yapabilmiş oluyoruz. Önce temiz bir liste oluşturalım:

```
>>> liste = ["elma", "armut", "kiraz", "karpuz", "kavun"]
```

Bu listenin en sonuna bir veya birden fazla öğe eklemek için (**`append()`** metoduna benzer şekilde...)

```
>>> liste[5:5] = ["domates", "salata"]
```

komutunu kullanıyoruz.

Hatırlarsanız, **`append()`** metoduyla listeye yalnızca bir adet öğe ekleyebiliyorduk. Yukarıdaki yöntem yardımıyla listeye birden fazla öğe de ekleyebiliyoruz.

Bu listenin 3. sırasına bir veya birden fazla öğe yerleştirmek için şu komutu kullanabiliriz (**`insert()`** metoduna benzer şekilde) .

```
>>> liste[3:3] = ["kebap", "lahmacun"]
```

Bu listenin 2. sırasındaki öğeyi silmek için ise şu komutu (**`remove()`** metoduna benzer şekilde...)

```
>>> liste[2:3] = []
```

Bu listenin 2. sırasındaki öğeyi silip yerine bir veya birden fazla öğe eklemek için:

```
>>> liste[2:3] = ["nane", "limon"]
```

Bu listenin 2. sırasındaki öğeyi silip yerine bir veya birden fazla öğeye sahip bir liste yerleştirmek için de şöyle bir şey yazıyoruz:

```
>>> liste[2] = ["ruj", "maskara", "rimel"]
```

Hangi işlemi yapmak için nasıl bir sayı dizilimi kullandığımıza dikkat edin. Bu komutlar başlangıçta biraz karışık gelebilir. Ama eğer yeterince örnek yaparsanız, bu komutları karıştırmadan uygulamayı öğrenebilirsiniz.

Artık listeler konusunu burada noktalayıp demetler (*tuples*) konusuna geçebiliriz...

DEMETLER

Demetler, listelere 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'>
```

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** (*unpacking*) deniyor.

Önce demetleyelim:

```
>>> aile = "Anne", "Baba", "Kardesler"
```

Şimdi demeti çözelim:

```
>>> a, b, c = aile
```

Bu şekilde komut satırına **a** yazarsak, "*Anne*" öğesi; **b** yazarsak "*Baba*" öğesi; **c** yazarsak "Kardesler" öğesi ekrana yazdırılacaktır. Demet çözme işleminde dikkat etmemiz gereken nokta; eşittir işaretinin sol tarafında demetteki öğe sayısı kadar değişken adı belirlememiz gerektiğidir.

"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.

SÖZLÜKLER

Sözlüğün ne demek olduğunu tanımlamadan önce gelin isterseniz işe bir örnekle başlayalım:

```
>>> sozluk = {"elma": "meyve", "domates": "sebze", 1: "sayi"}
```

Burada mesela, "*elma*" bir "*anahtar*", "*meyve*" ise bu anahtarın **değeridir**. Aynı şekilde "*sebze*" değerinin anahtarı "*domates*"tir. Dolayısıyla Python'da sözlük; **anahtar** ve **değer** arasında bağ kuran bir veri tipidir.

Mesela; bir adres veya telefon defteri yazmak istediğimizde bu sözlüklerden faydalanabiliriz. Yani "*sözlük*" denince aklımıza sadece bildiğimiz sözlükler gelmemeli. Şu örneğe bir bakalım:

```
>>> 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.

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 çok uzun değil ve 0 ile başlamıyorsa, bu sayıyı tırnaksız da yazabilirsiniz. Üçüncüsü iki nokta üst üste ve virgüllerin nerede, nasıl kullanıldığına da dikkat etmeliyiz. Şimdi gelelim sözlüklerle neler yapabileceğimize...

Şu komuta bir bakalım:

```
>>> telefon_defteri["Ahmet"]
```

veya:

```
>>> telefon_defteri["Salih"]
```

Bu komutlar "*Ahmet*" ve "*Salih*" adlı **anahtarların** karşısında hangi **değer** varsa, onu ekrana yazdıracaktır. Dikkat edin; sözlükten öğe 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...

Şimdi gelelim bu sözlükleri nasıl yöneteceğimize... Diyelim ki sözlüğümüze yeni bir öğe eklemek istiyoruz:

```
telefon_defteri["Zekiye"] = "0544 444 01 00"
```

Peki, sözlüğümüzdeki bir öğenin değerini değiştirmek istersek ne yapacağız?

```
telefon_defteri["Salih"] = "0555 555 55 55"
```

Buradan anladığımız şu: Bir sözlüğe yeni bir öğe eklerken de, var olan bir öğeyi değiştirirken de aynı komutu kullanıyoruz. Demek ki bir öğeyi değiştirirken aslında öğeyi değiştirmiyor, silip yerine yenisini koyuyoruz.

Eğer bir öğeyi listeden silmek istersek şu komutu kullanıyoruz:

```
del telefon_defteri["Salih"]
```

Eğer biz sözlükteki bütün öğeleri silmek istersek şu komut kullanılıyor:

```
telefon_defteri.clear()
```

Şu son örnekte gördüğümüz **clear()** ifadesi, Python sözlüklerinin metotlarından biridir. Sözlüklerin bunun dışında başka metotları da vardır. Bunlar içinde en önemlileri ise **keys()** ve **values()** adlı metotlardır. Kısaca söylemek gerekirse; **keys()** metodu bir sözlükteki anahtarları, **values()** metodu ise sözlükteki değerleri verir. Mesela:

```
>>> print telefon_defteri.keys()
['Ahmet', 'Salih', 'Selin']
>>> print telefon_defteri.values()
['0533 123 45 67', '0532 321 54 76', '0533 333 33 33']
```

Sözlüklerin ne olduğunu ve ne işe yaradığını öğrendiğimize göre, şimdi isterseniz, Python sözlüklerinin pratikliğini bir örnek yardımıyla görmeye çalışalım.

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 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 metotlarından biri olan **`get()`** metodunu seçiyoruz. Bu metot bize sözlük içinde bir değerin var olup olmadığını denetleme imkânının yanısıra, adı geçen değerin sözlük içinde var olmaması durumunda kullanıcıya gösterilecek bir mesaj seçme olanağı da sunar. Python sözlüklerinde bulunan bu **`get()`** metodu bizi bir else veya sonraki derslerimizde işleyeceğimiz try-except bloğu 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 olmadığını 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.

Burada ayrıca uzun karakter dizilerini nasıl böldüğümüze özellikle dikkat edin. Kodlarımızın sayfanın sağ tarafına doğru çirkin bir biçimde yayılmaması için \ işaretini kullanarak alt satıra geçiyoruz. Eğer bu işareti kullanmadan sadece **ENTER** tuşuna basarak alt satıra geçmeye çalışırsak kodlarımız hata verecektir. Ayrıca tıpkı daha önce komut satırında yaptığımız gibi, burada da sözlük öğelerinin sağa doğru uzamaması için öğeleri virgül işaretlerinden bölüp alt satıra geçtik.

Sıralı Sözlükler

Bu bölümde öğrendiğimiz **sözlük** adlı veri tipi **sırasız** bir veri tipidir. Peki, bu ne demek?

İsterseniz bunun ne demek olduğunu örnekler üzerinden anlatmaya çalışalım.

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}
```

Gördüğünüz gibi öğeler tanımladığımız sırada görünmüyor.

Ancak bazı durumlarda, bir sözlük yapısı içinde tanımladığınız öğelerin sırasını korumanız gerekebilir. Örneğin; şöyle bir personel kaydı oluşturduğumuzu düşünelim:

```
>>> personel = {'Ahmet': '19.01.2013', 'Mehmet': '21.03.2013',
...     'Selin': '30.06.2013'}
```

Burada, çalışanları işe giriş tarihlerine göre sıraladık. Ancak bu sözlüğü ekrana yazdırdığımızda bu sıra bozulacaktır:

```
>>> print personel
{'Selin': '30.06.2013', 'Ahmet': '19.01.2013', 'Mehmet': '21.03.2013'}
```

Gördüğünüz gibi, işe en son giren kişi Selin'di. Ama çıktıda bu kişi en başta görünüyor. Dediğimiz gibi, bu durumun sebebi sözlüklerin sırasız bir veri tipi olmasıdır.

İşte eğer herhangi bir şekilde sözlük yapısı içinde tanımladığınız öğelere, bu öğeleri tanımladığınız sıra ile erişmeniz gerekirse Python'daki başka bir araçtan yararlanacaksınız. Bu özel aracın adı **OrderedDict**.

OrderedDict, `Collections` adlı bir modül içinde yer alır. Biz henüz modül konusunu işlemedik. Ama Python programlama dilinde şimdiye kadar öğrendiklerimiz sayesinde, biraz sonra anlatacaklarımızı rahatlıkla anlayabilecek kadar Python bilgisine sahibiz. O halde hemen örneğimizi verelim…

Hatırlarsanız, sayılar konusundan bahsederken, iki tam sayı arasındaki bir bölme işleminin sonucunun küsuratsız olacağını, yani bu bölme işleminin sonucunun da bir tam sayı olacağını öğrenmiştik:

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

Normal şartlar altında 5 sayısını 2'ye böldüğümüzde 2.5 sayısını elde ederiz. Ama bölme işlemine giren 5 ve 2 sayıları birer tam sayı olduğu için, bölme işleminin sonucu da tam sayı oluyor. Eğer bölme işleminde ondalık kısımları da görmek istersek şöyle bir kod yazabiliyorduk:

```
>>> 5.0 / 2.0
2.5
```

Ya da şöyle bir yol takip edebiliyorduk:

```
>>> from __future__ import division
>>> 5/2
2.5
```

Gördüğünüz gibi, burada **`__future__`** adlı bir modül içindeki **`division`** adlı bir aracı kullandık. Aynı şekilde Python'daki sıralı sözlüklerden faydalanabilmek için de bir modül içindeki başka bir aracı kullanacağız. Dikkatlice bakın:

```
>>> from collections import OrderedDict
```

Dediğimiz gibi, **`OrderedDict`**, **`Collections`** adlı bir modül içinde yer alır. İşte biz de yukarıdaki satır yardımıyla bu modülün içindeki **`OrderedDict`** adlı aracı kodlarımızın içine aktardık. Böylece bu aracı kodlarımızda kullanabileceğiz.

Sıralı sözlükleri şöyle tanımlıyoruz:

```
>>> personel = OrderedDict([("Ahmet", "19.01.2013"),
... ("Mehmet", "21.03.2013"), ("Selin", "30.06.2013")])
```

Gördüğünüz gibi, sıralı sözlükler bir liste içindeki iki öğeli demetler şeklinde tanımlanıyor. Bu demetlerdeki ilk öğe sözlük anahtarlarını, ikinci öğe ise sözlük değerlerini temsil ediyor. Yalnız bu tanımlama şekli gözünüze biraz uğraştırıcı görünmüş olabilir. Eğer öyleyse, sıralı sözlükleri şu şekilde de tanımlayabilirsiniz:

Önce boş bir sıralı sözlük oluşturalım:

```
>>> personel = OrderedDict()
```

Şimdi bu sıralı sözlüğe öğeleri teker teker ekleyelim:

```
>>> personel["Ahmet"] = "19.01.2013"
>>> personel["Mehmet"] = "21.03.2013"
>>> personel["Selin"] = "30.06.2013"
```

Gördüğünüz gibi, bu biçim normal sözlüklere benziyor. Şimdi personeli ekrana yazdıralım:

```
>>> print personel
OrderedDict([('Ahmet', '19.01.2013'), ('Mehmet', '21.03.2013'),
('Selin', '30.06.2013')])
```

Bu çıktıya baktığımızda, öğelerin aynen bizim tanımladığımız sırada olduğunu görüyoruz.

Sıralı sözlükler, normal sözlüklerle aynı metotlara sahiptir. Dolayısıyla yukarıdaki sıralı sözlük üzerinde şu işlemleri yapabiliriz:

```
>>> personel.keys()
['Ahmet', 'Mehmet', 'Selin']
>>> personel.values()
['19.01.2013', '21.03.2013', '30.06.2013']
>>> print personel.get("Sedat", "Böyle biri yok!")
Böyle biri yok!
```

Sıralı sözlüklerin programcılık maceranız boyunca çok işinize yarayacağından emin olabilirsiniz. Hatta özellikle grafik arayüz geliştirirken bu yapının epey işinize yaradığını göreceksiniz.

Böylelikle Python'da Listeler, Demetler, Sözlükler ve Sıralı Sözlükler konusunu bitirmiş olduk. Bu konuyu sık sık tekrar etmek, hiç olmazsa arada sırada göz gezdirmek bazı şeylerin zihnimizde yer etmesi açısından oldukça önemlidir.

PYTHON'DA FONKSİYONLAR

5

Şimdi şöyle bir düşünün: Diyelim ki çalıştığınız işyerinde her gün bir yerlere bir 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.

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

print """Sayın Mehmet Bey,

19.12.2009 tariginde yaptığımız başvurunun sonuçlandırılması
hususunda yardımlarınızı rica ederiz.

Saygılarımızla,
Orçun Kunek"""
```

Bu dilekçe Mehmet Bey'e gidiyor. İsterseniz bir tane de Ahmet Bey'e yazalım...

```
print """Sayın Ahmet Bey,

15.01.2010 tariginde yaptığımız başvurunun sonuçlandırılması
hususunda yardımlarınızı rica ederiz.

Saygılarımızla,
Orçun Kunek"""
```

Burada dikkat ederseniz iki dilekçenin metin içeriği aslında aynıdır. 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 olabildiğince ayrıntılı bir şekilde incelemeye çalışacağız.

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 kullanmamanız gerekiyor. Ayrıca fonksiyonları tanımlarken en sona parantez ve iki nokta üst üste işaretlerini de koymayı unutmuyoruz.

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 dünya!"
```

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:

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

def fonksiyon_adi():
 print "merhaba dünya!"
```

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:

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

def fonksiyon_adi():
 print "merhaba dünya!"

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 çağrısı** (*function call*) adı verilir.

Yukarıdaki kodları çalıştırdığımızda ekrana "*merhaba dünya!*" 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:

```
# -*- 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.

Python'da fonksiyonlar bizi aynı şeyleri tekrar tekrar yazma zahmetinden kurtarır. Fonksiyonlar bir bakıma bir **taslaklama** sistemi gibidir. Biraz sonra vereceğimiz örneklerde bu durumu daha net olarak göreceğiz.

Python'da fonksiyonları kullanabilmek için öncelikle fonksiyonu tanımlamamız gerekir. Bir fonksiyonu tanımlamak için `def` adlı parçacıktan yararlanıyoruz.

Python'da bir fonksiyon tanımı şöyle bir yapıya sahiptir:

```
def fonksiyon_adi():
```

Fonksiyon adlarını belirlerken Türkçe karakter kullanmıyoruz. Fonksiyonlarımıza vereceğimiz adların olabildiğince betimleyici olması herkesin hayrınadır.

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. Bir fonksiyon, gövdedeki girintili kısmın bittiği yerde biter. Örneğin şu bir fonksiyondur:

```
def selamla():
 print "merhaba dünya!"
 print "nasılsın?"
```

Bu fonksiyon def **`selamla():`** satırıyla başlar, **`print "nasılsın?"`** satırıyla biter.

Fonksiyonların işlevli olabilmesi için bu fonksiyonlar tanımlandıktan sonra çağrılmalıdır. Örneğin yukarıdaki fonksiyonu şöyle çağırıyoruz:

```
def selamla():
 print "merhaba dünya!"
 print "nasılsın?"

selamla()
```

Dediğimiz gibi, bu fonksiyon, **`def selamla():`** satırıyla başlar, **`print "nasılsın?"`** satırıyla biter. Fonksiyon çağrısı 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 bir örnek daha yaparak, bu konuya ısınmanızı sağlayalım:

```
# -*- 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 vermek. **`cift()`** adlı fonksiyonun görevi ise ekrana "*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 sayi değişkeni de fonksiyon alanının dışında yer alır.

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 verdiğimizi düşünelim:

```
# -*- 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.

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

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:

```
# -*- 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:

```
def selamla(isim):
 print "merhaba, benim adım %s!" %isim

selamla("Ahmet Efendi")
```

Bu defa çıktımız şöyle:

```
merhaba, benim adım Ahmet Efendi!
```

Burada önemli olan nokta, **selamla()** adlı fonksiyonun 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 işe yaradığını 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 kavrarsanız 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 bulunur. Bu fonksiyonu şöyle kullanıyoruz:

```
>>> 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 çarparız? 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
```

Böylece listedeki bütün sayıların çarpımını gösteren değer **a** değişkenine atanmış oldu. İsterseniz bu **a** 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 = 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:

- Önce bir sayı listesi tanımlarsınız,
- Daha sonra, değeri 1 olan bir değişken tanımlarsınız,

- Son olarak da listedeki bütün sayıları bu değişkenin değeriyle çarpıp, elde ettiğiniz değeri tekrar bu değişkene atarsınız,
- 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:

```
def carp(liste):
 a = 1
 for i in liste:
  a = 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 çağırmak:

```
carp([3, 5, 7])
```

Burada dikkat ederseniz, **`carp()`** fonksiyonuna 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:

```
sayilar = [3, 5, 7]
carp(sayilar)
```

Şimdi kodları topluca görelim:

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

def carp(liste):
 a = 1
 for i in liste:
  a = a * i
 print(a)

sayilar = [3, 5, 7]
carp(sayilar)
```

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 değerin liste adını taşıma zorunluluğu yoktur. Mesela; bizim örneğimizde **`carp()`** fonksiyonunu **`sayilar`** adlı bir liste ile çağırdık...

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:

```
# -*- 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.

İsimli Argümanlar

Python'daki fonksiyonlara istediğiniz sayıda parametre verebilirsiniz. Mesela; şöyle bir fonksiyon tanımlayabilirsiniz:

```
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 **kayit** adlı bir sözlük oluşturduk. Bu sözlük, ekleyeceğimiz bağlantıya dair bilgileri tutacak. Daha sonra, oluşturduğumuz bu sözlüğe öğe 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 bulunur. Bu metot yardımıyla bir sözlük içinde bulunan bütün anahtar ve değer çiftlerini elde edebiliriz. Dilerseniz buna bir örnek verelim. 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 düzenleyici Kwrite
```

items() metodunun ne işe yaradığını gördüğümüze 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 **kayit** 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:

```
# -*- 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** devreye girer ve bizi büyük bir dertten kurtarır.

Nasıl mı? İsterseniz yukarıda verdiğimiz örnekten yararlanalım:

```
# -*- 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:

```
# -*- 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"* argümanıyla 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 kaydırdığımıza dikkat edin.

Yukarıdaki örneği, isimli argümanları kullanarak yazarsak sıralama meselesini dert etmemize gerek kalmaz:

```
# -*- 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 öğe sıralamasının önem taşıdığı argümanlara **sıralı argüman** adı verilir. Yabancılar buna **positional argument** diyor... Python fonksiyonlarında sıralı argümanlardan bolca yararlanılır. Ancak argüman sayısının çok fazla olduğu durumlarda isimli argümanları kullanmak da işinizi bir hayli kolaylaştırabilir.

Gömülü Fonksiyonlar (Built-in Functions)

Python'da en büyük nimetlerden biri de bu dilin yapısında bulunan **gömülü fonksiyonlar**'dır (*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()` 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 halihazırda 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. İstediğinizde 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.

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:

```
# -*- 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:

```
# -*- 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:

```
# -*- 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... Gördüğünüz 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:

```
# -*- 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:

```
# -*- 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 deyimi bir fonksiyon içindeki değerlere fonksiyon dışından da erişmemize yardımcı oluyor... Şimdi şöyle bir şey yazdığımızı düşünün:

```
# -*- 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 bir **`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:

```
# -*- 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.

RETURN DEYİMİ

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 döndürmekten 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 değerin 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:

Yukarıdaki fonksiyonu şu şekilde çağıralım:

```
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 değerin 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 döndürdüğümüz 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...

PASS DEYİMİ

`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/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.

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 verdiniz, ama fonksiyon içeriğini nasıl yazacağınızı düşünmediniz. Eğer program içinde sadece fonksiyonun ismini yazıp bırakırsanız programınız çalışma sırasında hata verecektir. İşte böyle bir durumda **pass** deyimi imdadınıza yetişir. Bu deyimi 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** deyimi yerleştirebilirsiniz. Daha sonra gövde kısmına ne yazacağınıza karar verdiğinizde bu **pass** deyimini silebilirsiniz. Böylece fonksiyonlar konusunu tamamlamış olduk. Artık yeni ve çok önemli bir konu olan "Modüller"e başlayabiliriz...

MODÜLLER

6

Bu bölümde Python'daki en önemli konulardan biri olan modüllerden söz edeceğiz. Ancak modülleri kullanabilmek için öncelikle modül denen şeyin ne olduğunu anlamamız gerekiyor. Hakikaten, nedir bu modül denen şey?

Diyelim ki bir program yazıyorsunuz. Yazdığınız bu programın içinde karakter dizileri, sayılar, değişkenler, listeler, demetler, sözlükler ve fonksiyonlar var. Programınız da **.py** uzantılı bir metin dosyası içinde yer alıyor. İşte bütün bu öğeleri ve veri tiplerini içeren **.py** uzantılı dosyaya **modül** adı verilir. Yani şimdiye kadar yazdığınız ve 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...

Modülleri İçe Aktarma (Importing Modules)

Demiştik ki, "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 bu işleme **import** deniyor... Peki, biz neyi içe aktaracağız?

Diyelim ki 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)
```

Bu program içinde `carp()` fonksiyonuna ihtiyacınız olduğunda yapmanız gereken tek şey bu fonksiyonu çağırmaktır. Bu işlemi nasıl yapacağımızı bir önceki bölümde tartışmıştık. Aynı fonksiyona başka bir programda da ihtiyaç duymamız halinde yine yapacağımız şey bu fonksiyonu o program içinden çağırmak olacak. Ancak bunu yapmanın belli kuralları var. Peki, nedir bu kurallar?

Öncelikle *"Modül nedir?"* sorusunu tekrar soralım. Bu bölümün en başında yaptığımız tanıma göre, **carp()** fonksiyonunu içeren yukarıdaki **deneme.py** adlı program bir modüldür. Bu modülün adı da **deneme**'dir. Python'da modüller .py uzantısına sahiptir. Ancak bir modülün adı söylenirken bu **.py** uzantısı es geçilir ve sadece isim kısmı dikkate alınır. Bu yüzden elinizdeki deneme.py adlı programın (ya da modülün) adı deneme olacaktır. Şimdi bu **deneme.py** adlı dosyanın bulunduğu dizin içinde bir komut satırı açıp Python'un etkileşimli kabuğunu çalıştırın. Mesela eğer deneme.py dosyasını masaüstüne kaydettiyseniz bir komut satırı açın, **cd Desktop** komutuyla masaüstüne gelin ve python komutunu vererek etkileşimli kabuğu başlatın. Şimdi şu komutu verin:

```
>>> import deneme
```

Eğer hiçbir şey olmadan bir alt satıra geçildiyse modülünüzü başarıyla içe aktardınız demektir. Eğer **No module named deneme** gibi bir hata mesajıyla karşılaşıyorsanız, muhtemelen Python'u masaüstünün olduğu dizinde başlatamamışsınızdır.

import deneme 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 görebiliriz:

```
>>> dir(deneme)
```

Bu komut bize şöyle bir çıktı verir:

```
['__builtins__', '__doc__', '__file__', '__name__',
'__package__', 'carp']
```

Burada bizi ilgilendiren kısım en sondaki **carp** adlı öğe. Bu çıktıdan anlıyoruz ki, **deneme** adlı modülün içinde **carp** adlı bir fonksiyon var ve biz bu fonksiyonu kullanma imkânına sahibiz. O halde gelin bu fonksiyonu kullanmak için sırasıyla şöyle bir şeyler yazalım:

```
>>> liste = [45, 66, 76, 12]
>>> deneme.carp(liste)
```

Bu komutlar şöyle bir çıktı verir:

```
2708640
```

Gördüğünüz gibi, deneme modülü içindeki **carp()** adlı fonksiyonu kullanarak **liste** içindeki sayıları birbiriyle çarptık. **deneme** modülünü nasıl içe aktardığımıza ve bu modülün içindeki bir fonksiyon olan **carp()** fonksiyonunu nasıl kullandığımıza çok dikkat edin. Önce modülümüzün adı olan deneme'yi 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 **deneme.py** adlı dosyanızı açıp dosyanın en sonuna şu kodu yazın:

```
a = 23
```

Yani **deneme.py** dosyasının son hali şöyle olsun:

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

def carp(liste):
 a = 1
 for i in liste:
  a = a * i
 print(a)

a = 23
```

Şimdi tekrar komut satırına dönüp şu komutu verin:

```
dir(deneme)
```

Bu komut biraz öncekiyle aynı çıktıyı verecektir. Halbuki biz modülümüze a adlı bir değişken daha ekledik. O halde neden bu değişken listede görünmüyor?

Python'da bir modülü 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(deneme)
```

Bu komut şöyle bir çıktı verir:

```
<module 'deneme' from 'deneme.py'>
```

Bu çıktı modülün başarıyla yeniden yüklendiğini gösteriyor. Şimdi **dir(deneme)** komutunu tekrar verelim:

```
>>> dir(deneme)
```

Bu defa listede **carp** öğesiyle birlikte **a** öğesini de göreceksiniz. Dolayısıyla artık bu öğeye de erişebilirsiniz:

```
>>> deneme.a
23
```

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 **deneme** modülünü örnek alalım:

```
>>> from deneme import *
>>> liste = [2, 3, 4]
>>> carp(liste)
24
```

Gördüğünüz gibi, bu defa **deneme.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 çağırmamızı sağlayacaktır. Yani bütün içeriği değil, bizim istediğimiz fonksiyonları içe aktarmakla yetinecektir. Örneğin:

```
>>> from deneme import carp
```

Bu şekilde **deneme** modülünün yalnızca **carp()** fonksiyonunu içe aktarmış olduk. Bu şekilde **carp()** fonksiyonuna erişebiliriz:

```
>>> liste = [2, 3, 4]
>>> carp(liste)
24
```

Ama **a** niteliğine erişemeyiz. Çünkü biz burada sadece **carp()** fonksiyonunu içe aktardık. Eğer **a** niteliğine de erişebilmek istersek modülümüzü şu şekilde içe aktarmamız gerekir:

```
>>> from deneme import carp, a
```

Bu şekilde hem **carp()** fonksiyonunu, hem de **a** 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 deneme as den
>>> liste = [2, 3, 4]
>>> den.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 **deneme** 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; **deneme.carp(liste)** gibi bir komuta baktığınızda **carp()** fonksiyonunun deneme 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, **deneme** adlı modülden **carp()** fonksiyonunu aldığınızda isim çakışması nedeniyle hiç istemediğiniz sonuçlarla karşılaşabilirsiniz... Buna bir örnek verelim.

Komut satırında şöyle bir kod yazın:

```
>>> a = 46
```

Şimdi **deneme** adlı modülü şu yöntemle içe aktarın:

```
>>> from deneme import *
```

Bakın bakalım **a** değişkeninin değeri ne olmuş?

```
>>> print a
23
```

Gördüğünüz gibi, **deneme** modülündeki **a** niteliği, sizin kendi programınızdaki **a** 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...

Elbette programcılık hayatınız boyunca sadece kendi yazdığınız modülleri kullanmayacaksınız. İnternet üzerinde, başkaları tarafından yazılmış binlerce modüle ulaşabilirsiniz. Hatta Python'un kendi içinde de Python geliştiricileri tarafından yazılmış onlarca faydalı modül bulunur. Zaten Python programlama diline asıl gücünü katan da bu ilave modüllerdir.

Bu noktaya kadar modüller hakkında genel bir bilgi edindiğimize göre, Python'un kendi içinde bulunan önemli modüllerden os modülünü inceleyerek modüller hakkında daha kapsamlı bir bilgi edinmeye başlayabiliriz:

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/python2.7/** dizini içinde bulunur. Windows sistemlerinde ise bu modülü bulmak için **C:/Python27/Lib** adlı dizinin içine bakabilirsiniz.

os modülü bize, kullanılan işletim sistemiyle ilgili işlemler yapma olanağı sunar. Modülün kendi belgelerinde belirtildiğine göre, bu modülü kullanan programların farklı işletim sistemleri üzerinde çalışma şansı daha fazla. Bunun neden böyle olduğunu biraz sonra daha iyi anlayacaksınız.

Bu modülü yukarıda anlattığımız şekilde içe 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 olabilirdiniz. 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 işlevi 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 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 alıyoruz. 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 NİTELİĞİ

`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:

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

import os

if os.name == "posix":
 a = raw_input("Linus Torvalds'a mesajınızı yazın:")
 print "Merhaba Linux kullanıcısı!"

if os.name == "nt":
 a = raw_input("Bill Gates'e mesajınızı yazın:")
 print "Merhaba Windows Kullanıcısı!"
```

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. **`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**"

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:")
 print "Merhaba Linux kullanıcısı!"

if name == "nt":
 a = raw_input("Bill Gates'e mesajınızı yazın:")
 print "Merhaba Windows Kullanıcısı!"
```

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.

LİSTDİR FONKSİYONU

`os` modülü içinde yer alan bu fonksiyon bize bir dizin içindeki dosyaları veya klasörleri listeleme imkânı veriyor. Bunu şöyle kullanıyoruz:

```
>>> import os
>>> a = os.listdir("/home/")
>>> print a
```

Yukarıdaki örnekte her zamanki gibi, modülümüzü **`import os`** komutuyla programımıza aktardık ilk önce. Ardından kullanım kolaylığı açısından **`os.listdir`** fonksiyonunu **`a`** adlı bir değişkene atadık. Örnekte **`os.listdir`** fonksiyonunun nasıl kullanıldığını görüyorsunuz. Örneğimizde **/home** dizini altındaki dosya ve

klasörleri listeliyoruz. Burada parantez içinde tırnak işaretlerini ve yatık çizgileri nasıl kullandığımıza dikkat edin. En son da **`print a`** komutuyla **/home** dizininin 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("/home/")

for dosyalar in a:
 print dosyalar
```

Eğer dosyalarımıza numara da vermek istersek şöyle bir şey yazabiliriz:

```
import os

a = os.listdir("/home/")
c = 0

for dosyalar in a:
 if c < len(a):
  c = c+1
  print c, dosyalar
```

Hatta Python'daki **`enumerate()`** adlı özel bir fonksiyonu kullanarak bu işlemi çok daha kısa bir yoldan halledebilirsiniz:

```
import os

a = os.listdir("/home/")

for s, d in enumerate(a, 1):
 print s, d
```

GETCWD FONKSİYONU

`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. Bunun için, kolaylık açısından, hemen Python komut satırını açalım ve **>>>** işaretinden hemen sonra şu komutu yazalım:

```
>>> import os
```

Bu komutu yazıp **ENTER** tuşuna bastıktan sonra da şu komutu verelim:

```
>>> 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 programımıza aktardık. Daha sonra **`mevcut_dizin`** adında bir değişken tanımlayıp **`os.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 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()
```

Buradaki **`pardir`** sabiti, İngilizce **parent directory** (*bir üst dizin*) ifadesinin kısaltması oluyor. **`pardir`** sabiti dışında, bir de **`curdir`** sabiti vardır. Bu sabiti kullanarak "mevcut dizin" üzerinde işlemler yapabiliriz:

```
>>> import os
>>> os.listdir(os.curdir)
```

Gördüğünüz gibi bu curdir sabiti **`getcwd()`** fonksiyonuna benziyor. Bunun dışında, istersek gitmek istediğimiz dizini kendimiz elle de belirtebiliriz:

```
>>> import os
>>> os.chdir("/var/tmp")
```

MKDIR() VE MAKEDIRS() FONKSİYONLARI

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?

```
Traceback (most recent call last):
File "deneme.py", line 2, in ?
os.mkdir("/home/istihza/Programlar/Python")
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")
```

Gördüğünüz gibi **/home/kullanıcı_adımız/** 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...

RMDIR() VE REMOVEDIRS() FONKSİYONLARI

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 irdeleyelim. 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:

```
Traceback (most recent call last):
File "deneme.py", line 4, in ?
os.makedirs(test1/test2)

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...

OSSEP NİTELİĞİ

Bu fonksiyon, 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 **os.sep** niteliği ne işe yarar? Yazdığımız kodlarda doğrudan dizin ayracı vermek yerine bu niteliği kullanırsak, programımızı farklı işletim sistemlerinde çalıştırırken, sistemin kendine özgü dizin ayracı'nın 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 **os.sep** niteliği, ilgili işletim sistemi hangisiyse ona ait olan dizin ayracının otomatik olarak yerleştirilmesini sağlayacaktır...

Bu **os.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 **os.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 **os.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 halledebiliyoruz. 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, **`os.sep`** niteliğinin, yazdığımız programların **taşınabilirliği** (*portability*), yani "farklı işletim sistemleri üzerinde çalışabilme kabiliyeti" açısından ne kadar önemli olabileceğini gösteriyor...

Bu bölümün başında şöyle bir soru sormuştuk: Acaba kendi yazdığımız modülleri içe aktarabilmek için modülün bulunduğu dizine gitmemiz gerekirken, os modülünü içe aktarmak için neden böyle bir şey yapmamıza gerek kalmıyor? Şimdi bu sorunun cevabını verelim:

Python'da **`sys`** adlı bir modül daha bulunur. Bu modülün **`path`** adlı bir de niteliği vardır. İsterseniz kendi gözlerimizle görelim:

```
>>> import sys
>>> sys.path
```

Bu komutlar bize dizin adlarını içeren uzun bir liste verir.

Biz herhangi bir modülü içe aktarmaya çalıştığımızda Python ilk olarak o anda içinde bulunduğumuz dizine bakar. Kendi yazdığımız modülü içe aktarabilmek için modülün bulunduğu dizine gitmemizin sebebi bu.

Eğer Python, modülü o anda çalışılan dizin içinde bulamazsa, `sys.path` çıktısında görünen dizinlerin içine bakar. Eğer aranan modül bu çıktıdaki dizinlerin herhangi birisinin içindeyse modülü bulup içe aktarabilir. Ama eğer modül bu dizinlerden birinde değilse içe aktarma sırasında hata verir.

Gördüğünüz gibi **/usr/lib/python2.7** dizini bu çıktıda görünüyor. O yüzden os modülünü rahatlıkla içe aktarabiliyoruz.

DOSYA İŞLEMLERİ

7

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. Burada Python'u kullanarak sistemimizde yeni dosyalar oluşturmanın yanı sıra, var olan dosyaları da, herhangi bir aracı program kullanmadan doğrudan Python ile açacağız.

Programcılık yaşamınız boyunca dosyalarla bol bol haşır neşir olacaksınız. O yüzden bu bölümü dikkatle takip etmenizi öneririm. İsterseniz lafı hiç uzatmadan konumuza geçelim.

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 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 kısaltmasıdır.

`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 bilgisayarınızda 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 var olan 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'
```

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 verdiğini aklımızın bir köşesine not edelim.

Eğer yazdığımız kod içinde yukarıdaki üç kipten hiçbirini kullanmazsak; Python, ön tanımlı 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.

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 bir 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.

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 --Orhan Veli--")

dosya.close()
```

Gördüğünüz gibi, "*şiir*" adlı bir metin dosyası oluşturup bu dosyaya yazma yetkisi verdik.

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()
```

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("şiir.txt","w")
>>> yeni.write("Sular çekilmeye başladı \
... köklerden...\nIsınmaz mı acaba ellerimde kan? \
... \nAh,ne olur! Bütün güneşler batmadan\nBi türkü \
... daha söyleyeyim bu yerde...")
>>> yeni.close()
>>> yeni = open("şiir.txt")
>>> print yeni.read()

Sular çekilmeye başladı köklerden...
Isınmaz mı acaba ellerimde kan?
Ah, ne olur! Bütün güneşler batmadan
Bi türkü daha söyleyeyim 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.

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 kurarak da dosyamızı okuyabiliriz:

```
>>> yeni = open("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 hiçbir 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 verirseniz; 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
```

Gördüğünüz 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.

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")
```

Dosyaya Rastgele Satır Eklemek

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 işlemini tanımlayalım:

```
>>> oku = kaynak.readlines()
```

Böylece **kaynak** dosya üzerinde yapacağımız satır okuma işlemini 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>
```

Biz bu dosyada **Item2** ile **Item3** arasına yeni bir satır eklemek istiyoruz. Dilerseniz bu işlemi nasıl yapacağımızı gösteren kodları verelim:

```
#!/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(3, eklenecek_satir+"\n")

oku.close()

yaz = open("falanca.xml", "w")
yaz.writelines(icerik)
yaz.close()
```

Şimdi de bu kodları tek tek inceleyelim.

```
oku = open("falanca.xml")
```

satırı yardımıyla **falanca.xml** adlı dosyayı okumak üzere açıyoruz.

Daha sonra dosyaya eklemek istediğimiz satırı bir değişkene atıyoruz.

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...

Bir sonraki satırda, dosyaya eklemek istediğimiz metni, **readlines()** metodu ile oluşturduğumuz listenin 4. sırasına yerleştiriyoruz. Burada listelerin **insert()** metodunu kullandığımıza dikkat edin.

Artık dosyayı okuma işlemi sona erdiği için dosyamızı **close()** metodunu kullanarak kapatıyoruz.

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")**).

Sonra yukarıda oluşturduğumuz içeriği yazmak üzere açtığımız dosyaya gönderiyoruz. Bunun için **writelines()** metodunu kullandık. Bildiğiniz gibi bu metot listeleri dosyaya yazdırmak için kullanılıyor.

Son olarak dosyayla işimizi bitirdiğimize göre, dosyamızı kapatmayı unutmuyoruz.

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 yine 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[2]
```

```
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 derecek 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.

HATA YAKALAMA 8

Hatalar programcılık deneyiminizin bir parçasıdır. Programcılık maceranız boyunca hatalarla sürekli karşılaşacaksınız. 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.

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 bakın:

```
#!/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. 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. Çünkü bir sayı 0'a bölünemez:

```
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.

pass Deyimi

pass kelimesi İngilizce'de **geçmek** anlamına gelir. Bu deyimin Python programlama dilindeki anlamı da buna çok yakındır. Bu deyimi Pyhon'da **görmezden gel**, **hiçbir şey yapma** anlamında kullanacağız. Mesela; bir hata ile karşılaşan programınızın hiçbir şey yapmadan yoluna devam etmesini isterseniz, bu deyimi 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 göründüğünden daha faydalı bir araç olduğunu anlayacaksınız.

KARAKTER DİZİLERİNİN METOTLARI

9

Bu bölümde, Python'daki karakter dizilerinin (*strings*) sahip oldukları metotlardan söz edeceğiz. Metotlar; Python'da bir karakter dizisinin, bir sayının, bir listenin veya sözlüğün niteliklerini kolaylıkla değiştirmemizi veya bu veri tiplerine yeni özellikler katmamızı sağlayan küçük parçacıklardır. Aslında bu metotları daha önceki derslerimizde de görmüştük.

Örneğin; listeler konusunu işlerken bu **metot** kavramıyla tanışmıştık. Örneğin; **`append`**, listelerin bir metodudur. Artık Python'da yeterince ilerleme sağladığımıza göre, daha önce kafa karıştırıcı olmaması için kullanmaktan kaçındığımız terimleri bundan sonra rahatlıkla kullanabilir, bunları hakiki şekilleriyle öğrenmeye girişebilir ve dolayısıyla Python'un terim havuzunda gönül rahatlığıyla kulaç atabiliriz.

Sözün özü; bu bölümde önceden de aşina olduğumuz bir kavramın, yani metotların, karakter dizileri üzerindeki yansımalarını izleyeceğiz. Önceki yazılarımızda işlediğimiz listeler ve sözlükler konusundan hatırlayacağınız gibi, Python'da metotlar genel olarak şu şablona sahip oluyorlar:

```
veritipi.metot
```

Dolayısıyla Python'da metotları gösterirken noktalı bir gösterme biçiminden söz ediyoruz. Daha önce sözünü ettiğimiz **`append()`** metodu da dikkat ederseniz, bu şablona uyuyordu.

Hemen bir örnek hatırlayalım:

```
>>> liste = ["elma", "armut", "karpuz"]
>>> liste.append("kebap")
>>> liste
["elma", "armut", "karpuz", "kebap"]
```

Gördüğünüz gibi noktalı gösterme biçimini uygulayarak kullandığımız **append()** metodu yardımıyla listemize yeni bir öğe ekledik. İşte bu bölümde, yukarıda kısaca değindiğimiz metotları karakter dizilerine uygulayacağız.

Kullanılabilir Metotları Listelemek

Dediğimiz gibi, bu yazıda karakter dizilerinin metotlarını inceleyeceğiz. Şu halde isterseniz gelin Python'un bize hangi metotları sunduğunu topluca görelim.

Mevcut metotları listelemek için birkaç farklı yöntemden faydalanabiliriz. Bunlardan ilki şöyle olabilir:

```
>>> dir(str)
```

Burada **dir()** fonksiyonuna parametre (argüman) olarak **str** adını geçiyoruz. Yeri gelmişken söyleyelim; eğer karakter dizileri yerine listelerin metotlarını listelemek isterseniz, kullanacağınız biçim şu olacaktır:

```
>>> dir(list)
```

Sözlüklerin metotlarını listelemek isteyen arkadaşlarımız ise, şu ifadeyi kullanacaktır:

```
>>> dir(dict)
```

Ama bizim şu anda ilgilendiğimiz konu karakter dizileri olduğu için derhal konumuza dönüyoruz. **dir(str)** fonksiyonu dışında, karakter dizilerini listelemek için şu yöntemden de yararlanabiliriz:

```
>>> dir("")
```

Açıkçası benim en çok tercih ettiğim yöntem de budur. Zira kullanılabilecek yöntemler içinde en pratik ve kolay olanı bana buymuş gibi geliyor. Burada

gördüğünüz gibi, **dir()** fonksiyonuna parametre olarak boş bir karakter dizisi veriyoruz. Biliyorsunuz, Python'da karakter dizileri tırnak işaretleri yardımıyla öteki veri tiplerinden ayrılıyor. Dolayısıyla içi boş dahi olsa, yan yana gelmiş iki adet tırnak işareti, bir karakter dizisi oluşturmak için geçerli ve yeterli şartı yerine getirmiş oluyor. İsterseniz bunu bir de **type()** fonksiyonunu kullanarak test edelim (Bu fonksiyonu önceki yazılarımızdan hatırlıyor olmalısınız):

```
>>> a = “”
>>> type(a)
<type ‘str’>
```

Demek ki gerçekten de bir karakter dizisi oluşturmuşuz. Şu halde emin adımlarla yolumuza devam edebiliriz.

Karakter dizisi metotlarını listelemek için kullanabileceğimiz bir başka yöntem de **dir()** fonksiyonu içine parametre olarak doğrudan bir karakter dizisi vermektir. Bu yöntem, öteki yöntemler içinde en makul yöntem olmasa da, en fazla kodlama gerektiren yöntem olması açısından parmak jimnastiği için iyi bir yöntem sayılabilir:

```
>>> dir(“herhangibirkelime”)
```

Dediğimiz gibi fonksiyon içinde doğrudan bir karakter dizisi vermenin bir anlamı yoktur. Ama Python yine de sizi kırmayacak ve öteki yöntemler yardımıyla da elde edebileceğiniz şu çıktıyı ekrana dökecektir:

```
[‘__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’, ‘format’,
‘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 var. 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 tek tek incelemeye… Yalnız öncelikle şunu söyleyelim: Bu bölümde __xxx__ şeklinde listelenmiş metotları incelemeyeceğiz. Karakter dizisi metotları dendiği zaman temel olarak anlaşılması gereken şey, **`dir("")`** fonksiyonu ile listelenen metotlar arasında __xxx__ şeklinde GÖSTERİLMEYEN metotlardır. __xxx__ şeklinde gösterilenler **özel metotlar** olarak adlandırılıyorlar ve bunların, bu yazının kapsamına girmeyen, farklı kullanım alanları var.

Bunu da söyledikten sonra artık asıl konumuza dönebiliriz.



Aşağıdaki metotları Türkçe karakter içeren karakter dizileri ile birlikte kullanmaya çalıştığınızda beklediğiniz sonuçları alamadığınızı görebilirsiniz. Bu metotların Türkçe karakterler ile birlikte nasıl kullanılması gerektiği bölüm sonunda açıklanacaktır.

CAPITALIZE METODU

Bu metot yardımıyla karakter dizilerinin ilk harflerini büyütmemiz mümkün oluyor. Örneğin:

```
>>> "adana".capitalize()

Adana
```

Ya da değişkenler yardımıyla:

```
>>> a = adana
>>> a.capitalize()

Adana
```

Yalnız dikkat etmemiz gereken bir nokta var: Bu metot yardımıyla birden fazla kelime içeren karakter dizilerinin sadece ilk kelimesinin ilk harfini büyütebiliyoruz:

```
>>> a = "parolanızı giriniz"
>>> a.capitalize()
```

Bir örnek daha verelim:

```
>>> a = ["elma","armut","kebap","salata"]
>>> for i in a:
...  print i.capitalize()
...
Elma
Armut
Kebap
Salata
```

UPPER METODU

Bu metot yardımıyla tamamı küçük harflerden oluşan bir karakter dizisinin bütün harflerini büyütebiliyoruz:

```
>>> "enflasyon".upper()

ENFLASYON
```

LOWER METODU

Bu metot **`upper()`** metodunun yaptığı işin tam tersini yapıyor. Yani büyük harflerden oluşan karakter dizilerini küçük harfli karakter dizilerine dönüştürüyor:

```
>>> a = "ARMUT"
>>> a.lower()

armut
```

SWAPCASE METODU

Bu metot da karakter dizilerindeki harflerin büyüklüğü/küçüklüğü ile ilgilidir. Metodumuz bize, bir karakter dizisinin o anda sahip olduğu harflerin büyüklük ve küçüklük özellikleri arasında geçiş yapma imkânı 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 = "kebap"
>>> a.swapcase()
KEBAP
>>> b = "KEBAP"
>>> b.swapcase()
kebap
>>> c = "KeBaP"
>>> c.swapcase()
kEbAp
```

TITLE METODU

Hatırlarsanız, yukarıda bahsettiğimiz metotlardan biri olan **`capitalize()`** bir karakter dizisinin yalnızca ilk harfini büyütüyordu. Bu **`title()`** metodu ise bir karakter dizisi içindeki bütün kelimelerin ilk harflerini büyütüyor:

```
>>> a = "python programlama dili"
>>> a.title()
Python Programlama Dili"
```

CENTER METODU

Bu metot, karakter dizilerinin sağında ve solunda, programcının belirlediği sayıda boşluk bırakarak karakter dizisini iki yana yaslar:

```
>>> "a".center(15)
'      a      '
```

İstersek boşluk yerine kendi belirlediğimiz bir karakteri de yerleştirebiliriz:

```
>>> "a".center(3,"#")
'#a#'
```

Gördüğünüz gibi, parantez içinde belirttiğimiz sayı bırakılacak boşluktan ziyade, bir karakter dizisinin ne kadar yer kaplayacağını gösteriyor. Yani yukarıdaki örneği göz önüne alırsak, asıl karakter dizisi **("a")+2** adet **#** işareti = 3 adet karakter dizisinin yerleştirildiğini görüyoruz. Eğer karakter dizimiz, tek harf yerine üç harften oluşsaydı, parantez içinde verdiğimiz üç sayısı hiç bir işe yaramayacaktı. Böyle bir durumda, **#** işaretini çıktıda gösterebilmek için parantez içinde en az 4 sayısını kullanmamız gerekirdi.

LJUST METODU

Bu metot, karakter dizilerinin sağında boşluk bırakarak, karakter dizisinin sola yaslanmasını sağlar:

```
>>> "a".ljust(15)
'a              '
```

Tıpkı **center()** metodunda olduğu gibi, bunun parametrelerini de istediğimiz gibi düzenleyebiliriz:

```
>>> "a".ljust(3,"#")
'a##'
```

RJUST METODU

Bu metot ise **ljust()**'un tersidir. Yani karakter dizilerini sağa yaslar:

```
>>> "a".rjust(3,"#")
'##a'
```

ZFİLL METODU

Yukarıda bahsettiğimiz **ljust()**, **rjust()** gibi metotlar yardımıyla karakter dizilerinin sağını-solunu istediğimiz karakterlerle doldurabiliyorduk. Bu **zfill()** metodu yardımıyla da bir karakter dizisinin soluna istediğimiz sayıda 0 yerleştirebiliyoruz:

```
>>> 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
```

REPLACE METODU

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
```

Gördüğünüz 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 **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...

STARTSWITH METODU

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”,”kebap”]
>>> for i in liste:
...  if i.startswith(“e”):
...   i.replace(“e”,”i”)
...
‘ilma’
‘irik’
‘iv’
‘ilbisi’
```

Sizin bu metodu kullanarak daha faydalı kodlar yazacağınıza inanıyorum...

ENDSWITH METODU

Bu metot, yukarıda anlattığımız **`startswith()`** metodunun yaptığı işin tam tersini yapıyor. Hatırlarsanız **`startswith()`** metodu ile bir karakter dizisinin hangi harfle başladığını denetliyorduk. İşte bu **`endswith()`** metoduyla da 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 = “damlaya damlaya göl olur...”
>>> kd4 = “el elden üstündür...”
```

```
>>> for i in kd1,kd2,kd3,kd4:
...  if i.endswith("!"):
...   print i.replace("!","")
...
ekmek elden su gölden
sakla samanı gelir zamanı
```

COUNT METODU

count() metodu bize bir karakter dizisi içinde bir karakterden kaç adet bulunduğunu denetleme imkânı 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...

ISALPHA METODU

Bu metot yardımıyla bir karakter dizisinin alfabetik olup olmadığını denetleyeceğiz. Peki, alfabetik ne demektir? Eğer bir karakter dizisi içinde yalnızca alfabe harfleri (a, b, c gibi...) varsa o karakter dizisi için alfabetik diyoruz. Bir örnekle bunu doğrulayalım:

```
>>> a = "kezban"
>>> a.isalpha()

True
```

Ama:

```
>>> b = "k3zb6n"
>>> b.isalpha()

False
```

İSDİGİT METODU

Bu metot da **`isalpha()`** metoduna benzer. Bunun yardımıyla bir karakter dizisinin sayısal olup olmadığını denetleyebiliriz. Sayılardan oluşan karakter dizilerine sayı karakter dizileri adı verilir. Örneğin; şu bir sayı karakter dizisidir:

```
>>> a = "12345"
```

Metodumuz yardımıyla bunu doğrulayabiliriz:

```
>>> a.isdigit()
True
```

Ama şu karakter dizisi sayısal değildir:

```
>>> b = "123445b"
```

Hemen kontrol edelim:

```
>>> b.isdigit()
False
```

ISALNUM METODU

Bu metot, bir karakter dizisinin **alfanümerik** olup olmadığını denetlememizi sağlar. Peki, alfanümerik nedir?

Daha önce bahsettiğimiz metotlardan hatırlayacaksınız:

Alfabetik karakter dizileri, alfabe harflerinden oluşan karakter dizileridir.

Sayısal karakter dizileri, sayılardan oluşan karakter dizileridir.

Alfanümerik karakter dizileri ise bunun birleşimidir. Yani sayı ve harflerden oluşan karakter dizilerine alfanümerik karakter dizileri adı verilir. Örneğin; şu karakter dizisi alfanümerik bir karakter dizisidir:

```
>>> a = "123abc"
```

İsterseniz hemen bu yeni metodumuz yardımıyla bunu doğrulayalım:

```
>>> a.isalnum()

True
```

Eğer denetleme sonucunda **`True`** alıyorsak, o karakter dizisi alfanümeriktir. Bir de şuna bakalım:

```
>>> b = "123abc>"
>>> b.isalnum()

False
```

`b` değişkeninin tuttuğu karakter dizisinde alfanümerik karakterlerin yanı sıra ("*123abc*"), alfanümerik olmayan bir karakter dizisi de bulunduğu için (**>**), **`b.isalnum()`** şeklinde gösterdiğimiz denetlemenin sonucu **`False`** (*yanlış*) olarak görünecektir.

Dolayısıyla, bir karakter dizisi içinde en az bir adet alfanümerik olmayan bir karakter dizisi bulunursa (bizim örneğimizde **<**), o karakter dizisi alfanümerik olmayacaktır.

ISLOWER METODU

Bu metot, bize bir karakter dizisinin tamamının küçük harflerden oluşup oluşmadığını denetleme imkânı sağlayacak. Mesela:

```
>>> kent = "istanbul"
>>> kent.islower()

True
```

Demek ki **`kent`** değişkeninin değeri olan karakter dizisi tamamen küçük harflerden oluşuyormuş.

Aşağıdaki örnekler ise **`False`** (*yanlış*) çıktısı verecektir:

```
>>> a = "Ankara"
>>> a.islower()

False
```

```
>>> b = "ADANA"
>>> b.islower()

False
```

ISUPPER METODU

Bu metot da **`islower()`** metoduna benzer bir şekilde, karakter dizilerinin tamamının büyük harflerden oluşup oluşmadığını denetlememizi sağlayacak:

```
>>> a = "ADANA"
>>> a.isupper()

True
```

ISTITLE METODU

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 kontrol etmemizi sağlayacak:

```
>>> a = "Karakter Dizisi"
>>> a.istitle()

True

>>> b = "karakter dizisi"
>>> b.istitle()

False
```

Gördüğünüz gibi, eğer karakter dizisinin ilk harfleri büyükse bu metot **`True`** çıktısı; aksi halde **`False`** çıktısı veriyor.

ISSPACE METODU

Bu metot ile bir karakter dizisinin tamamen **boşluk** karakterlerinden oluşup oluşmadığını kontrol ediyoruz. Eğer bir karakter dizisi tamamen boşluk karakterinden 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.

EXPANDTABS METODU

Bu metot yardımıyla bir karakter dizisi içindeki sekme boşluklarını genişletebiliyoruz. Örneğin:

```
>>> a = "elma\tbir\tmeyvedir"
>>> print a.expandtabs(10)

elma  bir  meyvedir
```

FIND METODU

Bu metot, bir karakterin, karakter dizisi içinde hangi konumda yer aldığını söylüyor bize:

```
>>> a = "armut"
>>> a.find("a")

0
```

Bu metot 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 bulduğu "*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. 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"
```

Bununla ilgili kendi kendinize bazı denemeler yaparak, işleyişi tam anlamıyla kavrayabilirsiniz.

RFIND METODU

Bu metot yukarıda anlattığımız find()metodu ile aynı işi yapar. Tek farklı karakter dizilerini sağdan sola doğru okumasıdır. Yukarıdaki **`find()`** metodu karakter dizilerini soldan sağa doğru okur. Mesela:

```
>>> a = "adana"
>>> a.find("a")
0
>>> a.rfind("a")
4
```

Gördüğünüz gibi, **`rfind()`** metodu karakter dizisini sağdan sola doğru okuduğu için öncelikle en sondaki "*a*" harfini döndürdü.

INDEX METODU

`index()` metodu 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 METODU

rindex() metodu da **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
```

JOIN METODU

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 yaptığını anladınız. "*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 metotlarınkinden biraz farklı. **`join()`** metodunda parantez içine doğrudan değişkenin kendisi yazdık. Yani **`a.join("*")`** gibi bir şey yazmıyoruz. Bu metot yardımıyla ayrıca listeleri de etkili bir biçimde karakter dizisine çevirebiliriz. Mesela; elimizde şöyle bir liste olsun:

```
>>> a = ["python", "php", "perl", "C++", "Java"]
```

Bu listenin öğelerini karakter dizileri halinde ve belli bir ölçüte göre sıralamak 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']
```

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: utf-8 -*-
```

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 1 harf 4 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: utf-8 -*-

import string

metin = "tbyksr çsn jücho elu gloglu"
kaynak= "defghijklmnoprstuvyzabc"
hedef = "abcdefghijklmnoprstuvyz"
cevir = string.maketrans(kaynak,hedef)
soncevir = metin.translate(cevir)
print soncevir
```

Bu programı komut satırından çalıştırdığınızda ne elde ettiniz?

PARTITION METODU

Bu metot yardımıyla bir karakter dizisini belli bir ölçüte göre üçe bölüyoruz. Örneğin:

```
>>> a = "istanbul"
>>> a.partition("an")

('ist', 'an', 'bul')
```

Eğer **`partition()`** metoduna parantez içinde verdiğimiz ölçüt karakter dizisi içinde bulunmuyorsa, şu sonuçla karşılaşırız:

```
>>> a = "istanbul"
>>> a.partition("h")

('istanbul', '', '')
```

RPARTITION METODU

Bu metot da **`partition()`** metodu ile aynı işi yapar, ama yöntemi biraz farklıdır. **`partition()`** metodu karakter dizilerini soldan sağa doğru okur. **`rpartition()`** metodu ise sağdan sola doğru. Peki, bu durumun ne gibi bir sonucu vardır? Hemen görelim:

```
>>> b = "istihza"
>>> b.partition("i")

('', 'i', 'stihza')
```

Gördüğünüz gibi, **`partition()`** metodu karakter dizisini ilk "*i*" harfinden böldü. Şimdi aynı işlemi **`rpartition()`** metodu ile yapalım:

```
>>> b.rpartition("i")

('ist', 'i', 'hza')
```

`rpartition()` metodu ise, karakter dizisini sağdan sola doğru okuduğu için ilk "*i*" harfinden değil, son "*i*" harfinden böldü karakter dizisini...

partition() ve **rpartition()** metotları, ölçütün karakter dizisi içinde bulunmadığı durumlarda da farklı tepkiler verir:

```
>>> b.partition("g")
('istihza', '', '')
>>> b.rpartition("g")
('', '', 'istihza')
```

Gördüğünüz gibi, **partition()** metodu boş karakter dizilerini sağa doğru yaslarken, **rpartition()** metodu sola doğru yasladı.

STRIP METODU

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 METODU

Bu metot 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 METODU

Bu metot bir karakter dizisinin sadece başında (solunda) yer alan boşluk ve yeni satır (**\n**) gibi karakterleri siler:

```
>>> a = "boşluk "
>>> a.lstrip()

'boşluk'

>>> b = "boşluk\n"
>>> b.lstrip()

'boşluk'
```

SPLITLINES METODU

Bu metot yardımıyla, bir karakter dizisini satır kesme noktalarından bölerek, bölünen öğeleri liste haline getirebiliyoruz:

```
>>> satir = "Birinci satır\nİkinci satır"
>>> print satir.splitlines()

["Birinci satır", 'İkinci satır']
```

SPLIT METODU

Bu metot biraz **join()** metodunun yaptığı işi tersine çevirmeye benzer. Hatırlarsanız **join()** metodu yardımıyla bir listenin öğelerini etkili bir şekilde 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']
```

Böylece her karakter dizisi farklı bir liste öğesi haline geldi:

```
>>> yeni[0]
'Debian'
>>> yeni[1]
'Pardus'
>>> yeni[2]
'Ubuntu'
>>> yeni[3]
'SuSe'
```

Bu metotta ayrıca isterseniz ölçütün yanı sıra, ikinci bir parametre daha kullanabilirsiniz:

```
>>> 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 öğesi "*Debian*", öteki öğesi de "*Pardus*, *Ubuntu*, *SuSe*" oldu. Bunu şu şekilde daha açık görebiliriz:

```
>>> c[0]
'Debian'
>>> c[1]
' Pardus, Ubuntu, SuSe'
```

Gördüğünüz gibi listenin 0. öğesi Debian'ken; listenin 1. öğesi "*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 kendine denemeler yapmanız, metodu daha iyi anlamanıza yardımcı olacaktır.

RSPLIT METODU

Bu metot, yukarıda anlattığımız **`split()`** metoduna çok benzer. Hatta tamamen aynı işi yapar. Tek bir farkla: **`split()`** metodu karakter dizilerini soldan sağa doğru okurken; **`rsplit()`** metodu sağdan sola doğru okur. Önce şöyle bir örnek verip bu iki metodun birbirine ne kadar benzediğini görelim:

```
>>> a = "www.python.quotaless.com"
>>> a.split(".")
['www', 'python', 'quotaless', 'com']
>>> a.rsplit(".")
['www', 'python', 'quotaless', 'com']
```

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'
```

Böylece Karakter Dizisi Metotları'nı bitirmiş olduk. Sıra geldi bu metotların Türkçe karakterler ile olan uyumsuzluklarını ve bu sorunu nasıl çözeceğimizi anlatmaya…

Metotlarda Türkçe Karakter Sorunu

Yukarıda anlattığımız bazı metotları kullanırken bir şey dikkatinizi çekmiş olmalı. Karakter dizilerinin bazı metotları, içinde Türkçe karakterler geçen karakter dizilerini dönüştürmede sorun çıkarabiliyor:

Mesela şu örneklere 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 bir **unicode karakter dizisi** olarak tanımladık. Gelin isterseniz bunu doğrulayalım:

```
>>> a = "şekerli çay"
>>> type(a)
<type 'str'>
```

Karakter dizisini normal bir şekilde tanımladığımızda **type(a)** sorgusu **<type 'str'>** değerini veriyor.

Bir de şuna bakalım:

```
>>> a = u"şekerli çay"
>>> type(a)
<type 'unicode'>
```

Karakter dizisinin dış tarafına bir adet '*u*' harfi eklediğimizde ise normal karakter dizilerinden farklı bir veri tipi olan unicode karakter dizisi elde etmiş oluyoruz.

Böylece **capitalize()** metodu bu karakter dizisinin ilk harfini doğru bir şekilde büyütebildi.



Unicode konusunu birkaç bölüm sonra ayrıntılı olarak ele alacağız.

Aynı sorun öteki metotlar için de geçerlidir:

```
>>> a = "şekerli çay"
>>> print a.upper() #"ŞEKERLİ ÇAY" vermeli.
şEKERLI çAY
>>> print a.title() #"Şekerli Çay" vermeli.
şEkerli çAy
>>> a = "şEkErLi çAy"
>>> print a.swapcase() #"ŞeKeRlİ ÇaY" vermeli.
şeKeRlI 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. Türkçedeki "*i*" harfi hiçbir dönüşümde "*İ*" sonucunu vermez... Örneğin:

```
>>> a = u"şekerli çay"
>>> print a.upper()
ŞEKERLI ÇAY
>>> a = "şEkErLi çAy"
>>> print a.swapcase()
ŞeKeRlI ÇaY
```

Gördüğünüz gibi "*i*" 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 metodunuzu icat etmeyi deneyebilirsiniz. Mesela; şöyle bir şey yazabilirsiniz:

```
# -*- coding: utf-8 -*-
donusturme_tablosu = {u'i': u'İ',
                      u'İ': u'i',
                      u'ı': u'I',
                      u'I': u'ı'}
def duzelt(kardiz):
 s = ''
 for i in kardiz:
  s += donusturme_tablosu.get(i, i)
 return s
```

Burada öncelikle **donusturme_tablosu** adlı bir sözlük tanımladık. Bu sözlükte Türkçeye özgü karakterlerin doğru şekilde büyütülmüş ve küçültülmüş hallerini tutuyoruz.

Ardından da **duzelt()** adlı bir fonksiyon tanımladık. Bu fonksiyonda, kardiz parametresi içinde geçen her bir harfi dönüştürme tablosu içinde tek tek arıyoruz. Eşleşen karakterleri dönüştürerek, eşleşmeyen karakterleri ise olduğu gibi s adlı karakter dizisine gönderiyoruz. Bu arada şu satır size biraz farklı görünmüş olabilir:

```
s += donusturme_tablosu.get(i, i)
```

Bu satır şununla aynı anlama gelir:

```
s = s + donusturme_tablosu.get(i, i)
```

Burada **get()** metodunu kullanarak, kardiz içindeki her bir karakteri **donusturme_tablosu** içinde arıyoruz. Eğer kardiz içinde mesela '*i*' harfi varsa, bunu sözlükteki karşılığı olan '*İ*' harfine dönüştürüp **s** adlı değişkene gönderiyoruz. Örneğin; '*kitap*' gibi bir kelime, *s* değişkenine şu şekilde gönderilecektir:

```
kİtap
```

Böylece '*i*' harfini büyütme işini Python'a bırakmayıp, kendimiz halletmiş oluyoruz.

Bu fonksiyonu şöyle kullanıyoruz:

```
print duzelt(u"şekerli çay").upper()
print duzelt(u"şekerli çay").capitalize()
print duzelt(u"şekerli çay").title()
print duzelt(u"SICAK ÇAY").lower()
```

Gördüğünüz gibi karakter dizisini önce **duzelt()** fonksiyonuna gönderip içindeki Türkçe karakterleri düzgün bir şekilde dönüştürüyoruz, ardından da bu düzeltilmiş karakter dizisini ilgili karakter dizisi metoduna gönderiyoruz.

Yukarıdaki metot **swapcase()** hariç bütün karakter dizisi metotlarındaki Türkçe problemini çözer.

swapcase() metodu için ise şöyle bir fonksiyon tanımlayabiliriz:

```
def tr_swapcase(kardiz):
 s = ''
 for i in kardiz:
  if i.isupper():
   s += duzelt(i).lower()
  else:
   s += duzelt(i).upper()
 return s
```

Gördüğünüz gibi **tr_swapcase()** metodunda da biraz önce tanımladığımız **duzelt()** fonksiyonundan yararlandık. **tr_swapcase()** metodunu şöyle kullanıyoruz:

```
print tr_swapcase(u"şEkErLi çAy")
```

Böylece Python'daki hemen hemen bütün karakter dizisi metotlarını inceledik. Ama dikkat ederseniz metot listesi içindeki üç metodu anlatmadık. Bunlar; **encode**, **decode** ve **format** metotları... Bunları daha sonraki konularda 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.

DÜZENLİ İFADELER (REGULAR EXPRESSIONS)

10

Düzenli ifadeler, Python Programlama Dili'ndeki 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 sürecek 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 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 sorunla 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.

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.

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 bulduğunu söylemiyor. Bize söylediği tek şey, aradığımız şeyi bulduğu... Bunu söyleme tarzı da yukarıdaki gibi... Yukarıdaki çıktıda gördüğümüz ifadeye Python'cada **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. 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 öbür 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.

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 görecek 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
programlama
```

Gördüğünüz gibi *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 bilgimiz 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...

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 kurarak **`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.

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 işin 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 ifadelere kişiliğini kazandıran şeydir. Biraz sonra ne demek istediğimizi daha açık anlayacaksınız. Artık gerçek anlamıyla düzenli ifadelere giriş yapıyoruz!

Öncelikle, elimizde aşağıdaki gibi bir liste olduğunu varsayalım:

```
>>> liste = ["özcan", "mehmet", "süleyman", "selim",
... "kemal", "özkan", "esra", "dündar", "esin",
... "esma", "özhan", "özlem"]
```

Diyelim 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 alırız. 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", "dündar",
... "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 `[]` metakarakterinin 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 ayıkladık. 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.

[ ] (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 tanımaktır. 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.

. (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 öğe 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.

* (Yıldız)

Bu metakarakter, 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 öğesi olan "*falanca*"da 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*" öğesi bu koşulları karşılamadığı için süzgecimizin dışında kaldı.

Burada dikkat edeceğimiz nokta, ***** metakarakterinin 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 `[]` metakarakteri yardımıyla "*sa*" harflerini nasıl grupladığımıza dikkat edin...

Şimdi "**.**" metakarakterini 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 metakarakteri incelemeye başlayabiliriz.

+ (Artı)

Bu metakarakter, bir önceki metakarakterimiz olan * ile benzerdir. Hatırlarsanız, * metakarakteri kendisinden önceki sıfır veya daha fazla sayıda tekrar eden karakterleri ayıklıyordu. + metakarakteri 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 öğe de bu düzenli ifadenin kapsamına girecektir. Mesela; listemiz şöyle olsa idi:

```
>>> liste = ["ahmet","mehmet","met","kezban"]
```

Yukarıdaki düzenli ifade bu listedeki "*met*" öğesini de içine alacaktı. Çünkü * adlı metakarakter 0 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' öğesini 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*" öğesini almaktı. Dikkat ederseniz, öğenin 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 y azmak 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 metakarakteri incelemeye başlayabiliriz.

? (Soru İşareti)

Hatırlarsanız, ***** karakteri **0** ya da daha fazla sayıda eşleşmeleri; **+** ise bir ya da daha fazla sayıda eşleşmeleri kapsıyordu. İşte şimdi göreceğimiz **?** sembolü de eşleşme sayısının **0** veya **1** 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
```

Gördüğünüz gibi **?** adlı metakarakterimiz, kendisinden önce gelen karakterin hiç bulunmadığı (yani 0 sayıda olduğu) ve 1 adet bulunduğu durumları içine alıyor. Bu yüzden de çıktı olarak bize sadece "*st*" ve "*sat*" öğelerini veriyor.

Şimdi bu metakarakteri 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 ilişkilerin hukuksal boyutunu bilimsel bir
disiplin içinde inceler. Devletlerarası hukuk da denir. Ancak uluslar
arası ilişkilere yeni aktörlerin girişi bu dalı sadece devletlerarası
olmaktan çıkarmıştır."""
```



Bu metin *http://tr.wikipedia.org/wiki/Uluslararas%C4%B1_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) 0 veya 1 kez geçtiği durumları arıyor. Burada `?` sembolünü `" "` karakterinden, yani boşluk karakterinden sonra kullandık. Dolayısıyla **boşluk karakterinin 0 veya 1 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.

{ } (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 0 kez, en çok da 3 kez tekrar etmesini istediğimiz söyledik. Dolayısıyla, "*a*" harfinin 0, 1, 2 ve 3 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...

^ (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
```

Gördüğünüz gibi **^** 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 kırpılmış 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:

- Aradığımız öğe bir sayı veya büyük harf ile başlamalı.
- En baştaki sayı veya büyük harften sonra küçük harf gelmemeli (Bu ölçütü **^** işareti sağlıyor).
- 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 öğe "*1agAY54*" olacaktır. Dolayısıyla bu öğe 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.

$ (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 10 öğelik 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 "*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.

\ (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, bunların taşıdıkları özel anlam yüzünden bu sembolleri ayıklamak hemencecik mümkün olmayacaktır. Yani 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", "25£"]
```

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...

| (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...

```
>>> for i in renkler:
...  if re.search("kırmızı|mavi|sarı",i):
...   print i
```

Sırada son metakarakterimiz olan () var...

( ) (Parantez)

Bu metakarakter yardımıyla düzenli ifade kalıplarını gruplayacağız. Bu metakarakter bizim bir karakter dizisinin istediğimiz kısımlarını ayıklamamızda çok büyük kolaylıklar sağlayacak.

Diyelim ki biz *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('href=".+html">.+</a>',i)
 if nesne:
  print nesne.group()
```

Burada yaptığımız şey şu:

1. Öncelikle *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: **<href=".+html">.+** Çünkü bahsi geçen web sayfasındaki html uzantılı dosyalar bu şekilde gösteriliyor. Bu durumu, web tarayıcınızda *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 düşünürsek, yazdığımız şey şu anlama geliyor: "*Bana 'html' ile biten bütün karakter dizilerini bul!*"
5. *http://www.istihza.com/py2/icindekiler_python.html* adresinin kaynağına baktığımız zaman aradığımız bilgilerin hep şu şekilde olduğunu görüyoruz: **`href="temel_bilgiler.html">1. 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:

```
href="http://www.istihza.com/py3/icindekiler_python.html">şuradaki</a>
href="temel_bilgiler.html">1. Temel Bilgiler</a>
...
```

   Hemen hemen amacımıza ulaştık sayılır. Ama gördüğünüz gibi çıktımız biraz karmaşık. Bunları istediğimiz gibi düzenleyebilsek 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('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('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* sitesinde herhangi bir değişiklik olursa yukarıdaki kodların istediğiniz çıktıyı vermeyeceğini bilmelisiniz. Çünkü yazdığımız düzenli ifade istihza.com sitesinin sayfa yapısıyla sıkı sıkıya bağlantılıdır.

Eşleşme Nesnelerinin Metotları

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 programlama 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 öğesi 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.

GROUPS() METODU

Bu metot, 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('href="(.+html)">(.+)</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('href="(.+html)">(.+)</a>',i)
 if nesne:
  print nesne.groups()
```

Gördüğünüz gibi şuna benzer çıktılar elde ediyoruz:

```
('temel_bilgiler.html', '1. Temel Bilgiler')
('pythona-giris.html', '2. Python'a Giriş')
('python-programlarini-kaydetmek.html', '3. Python Programlarını Kaydetmek')
('kullaniciyla_iletisim.html', '4. Kullanıcıyla İletişim: Veri Alışverişi')
...
...
...
```

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.

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 gösterdiğimize 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 anlama 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-Za-z0-9_]** ile eşdeğerdir.

\D ve **\W** dizilerinin yeterince anlaşılır olduğunu zannediyorum. Burada sanırım sadece S dizisi bir örnekle somutlaştırılmayı hak ediyor:

```
>>> 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 öğeyi 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 yığın var: *http://www.istihza.com/denemeler/yigin.txt*

Yapmanız gereken, bu yığın içindeki gizli mesajı düzenli ifadeleri kullanarak bulmak... Cevaplarınızı *kistihza@yahoo. com* adresine gönderebilirsiniz.

Düzenli İfadelerin Derlenmesi

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 yapmasa da, özellikle büyük çaplı programlarda bu metodu kullanmak oldukça faydalı olacaktır.

Basit bir örnekle başlayalım:

```
>>> liste = [“Python2.5”, “Python2.6”, “Python2.7”,
... “Python3.3”, “Java”]
>>>derli = re.compile(“[A-Za-z]+[0-9]\.[0-9]”)

>>> for i in liste:
...  nesne = derli.search(i)
...  if nesne:
...   print nesne.group()
...
Python2.5
Python2.6
Python2.7
Python3.3
```

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: utf-8 -*-

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 dolar'ın 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.

compile() İle 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.

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: utf-8 -*-

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)
```

Bu programı çalıştırdığımızda şu çıktıyı alıyoruz:

```
['Programlama', 'Programlama', 'programlama', 'programlama']
```



Bu metin *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...

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, 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 kazandıracağız:

```
>>> derle = re.compile("Python.*",re.DOTALL)
>>> nesne = derle.search(a)
>>> if nesne:
...  print nesne.group()
```

re.DOTALL seçeneğini sadece **re.S** şeklinde de kısaltabilirsiniz...

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", "kebap"]
>>> for i in liste:
...  nesne=re.search("\w*",i)
...  if nesne:
...   print nesne.group()
...
f
kebap
```

Burada alfanümerik karakterler içeren öğeleri ayıklamaya çalıştık. Normalde çıktımız "*çilek fıstık kebap*" 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...

Dikkatlice bakın:

```
# -*- coding: utf-8 -*-
import re
liste = ["çilek","fıstık", "kebap"]
derle = re.compile("\w*", re.UNICODE)
for i in liste:
 nesne = derle.search(i)
 if nesne:
  print nesne.group()
```

Düzenli İfadelerle Metin/Karakter Dizisi Değiştirme İşlemleri

sub() metodu

Şimdiye kadar hep düzenli ifadeler yoluyla bir karakter dizisini nasıl eşleştireceğimizi inceledik. Ama tabii ki düzenli ifadeler yalnızca bir karakter dizisi **bulmakla** 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."""



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 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 kopyalayarak 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 verirseniz **`group()`** metodu yardımıyla ne bulduğumuzu görebilirsiniz:

```
çileğ
çilek
çileğ
çilek
Çileğ
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.

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]
```

SONUÇ

Böylelikle düzenli ifadeler konusunu bitirmiş olduk. Buradaki amacımız, size düzenli ifadeler konusunda genel bir bakış sunabilmekti. 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 TABANLI PROGRAMLAMA OOP (NTP)

11

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.

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 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 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'ye bulaştırmanız 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, güzel cümlelerle bize anlatılacak faydaları özümseyebilelim. NTP'yi öğrenmeye çalışan birine birkaç sayfa boyunca "NTP şöyle iyidir, NTP böyle hoştur," demenin pek yararı olmayacaktır. Çünkü böyle bir çaba, konuyu anlatan kişiyi ister istemez okurun henüz bilmediği kavramları kullanarak bazı şeyleri açıklamaya çalışmaya itecek, bu da okurun zihninde birtakım bulanık 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.

Siniflar

NTP'de en önemli kavram sınıflardır. 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.

Sinif Tanimlamak

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 IlkSinif:
 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.

Sınıfları Çalıştırmak

Şimdi güzel güzel yazdığımız bu sınıfı nasıl çalıştıracağımıza 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 **konsole** (*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 Settings/İsminiz*** olacaktır. Orada şu komutu vererek masaüstüne geçiyoruz:

```
cd Desktop
```

Şimdi de şu komutu vererek Python komut satırını başlatıyoruz:

```
python
```

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: Örnekleme

Örnekleme (Instantiation)

Şimdi şöyle bir şey yazıyoruz:

```
deneme = IlkSinif()
```

Böylece oluşturduğumuz sınıfı bir değişkene atadık. NTP kavramlarıyla konuşacak olursak, sınıfımızı örneklemiş olduk.

Peki, bu örnekleme denen şey de ne oluyor? Hemen bakalım:

İngilizce'de **instantiation** olarak ifade edilen **örnekleme** kavramı sayesinde sınıfımızı kullanırken belli bir kolaylık sağlamış oluyoruz. Gördüğünüz gibi, **örnekleme** (*instantiation*) aslında şekil olarak yalnızca bir değişken atama işleminden ibarettir. Nasıl daha önce gördüğümüz değişkenler uzun ifadeleri kısaca adlandırmamızı sağlıyorsa, burada da örnekleme işlemi hemen hemen aynı vazifeyi görüyor. Yani böylece ilk satırda tanımladığımız sınıfa daha kullanışlı bir isim vermiş oluyoruz. Dediğimiz gibi, bu işleme **örnekleme** (*instantiation*) adı veriliyor. Bu örneklemelerin her birine ise **örnek** (*instance*) deniyor. Yani IlkSinif adlı sınıfa bir isim verme işlemine örnekleme denirken, bu işlem sonucu ortaya çıkan değişkene de, örnek diyoruz. Buna göre, burada deneme adlı değişken, **`IlkSinif`** adlı sınıfın bir örneğidir. Daha soyut bir ifadeyle, örnekleme işlemi **`Class`** (*sınıf*) nesnesini etkinleştirmeye yarar. Yani sınıfın bütününü 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.istihza.com adresi içindeki bütün bilgileri bir sınıf olarak düşünürsek, "*www.istihza.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 oluyorsunuz... Buna göre **`İnsan`** sınıfı insanın ne tür özellikleri olduğuna dair tanımlar (fonksiyonlar, veriler) içeriyor. Mehmet örneği (*instance*) ise bu tanımları, nitelikleri, özellikleri taşıyan bir nesne oluyor...

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 sürece tabi tutulacaktır. Burada bu sürecin ayrıntılarına girmeyeceğiz. Ama kısaca şöyle anlatabiliriz: Python'da (ve birçok 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
```

Niteliklere Değinme (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, örneğin:

```
>>> deneme.mesele
```

dediğimiz zaman, buradaki mesele; **`deneme`** adlı sınıf **örneğinin** (*instance*) bir

niteliği (*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 değişken oluşturduk. **c** değişkeni **a** ve **b** değişkenlerinin toplamıdır.

Bu sınıftaki **a**, **b** ve **c** değişkenleri ise, **Toplama** sınıf örneğinin birer niteliği oluyor. Bundan önceki örneğimizde ise mesele adlı değişken, deneme adlı sınıf örneğinin bir niteliği idi...

Bu sınıfı yazıp kaydettiğimiz 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 Pythonca'da **örnek** (*instance*) adı veriliyor. Yani **sonuc**, **Toplama** sınıfının bir örneğidir.

Artık,

```
>>> sonuc.a
>>> sonuc.b
>>> sonuc.c
```

biçiminde, **sonuc** örneğinin niteliklerine tek tek erişebiliriz.

Peki, kodları şöyle çalıştırırsak ne olur?

```
>>> import matematik
```

Eğer modülü bu şekilde içe aktarırsak (*import*), sınıfa ulaşmak için şu yapıyı kullanmamız gerekir:

```
>>> sonuc = matematik.Toplama()
```

Yani sınıfı örneklerken 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 bildiriliyorsak, 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** tuşuna 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 Pythonca'da örnek (*instance*) adını veriyoruz. Yani **`sonuc`** adlı değişken, **`Toplama`** adlı sınıfın bir örneğidir diyoruz.

```
>>> sonuc = Toplama()
```

Bu komutu verdikten sonra niteliklerimize erişebiliriz:

```
>>> sonuc.a
>>> sonuc.b
>>> sonuc.c
```

Dikkat ederseniz, niteliklerimize erişirken örnekten (*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 çalıştırıldıktan 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 terk etmesi 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.

__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 ilgileneceğiz. **`self`**'in ne olduğunu şimdilik bir kenara bırakıp, onu olduğu gibi kabul edelim. İşe hemen bir örnekle başlayalım. İsterseniz kendimizce ufacık bir oyun tasarlayalım:

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

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 hallettik. Komut satırına bırakmadık bu işi.

Şimdi bu kodları çalıştıracağız. 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 vermeden önce, **`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** tuşuna 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 işlemesini 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()
  pencere.mainloop()

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…

SELF NEDİR?

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 imkânsız 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`**, **`işci`** 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, isci 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 "xxxxxxxxxxxxxxxxxxx", 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()
```

Gördüğünüz gibi kodlar içinde yazdığımız değişkenlerin, fonksiyon dışından da çağrılabilmesi için, yani bir bakıma **global** bir nitelik kazanması için **self** olarak tanımlanmaları gerekiyor. 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()
  pencere.mainloop()

uygulama = Arayuz()
```

Burada tanımladığımız düğmenin bir iş yapmasını sağlayalım. Mesela; düğmeye basılınca komut ekranında bir yazı çıksın. Önce şöyle deneyelim:

```
# -*- coding: utf-8 -*-
from Tkinter import *

class Arayuz:
 def __init__(self):
  pencere = Tk()
  dugme = Button(text="tamam", command=yaz)
  dugme.pack()
  pencere.mainloop()

 def yaz():
  print "Görüşmek üzere!"

uygulama = Arayuz()
```

Tabii ki bu kodları çalıştırdığımızda şöyle bir hata mesajı alırız:

```
Traceback (most recent call last):
File "xxxxxxxxxxxxxxxxxxx", line 13, in <module>
uygulama = Arayuz()
File "xxxxxxxxxxxxxxxxxxxxxxxx", 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:

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

from Tkinter import *

class Arayuz:
 def __init__(self):
  pencere = Tk()
  dugme = Button(text="tamam", command=self.yaz)
  dugme.pack()
  pencere.mainloop()

 def yaz(self):
  print "Görüşmek üzere!"

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 bilgimiz 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, **command=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:

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

from Tkinter import *

class Arayuz:
 def __init__(armut):
  pencere = Tk()
  dugme = Button(text="tamam", command=armut.yaz)
  dugme.pack()
  pencere.mainloop()

 def yaz(armut):
  print "Görüşmek üzere!"

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.

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
```

Gördüğünüz 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.

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 aynısıyla 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? Tabi 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 aldığında 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 Pythonca'da şöyle ifade edebiliriz:

Dusman sınıfı **Oyun** sınıfını miras aldı.

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 görememizin 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
```

Gördüğünüz 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 büyüdü!"

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 büyüdü!"

dsman = Dusman()
```

En son oluşturduğumuz fonksiyonda nerede **`Oyun`** sınıfını doğrudan adıyla kullandığımıza ve nerede bu sınıfın örneğinden (*instance*) yararlandığımıza dikkat edin. Dikkat ederseniz, fonksiyon başlıklarını çağırırken doğrudan sınıfın kendi adını kullanıyoruz (mesela **`Oyun.__init__(self)`**). Bir fonksiyon içindeki değişkenleri çağırırken ise (mesela macera.fabrika), örneği 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 büyüdü!"

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 var olan bir niteliği iptal de edebiliriz. Diyelim ki **Oyuncu**'nun oyuna başlarken fabrikaları 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 işlemden etkilenmez. Yani aslında **del** komutuyla herhangi bir şeyi sildiğimiz yok! Sadece erişimi engelliyoruz.

Küçük bir not; burada bölge olarak bahsettiğimiz şey aslında Pythonca'da isim alanı (*namespace*) olarak adlandırılıyor.

Şimdi bir örnek de Tkinter ile yapalım. Yukarıda verdiğimiz örneği hatırlıyorsunuz:

```
# -*- coding: utf-8 -*-
from Tkinter import *

class Arayuz:
 def __init__(self):
  pencere = Tk()
  dugme = Button(text="tamam",command=self.yaz)
  dugme.pack()
  pencere.mainloop()

 def yaz(self):
  print "Görüşmek üzere!"

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 "Görüşmek üzere!"

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.

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 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)
```

Gördüğünüz 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 içini doldurma işini, hem de örnek belirleme işini komut satırından halledeceğiz. Önce sınıfımızı örnekliyoruz:

```
sinifimiz = BosSinif()
```

Gördüğünüz 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}
```

Gördüğünüz 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.

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 imkânına kavuşacak ve okuduğunu daha kolay anlayacaktır.

ASCII, UNICODE VE PYTHON

12

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.

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...

ASCII

Ne kadar gelişmiş olurlarsa olsunlar, bugün kullandığımız bilgisayarların anladığı tek şey sayılardır. 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 gelininceye 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 sekmelik 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 biplemeyi (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 "*çşöğüı*" gibi harfleri göremiyoruz...

Şimdi Python'da şöyle bir kod yazdığımızı düşünelim:

```
print "Türkçe karakterler: şçöğüıİ"
```

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 çıktıyı 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 noencoding declared; see http://www.python.
org/peps/pep-0263.html for details
```

Burada gördüğünüz gibi Python, ASCII olmayan karakterler kullandığımızdan yakınıyor. Aslında bu çok doğaldır. Çünkü dediğimiz ve gördüğümüz gibi, "*şçöğüıİ*" 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'nin 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 oluşur. Gayet mantıklı, değil mi?

İşte ASCII sisteminde tanımlanan bütün karakterler 7-bitlik bir sayı içine sığdırılabiliyor. 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 sayısıdır:

```
>>> 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 sayısıdır. 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 gelişiyle 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 imkân 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 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, ASCII dışında kalan karakterler için farklı karakter kümeleri ile farklı çıktılar verir.

Örneğin:

```
>>> print ord("ç")
135
>>> print chr(135)
ç
```

Burada **`ord("ç")`** çıktısı 135 sonucunu veriyor. 135 sayısına karşılık gelen karakter standart ASCII 'nin dışında kaldığı için bu sayı farklı karakter kümelerinde farklı karakterlerle gösterilir. Çünkü dediğimiz gibi standart ASCII sadece 128 karakterden oluşur. Dolayısıyla "*ç*" karakteri ikinci 128'lik kısımda yer alır. Bildiğiniz gibi, ikinci 128'lik kısımda yer alan karakterler bütün karakter kümelerinde aynı değil. Yukarıdaki komutlar 857 numaralı kod sayfasını kullanan bir Windows işletim sisteminde verildi. Örneğin; bu kod sayfasında Türkçe karakterler şu sayılara karşılık geliyor:

```
 >>> for i in 'şçöğüıİ':
 ...         print i, ord(i)
ş 159
ç 135
ö 148
```

```
ğ 167
ü 129
ı 141
İ 152
```

Bu karakterleri, *http://en.wikipedia.org/wiki/Code_page_857* adresindeki kod sayfası ile karşılaştırarak durumu teyit edebilirsiniz.

Yukarıdaki kodlar 1254 numaralı kod sayfasında ise daha farklı bir çıktı verecektir:

```
>>> for i in 'şçöğüıİ':
 ...        print i, ord(i)

ş 254
ç 231
ö 246
ğ 240
ü 252
ı 253
İ 221
```

Bunları da *http://en.wikipedia.org/wiki/Windows-1254* adresindeki kod sayfasından teyit edebilirsiniz.

Gördüğünüz gibi bütün Türkçe karakterler 128'den büyük. O yüzden bunlar standart ASCII tablosuna girmiyor. Ama Microsoft'un kullandığı 857 ve 1254 numaralı kod sayfası sayesinde bu karakterler (birbirlerinden farklı konumlarda olsa da) ikinci 128'lik kısma girebilmiş...

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 sayfasındaki 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...

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 yararlanıldı. 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'dir. 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 dâhil olduğu bazı dillere özgü karakterleri barındırır. Bu sayfada yer alan tabloda her karakterin bir kod konumu bulunduğunu görüyoruz. Mesela; "*ğ*" harfinin kod konumu 011F'dir. 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 mümkün. 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'miş... 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... Dediğim gibi bu kod çözücüler içinde en gözde olanı **UTF-8**'dir ve bu çözücü **GNU/Linux**'ta da standarttır.

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 yansımasının 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 bağdaştıracağımızı 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 = "ışık"
>>> print kardiz.upper()
ışıK
```

Gördüğünüz gibi **`upper()`** metodu, ASCII olmayan karakterler içeren '*ışık*' kelimesini düzgün bir şekilde büyütemedi. Bu sorunun üstesinden gelebilmek için, ascii olmayan karakterler içeren karakter dizilerini Unicode olarak tanımlamamız gerekir:

```
>>> kardiz = u"ışık"
>>> print kardiz.upper()
IŞIK
```

Bu defa '*ışık*' kelimesi doğru bir şekilde büyütülebildi.

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 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.

UNICODE() FONKSİYONU

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 = "şırınga"
>>> print a.upper()
şıRıNGA
```

Gördüğünüz gibi, karakter dizilerinin metotlarından biri olan **`upper()`**, içindeki Türkçe karakterlerden ötürü "*şırınga*" kelimesini büyütemedi. Bu kelimeyi düzgün bir şekilde büyütebilmek için iki yöntem kullanabilirsiniz. Önce basit olanını görelim:

```
>>> a = u"şırınga"
>>> print a.upper()
ŞIRINGA
```

Burada daha en baştan "*şırınga*" karakter dizisini Unicode olarak tanımladık. Bu işlemi, karakter dizisinin başına sadece bir "*u*" harfi getirerek yaptık. Peki ya baştaki karakter dizisini değiştirme imkânımız yoksa ne olacak? İşte bu durumda ikinci yolu tercih edebiliriz:

```
>>> a = "şırınga"
>>> b = unicode(a, "utf-8") #veya cp1254
>>> print b.upper()

ŞIRINGA
```

`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 verirsek, 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'un 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 yarıyor? Hemen bakalım:

```
>>> b = unicode(a, "ascii", errors = "replace")
>>> print b.upper()

????R??NGA
```

Burada da Python çözülemeyen karakterlerin yerine soru işaretleri yerleştirdi...

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()
```

veya

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

a = raw_input("herhangi bir karakter yazınız: ")
print unicode(a, "cp1254", 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...

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 bayt değil Unicode kod konumlarıdır. **unicode()** fonksiyonu içinde kullandığımız utf-8 çözücüsünün görevi ise "*ışık*" kelimesini Python'un doğru okumasını sağlamaktır...

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"
```

Şu da bir unicode dizisi:

```
>>> kardiz = u"elma"
```

Gördüğünüz gibi, Python'da Unicode dizisi oluşturmak oldukça kolay. Yapmamız gereken tek şey karakter dizisinin başına bir "*u*" harfi getirmek.

Unicode dizileri birtakım kod konumlarından meydana gelir. Bu 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 çözücüleri içinde en esnek ve yaygın olanı UTF-8 adlı kod çözücüdür. Python'da kod konumlarını bayta çevirmek için **`encode()`** adlı bir metottan yararlanabiliriz:

```
>>> unidiz = u"ışık"
>>> 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()** metodundan 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...

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")
'Ll'
```

category() metodu ise bir Unicode dizisinin Unicode sistemindeki kategorisini gösteriyor. Yukarıda verdiğimiz örneğe göre "*ğ*" harfinin kategorisi Ll, yani Latin-1.

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.

BİÇİM DÜZENLEYİCİLER

13

Bu bölümde, daha önce sık sık kullandığımız, ancak ayrıntılı bir şekilde hiç incelemediğimiz bir konudan söz edeceğiz. Konumuz; **karakter dizilerinde biçim düzenleyiciler**. İngilizce'de buna **format modifiers** adı veriliyor...

Dediğimiz gibi, biz daha önceki konularımızda biçim düzenleyicilerden yararlanmıştık. Dolayısıyla yabancısı olduğumuz bir konu değil bu.

Python'da her türlü biçim düzenleme işlemi için tek bir simge bulunur. Bu simge `%`'dir. Biz bunu daha önceki bölümlerde 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 sayısıyla, 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 kullanılabilecek tek karakter **s** değildir. Daha önce de dediğimiz gibi, **s** karakteri "*string*", yani "*karakter dizisi*" ifadesinin 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 değerin 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.

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 kullanılabilecek 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.

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 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.

İ Harfi

Bu harf de integer, yani **tamsayı** kelimesinin kısaltmasıdır. Kullanım ve işlev olarak, d harfinden hiç bir farkı yoktur.

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 sayısıdır." %(10, 10)

10 sayısının sekizlik düzendeki karşılığı 12 sayısıdır.
```

x Harfi

Bu harf hexadecimal, yani onaltılık düzendeki sayıları temsil eder:

```
>>> print "%i sayısının onaltılık düzendeki karşılığı \
... %x sayısıdır." %(20, 20)

20 sayısının onaltılık düzendeki karşılığı 14 sayısıdır.
```

Buradaki **x** küçük harf olarak kullanıldığında, onaltılık düzende harfle gösterilen sayılar da küçük harfle temsil edilecektir:

```
>>> print "%i sayısının onaltılık düzendeki karşılığı \
... %x sayısıdır." %(10, 10)

10 sayısının onaltılık düzendeki karşılığı a sayısıdır.
```

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 sayısıdır." %(10, 10)

10 sayısının onaltılık düzendeki karşılığı A sayısıdır.
```

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...

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 üreteci** (*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.

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.

Karakter Dizisi Biçimlendirmede Sözlükleri Kullanmak

Aslında Python'daki karakter dizisi biçimlendiricilerinin, yukarıda verdiğimiz örneklerde görünenden 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"Çarş.",
     "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"Çarş.",
  "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.

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 imkânına 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.

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 adet `0` 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 fark etmişsinizdir. 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
```

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 imkânı verir.

Karakter Dizilerini Hem Hizalamak Hem de Sola Sıfır Eklemek

Yukarıda karakter dizilerini nasıl hizalayacağımızı ve sol taraflarına nasıl 0 ekleyebileceğimizi ayrı ayrı gördük. Esasında Python'daki biçim düzenleyiciler bize bu işlemlerin her ikisini birlikte yapma imkânı 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...

FORMAT() METODU İLE BİÇİMLENDİRME

Python programlama dilinde karakter dizisi biçimlendirme işlemleri için en yaygın kullanılan işaret yukarıda örneklerini verdiğimiz % işaretidir. Ancak Python'un 3.x sürümleri ile birlikte karakter dizisi biçimlendirme işlemleri için format adlı bir metot da eklendi dile.

Bu metodu, hatırlarsanız karakter dizisi metotlarını listelediğimiz bölümde metot listesi arasında görmüş, ama o zaman bu metodu anlatma işini ertelemiştik. İşte bu bölümde **`format()`** adlı bu metottan kısaca söz edeceğiz.

Dediğimiz gibi **`format()`** metodu, aslında dile Python3 ile girmiş bir özelliktir. Ancak bu özellik Python'un 2.7 sürümlerine de aktarıldığı için Python'un 2.6 sonrası (2.6 hariç) sürümlerinde bu özelliği kullanabilirsiniz.

`format()` metodu da, tıpkı karakter dizilerinin öteki metotları gibi kullanılır. Hemen basit bir örnek verelim.

Bildiğiniz gibi% işaretini kullanarak aşağıdaki gibi bir kod yazabiliyoruz:

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

Aynı karakter dizisini, **`format()`** metodunu kullanarak ise şu şekilde yazabiliriz:

```
>>> print "{} ve {} iyi bir ikilidir!".format('Python', 'Django')
Python ve Django iyi bir ikilidir!
```

Yukarıdaki kodu şöyle de yazabilirsiniz:

```
>>> metin = "{} ve {} iyi bir ikilidir!"
>>> print metin.format('Python', 'Django')
```

Burada karakter dizisini önce bir değişkene atadık, daha sonra bu değişkene **`format()`** metodunu uyguladık.

Gördüğünüz gibi, bu metodun gerçekten de öteki karakter dizisi metotlarından hiçbir farkı yok. Bu metot da diğerleriyle tamamen aynı şekilde kullanılıyor. Yapı olarak ise, eski yöntemdeki `%` işaretinin yerini `{}` işaretleri alıyor.

format() metodunun bazı ilginç özellikleri bulunur. Mesela **{}** işaretleri içinde bir sıra numarası vererek, **format()** metoduna verdiğimiz parametrelerin sırasını istediğimiz şekilde düzenleyebiliriz:

```
>>> print '{1} ve {0} iyi bir ikilidir!'.format('Python', 'Django')
Django ve Python iyi bir ikilidir!
```

Normal şartlar altında, **{}** işaretleri arasında herhangi bir sayı belirtmediğimizde Python; **format()** metodunun parantezleri arasındaki değerleri, karakter dizisinin içine, parantez içinde bulundukları sıra ile yerleştirecektir. Yukarıdaki örnekten de gördüğünüz gibi, biz bu sıralamayı değiştirme imkanına sahibiz. Yukarıda yazdığımız kod sayesinde '*Django*' değerini, karakter dizisi içine 'Python' değerinden önce yerleştirebildik.

% işareti ile biçimlendirme konusunu anlatırken, bahsettiğimiz biçimlendirme karakterlerinden elbette **format()** metodundan da yararlanabiliriz. Şu örneklere dikkatlice bakın:

```
>>> print '{:c}'.format(100) #sayının karakter karşılığı
d

>>> print '{:x}'.format(100) #sayının onaltılık karşılığı
64

>>> print '{:o}'.format(100) #sayının sekizlik karşılığı
144

>>> print '{:f}'.format(100) #sayının kayan noktalı karşılığı
100.000000

>>> print '{:.2f}'.format(100) #noktadan sonraki hane sayısı
100.00
```

Gördüğünüz gibi, özel biçimlendirme işaretlerini **{}** işaretleri arasına **:** işaretinden sonra yerleştiriyoruz.

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.

KISIM II

ÖZEL KONULAR

MATH MODÜLÜ

1

İleri düzey modüllerden biri olan **math** modülü matematikle uğraşanların işlerini bir hayli kolaylaştıracak metotlar ve fonksiyonlar barındırır.

Python'da matematiksel fonksiyonları **math** modülü ile kullanmaktayız. Şimdi math modülümüzün içeriğini görelim. Unutmadan modülümüzü çalışmamıza çağıralım:

```
>>> import math
```

Bu komut ile modülümüzü çalışmamıza dâhil etmiş olduk. Şimdi içerdiği fonksiyonları aşağıdaki komutu vererek görelim:

```
>>> dir(math)
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan',
 'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp',
 'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',
 'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
 'sqrt', 'tan', 'tanh']
```

Modülümüzün içeriğini de gördüğümüze göre şimdi kosinüs, sinüs, tanjant, pi, karekök, üslü ifadeler gibi fonksiyonlarla ilgili örneklerle kullanımını anlamaya çalışalım.

Üslü İfadeler (pow)

Üslü ifadeler matematikte hep karşımıza çıkmıştır. Python'da bunun için **pow()** fonksiyonu kullanılır. Fonksiyon adında geçen **pow** ifadesi; **power** kelimesinin kısaltması olup Türkçe'de **kuvvet**, **üs** anlamlarına denk gelmektedir. Örnek verecek olursak 23 ifadesinin Python'daki karşılığı:

```
>>> math.pow(2, 3)
```

şeklindedir. Yukarıdaki kodu yazdığımızda Python bize cevap olarak şunu gösterir:

```
8.0
```

pow() fonksiyonunun kullanımında parantez içerisindeki ilk sayı tabanı, ikinci sayı ise üssü göstermektedir.

Pi sayısı (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 hacmini şu formülle buluyoruz: **Küre hacmi = 4/3(π.r^3)**

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.

Karekök (sqrt)

Karekök ile ilgili fonksiyonumuz **`sqrt()`**. Kullanımı ise aşağıdaki gibidir:

```
>>> math.sqrt(9)
```

kodunu yazıp **ENTER** tuşuna bastığımızda Python bize aşağıdaki sonucu gösterir:

```
3.0
```

Euler Sabiti (e)

Bu fonksiyon matematikteki euler sabitini veriyor. Kullanımı ise aşağıdaki gibi:

```
>>> math.e
```

Yukarıdaki kodu yazıp **ENTER** tuşuna bastığımızda karşımıza euler sabiti 2.7182818284590451 cevap olarak Python tarafından gösteriliyor. Kullanım açısından aynı pi sayısı gibi:

```
2.7182818284590451
```

EXP() FONKSİYONU

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 karesi*".

LOGARİTMA (LOG)

Logaritma ile ilgili 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
```

LOG10() FONKSİYONU

Bu fonksiyonun yukarıdakinden tek farkı taban sayısının önceden belirlenmiş ve 10 olması. Bu yüzden fonksiyonun kullanımı şöyle; **log10(x)** Burada x onluk tabana göre logaritması alınacak sayıdır:

```
>>> math.log10(10)
```

Dönen cevap:

```
1.0
```

DEGREES() FONKSİYONU

degrees() fonksiyonu girilen açıyı radyandan dereceye çevirmeye yarar. Kullanımı şu şekilde;

```
>>> math.degrees(x)
```

x, burada radyana çevrilecek açımızdır. Örnek olarak 45 derecelik bir açı verelim ve radyan karşılığını bulalım:

```
>>> math.degrees(45)
2578.3100780887044
```

RADİANS() FONKSİYONU

radians() fonksiyonu girilen açıyı dereceden radyana çevirmeye yarar. Kullanımı şu şekilde;

```
>>> math.radians(x)
```

x burada dereceye çevrilecek radyandır. Örnek olarak 45 radyanlık bir açı verelim ve derece karşılığını bulalım:

```
>>> math.radians(45)
0.78539816339744828
```

Kosinüs, Sinüs ve Tanjant ile ilgili fonksiyonlara girmeden önce belirtmem gereken önemli bir nokta bulunmaktadır. Bu fonksiyonlarda açı olarak Python, radyan kullanmaktadır. Bu yüzden aldığımız sonuçlar okulda öğrendiğimiz değerlerden farklı olacaktır. Bunu da **radians()** fonksiyonu ile düzelteceğiz.

KOSİNÜS (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
```

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.

Sinüs (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.

Tanjant (tan)

Tanjant fonksiyonu ile bir örnek yapalım:

```
>>> math.tan(0)
```

Python bunun sonucu olarak bize:

```
0.0
```

cevabını verir.

```
>>> math.tan(45)
1.6197751905438615
```

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
```

Bu da 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:

$$R^2 = F_1^2 + F_2^2 + 2 * F_1 * F_2 * \cos(\infty)$$

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 yapılabilir.

PYTHON'DA ID() FONKSİYONU, IS İŞLECİ VE ÖNBELLEKLEME MEKANİZMASI

2

Python'da her nesnenin bir kimliği (*identity*) vardır. Kabaca söylemek gerekirse, 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 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, oluşturduğumuzda Python açısından bir nesneye dönüşüverecektir 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 oluşturmuyor.

İlk kullanımda önbelleğine aldığı sayıyı, ikinci kez ihtiyaç olduğunda bellekten alıp kullanıyor. Ama bir de şu örneklere 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** 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","kebap"]
>>> id(liste)

3082284940L
```

Bunu başka bir liste üzerinde daha deneyelim:

```
>>> liste2 = ["elma","armut","kebap"]
>>> id(liste2)

3082284172L
```

Gördüğünüz gibi, içerik aynı olduğu halde iki listenin kimliği birbirinden farklı... Python bu iki listeyi bellekte iki farklı adreste depoluyor. Ama bu listelerin ait olduğu veri tipi (yani **`list`**), bellekte tek bir adreste depolanacaktır. Çünkü bir nesnenin veri tipinin kendisi de başlıbaşına bir nesnedir. Nasıl **`liste2`** nesnesinin kimlik numarası bu nesne ortalıkta olduğu sürece aynı kalacaksa, bütün listelerin ait olduğu büyük **`list`** veri tipi nesnesi de tek bir kimlik numarasına sahip olacaktır.

```
>>> id(type(liste))

>>> 3085544992L
>>> id(type(liste2))

3085544992L
```

Bu iki çıktı aynıdır, çünkü Python **`list`** veri tipi nesnesi için bellekte tek bir adres kullanıyor. Ama ayrı listeler için ayrı adres kullanıyor. Aynı durum tabii ki öteki veri tipleri için de geçerlidir. Mesela **`dict`** (*sözlük*) veri tipi.

```
>>> sozluk1 = {}
>>> id(sozluk1)

3082285236L

>>> sozluk2 = {}
>>> id(sozluk2)

3082285916L
```

Ama tıpkı **`list`** veri tipinde olduğu gibi **`dict`** veri tipinin nesne kimliği de hep aynı olacaktır:

```
>>> id(type(sozluk1)

3085549888L

>>> id(type(sozluk2)

3085549888L
```

Peki, biz bu bilgiden nasıl yararlanabiliriz? Şöyle: **`is`** işlecini doğrudan iki nesnenin kendisini karşılaştırmak için kullanamasak da bu nesnelerin veri tipini 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 veri tipine sahiptir"
        print "Nesnelerin tipi: %s"%(type(a))
    else:
        print "Bu iki nesne aynı veri tipine 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 veri tipi 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'U YOL'A (PATH) EKLEMEK

3

GNU/Linux kullananların Python'un 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'u 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 durumda Python'un etkileşimli kabuğuna ulaşmak için **Başlat / Programlar / Python 2.x/Python (Command Line)** yolunu takip etmek gerekiyor. Ancak etkileşimli kabuğa bu şekilde ulaşmanın bazı dezavantajları var. Örneğin; bu şekilde kabuğa ulaştığınızda Python'u istediğiniz dizin içinde başlatamamış oluyorsunuz. Bu şekilde Python'un etkileşimli kabuğu, Python'un 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 açamadığınız zaman, örneğin masaüstüne kaydettiğiniz bir modüle ulaşmak için biraz daha fazla uğraşacaksınız demektir...

Ayrıca komut satırından;

```
python program_adı
```

yazarak programınızı çalıştırma imkanınız da olmayacaktır. Daha doğrusu, bu imkânı elde etmek için Windows komut satırına daha fazla kod yazmanız gerekecek. Yani Python'a ulaşmak için, komut satırına her defasında Python'un ve/veya kendi programınızın tam yolunu yazmak zorunda kalacaksınız. 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 kolaylığı Windows'ta nasıl yapabileceğimizi anlatacağız. Yani bu yazımızda, Windows'ta sadece **`python`** komutunu vererek nasıl etkileşimli kabuğa ulaşabileceğimizi öğreneceğiz.

Öncelikle Windows'un masaüstündeki **Bilgisayarım** simgesine sağ tıklıyoruz.

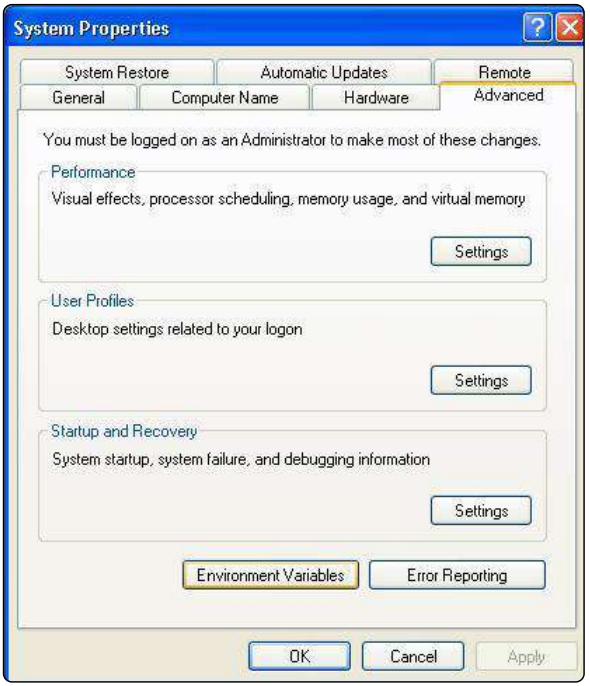


Bilgisayarım simgesine sağ tıkladıktan sonra açılan menünün en altında yer alan **Özellikler**'e (*Properties*) giriyoruz.

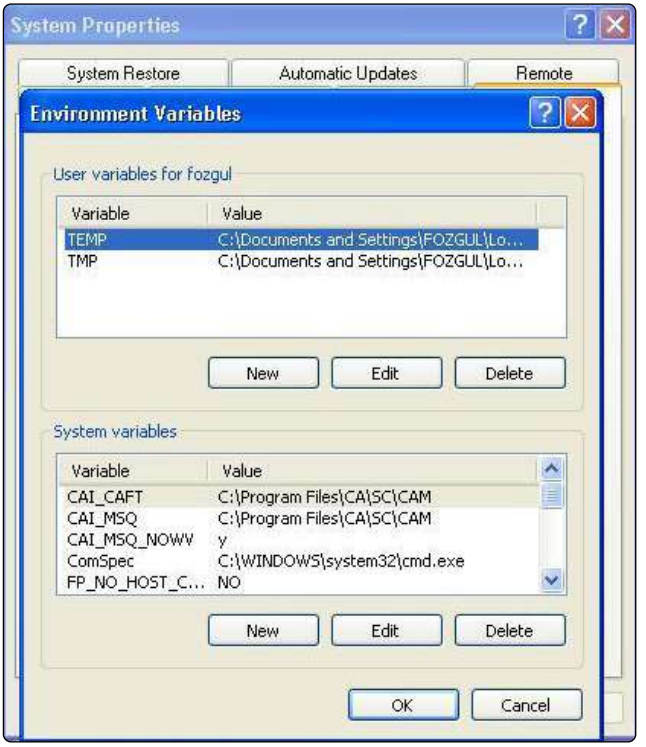
Bu girdiğimiz yerde **Gelişmiş** (*Advanced*) adlı bir sekme göreceğiz. Bu sekmeyi açıyoruz.



Gelişmiş sekmesine tıkladığımızda karşımıza şöyle bir ekran gelecek:

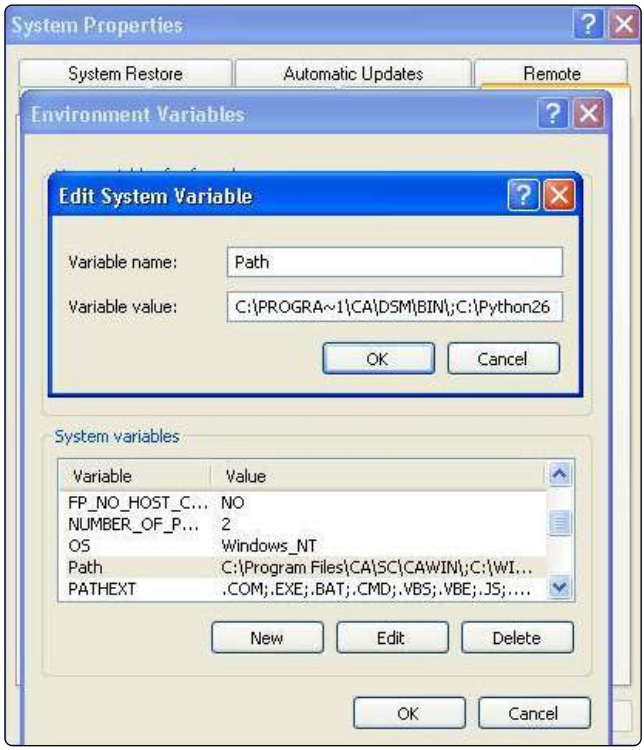


Burada **Çevre Değişkenleri** (*Environment Variables*) düğmesine tıklıyoruz. Bu düğmeye tıkladığımızda şöyle bir pencere açılacak:



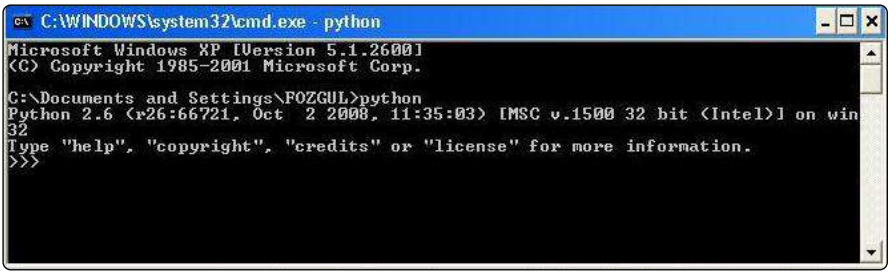
Bu ekranda **Sistem Değişkenleri** (*System Variables*) bölümünde yer alan liste içinde **Path** öğesini buluyoruz. Listedeki öğeler alfabe sırasına göre dizildiği için **Path**'i bulmanız zor olmayacaktır.

Path öğesi seçiliyken, **Sistem Değişkenleri** bölümündeki **Düzenle** (*Edit*) düğmesine tıklıyoruz. Karşımıza şöyle bir şey geliyor: (Aşağıdaki ekran görüntüsünde **Sistem Değişkenleri** bölümündeki **Path** öğesini de listede görebilirsiniz.)



Bu ekrandaki listenin en sonunda görünen `;C:\Python26` öğesine dikkat edin. Siz de listenin sonuna, sisteminizde kurulu olan Python sürümüne uygun ifadeyi ekleyeceksiniz. Yukarıdaki ekran görüntüsü Python'un 2.6 sürümüne göre. Eğer sizde mesela Python 2.7 kuruluysa oraya `;C:\Python27` yazacaksınız. Sizdeki sürümün tam adını öğrenmek için **C:/** dizinini kontrol edebilirsiniz. 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'un etkileşimli kabuğuna ulaşabiliriz.



Etkileşimli kabuktan çıkmak için önce **CTRL+Z** tuşlarına, ardından da **ENTER** tuşuna basıyoruz. Artık `cmd` ile ulaştığımız bu komut satırında şu komutu vererek Python programlarını çalıştırabiliriz:

```
python program_adı
```

Windows komut satırı Python'un Windows'ta sunduğu komut satırından biraz daha yeteneklidir. `cmd` komutu ile ulaştığınız ekranda kopyalama/yapıştırma işlemleri de yapabilirsiniz.

FARKLI PYTHON SÜRÜMLERİ

4

Python programlama dilini yakından takip ediyorsanız, şu anda piyasada iki farklı Python sürümünün olduğu dikkatinizi çekmiş olmalı. Şubat 2016 tarihi itibariyle resmi Python sitesinden (*https://www.python.org/downloads*) indirebileceğimiz en yeni Python sürümleri Python 2.7.11 ve Python 3.5.1'dir.

Eğer bir Python sürümü **2** sayısı ile başlıyorsa (mesela 2.7.11), o sürüm Python 2.x serisine aittir. Yok eğer bir Python sürümü **3** ile başlıyorsa (mesela 3.5.1), o sürüm Python 3.x serisine aittir.

Peki, neden piyasada iki farklı Python serisi var ve bu bizim için ne anlama geliyor?

Python programlama dili 1990 yılından bu yana geliştirilen bir dil. Bu süre içinde pek çok Python programı yazıldı ve insanların kullanımına sunuldu. Şu anda piyasadaki çoğu Python programı 2.x serisinden bir sürümle yazılmış durumda 3.x serisi ise yeni yeni yaygınlık kazanıyor.



Biz bu makalede kolaylık açısından Python'ın 3.x serisini Python3, 2.x serisini ise Python2 olarak adlandıracağız.

Python3, Python2'ye göre hem çok daha güçlüdür, hem de Python2'nin hatalarından arındırılmıştır. Python3'teki büyük değişikliklerden ötürü, Python2 ile yazılmış bir program Python3 altında çalışmayacaktır. Aynı durum bunun tersi için de geçerlidir. Yani Python3 kullanarak yazdığınız bir program Python2 altında çalışmaz.

Dediğimiz gibi, piyasada Python2 ile yazılmış çok sayıda program var. İşte bu sebeple Python geliştiricileri uzun bir süre daha Python2'yi geliştirmeye devam edecek. Elbette geliştiriciler bir yandan da Python3 üzerinde çalışmayı ve bu yeni seriyi geliştirmeyi sürdürecek.

Farklı Python serilerinin var olmasından ötürü, Python ile program yazarken hangi seriye ait bir sürümü kullandığınızı bilmeniz, yazacağınız programın kaderi açısından büyük önem taşır.

Python 3.x serisinin ilk kararlı sürümü 3 Aralık 2008 tarihinde yayımlandı. Ancak Python3 eski sürümlerle uyumlu olmadığı için insanlar hala Python3'e geçmekte tereddüt ediyor. Çünkü hem etraftaki Python programları, hem Python'la ilgili kaynaklar hem de önemli üçüncü şahıs modülleri henüz Python3'e taşınmadı. Dolayısıyla, henüz Python3'e taşınmamış modüllere bağımlılık duyan Python programlarının yazarları Python3'e geçiş konusunda ayak diretiyor.

Python geliştiricileri, bir yandan Python3'ü geliştirmekle uğraşırken, bir yandan da insanların Python3'e geçişini hızlandırmak ve korkuları gidermek için bilgilendirici ve yüreklendirici makaleler yayımlıyor. Bu yüreklendirici makalelerden biri de *http://wiki.python.org/moin/Python2orPython3* adresinde bulunuyor. Bu makalede, Python kullanıcılarının sıklıkla sorduğu "Python3'ü mü, yoksa Python2'yi mi kullanmalıyım?" sorusuna bir cevap vermeye çalışıyor Python geliştiricileri… Yukarıdaki makalenin Türkçe çevirisine *http://www.istihza.com/blog/python-3-ve-python-2.html/* adresinden ulaşabilirsiniz.

Peki, Python3 bize ne tür yenilikler getiriyor? Bu sorunun cevabını aşağıda madde madde vermeye çalışacağız.

Python3'te `print` deyim değil, fonksiyondur.

Bildiğiniz gibi, Python2'de herhangi bir şeyi ekrana çıktı olarak vermek istediğimizde şöyle bir kod yazıyorduk:

```
>>> print "Merhaba Zalim Dünya!"
```

Burada **`print`** bir deyimdir, ama artık bu kod geçerli değil. Python3'te aynı kodu şöyle yazmamız gerekiyor:

```
>>> print("Merhaba Zalim Dünya!")
```

Gördüğünüz gibi, Python2'de **`print`** olarak yazdığımız kod Python3'te **`print()`** biçimini alıyor. Yani Python3'te **`print`** bir fonksiyon olarak karşımıza çıkıyor.

Python3'te `range()` fonksiyonu liste çıktısı vermez.

Python2'de **`range()`** fonksiyonunu şöyle kullanıyorduk:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Gördüğünüz gibi, bu fonksiyon bize çıktı olarak bir liste veriyor. Gelelim Python3'teki duruma:

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

Bu çıktının Python2'deki çıktıdan farklı olduğunu görüyorsunuz. Eğer Python3'te de **`range()`** fonksiyonundan Python2'deki çıktıyı almak istiyorsanız, **`range()`** fonksiyonunun çıktısını elle bir liste haline getirmelisiniz:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Veya dilerseniz **`range()`** fonksiyonu üzerinde bir döngü de kurabilirsiniz:

```
>>> for i in range(10):
...     print(i)
```

Python3'te `raw_input()` diye bir fonksiyon yoktur.

Python3 üzerindeki ilk çalışmalar yapılırken, dildeki **`input()`** ve **`raw_input()`** fonksiyonlarının tamamen kaldırılması düşünülüyordu. Bunların yerine şöyle bir şey öneriliyordu:

```
>>> import sys
>>> print("isminiz nedir? ")
>>> isim = sys.stdin.readline()
```

Neyse ki bu düşüncenin işi gereksiz yere karmaşıklaştırdığı anlaşıldı ve nihai olarak Python2'deki **`input()`** fonksiyonunun tamamen kaldırılmasına, bunun yerine eski **`raw_input()`** fonksiyonunun da adının **`input()`** olarak değiştirilmesine karar verildi.

Dolayısıyla Python2'de **`raw_input()`** ve/veya **`input()`** fonksiyonuyla yaptığımız her şeyi artık Python3'te **`input()`** fonksiyonuyla yapacağız.

Python2'deki **`input()`** fonksiyonu, kullanıcıdan gelen veriyi bir Python komutu olarak yorumluyordu. Yani **`input()`** kullanılarak yazılmış bir programda kullanıcı herhangi bir Python komutunu doğrudan çalıştırma imkânı elde ediyordu. Bu durum aslında güvenlik açısından bazı sakıncalar doğurur. Örneğin; **`input()`** ile yazılmış bir programa şu cevabı verdiğimizi düşünelim:

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

Bu komut, programın çalıştığı sistemdeki mevcut çalışma dizininin içeriğini gösterir. Elbette kötü niyetli bir kullanıcı, örneğin sunucu üzerinde çalışan bir uygulamanın çok daha yıkıcı komutları çalıştırmasına yol açabilir.

Eğer Python2'deki **`input()`** fonksiyonunun bu işlevini Python3'te de elde etmek istersek şu komutu kullanacağız:

```
>>> eval(input())
```

`raw_input()`'un **`input()`**'a dönüşme sürecini **PEP-3111**'den takip edebilirsiniz (*http://www.python.org/dev/peps/pep-3111/*).

Python3'te unicode ve string (karakter dizisi) ayrımı yoktur.

Python2'de iki farklı karakter dizisi biçimi vardı. Bunlardan birisi düz karakter dizileri (strings) ikincisi ise Unicode karakter dizileriydi. Python2'de düz karakter dizilerini şöyle gösteriyorduk:

```
>>> "elma"
```

Unicode karakter dizilerini ise şöyle gösteriyorduk:

```
>>> u"elma"
```

Python3'le birlikte bu ayrım ortadan kalktı. Artık bütün karakter dizileri Unicode olarak temsil ediliyor. Bu ayrım ortadan kalktığı için, Python3'te `u"elma"` tarzı bir kod yazımı herhangi bir etkiye sahip olmayacaktır.

Python3'te öntanımlı kodlama biçimi UTF-8'dir.

Bildiğiniz gibi, içinde Türkçe karakterler barındıran bir Python kodu yazdığımızda, eğer programın ilk satırına `# -*- coding: utf-8 -*-` veya `# -*- coding: cp1254 -*-` gibi bir satır eklemezsek şöyle bir hata alıyorduk:

```
SyntaxError: Non-ASCII character '\xc4' in file deneme.py on line 3,
but no encoding declared; see http://www.python.org/peps/pep-0263.
html for details.
```

Bunun sebebi Python'un öntanımlı kod çözücüsünün ASCII olmasıydı. Ama Python3'le birlikte öntanımlı kod çözücüsü UTF-8 olduğu için artık `# -*- coding: utf-8 -*-` satırını yazmamıza gerek yok. Ancak eğer dosyalarınız için UTF-8 dışında farklı bir kodlama biçimi kullanacaksanız elbette uygun satırı eklemeniz gerekiyor.

Python'un öntanımlı kod çözücüsünün UTF-8 olarak değiştirilmesi Python2'de yaşadığımız pek çok Türkçe probleminin de artık yaşanmamasını sağlayacaktır.

Python3'te değişken adlarında Türkçe karakterler kullanılabilir.

Python2'de değişken adlarında Türkçe karakter kullanamıyorduk. Ama Python3'le birlikte bu sınırlama ortadan kaldırıldı. Dolayısıyla artık şöyle değişkenler tanımlayabiliyoruz:

```
>>> şekil = "dikdörtgen"
>>> sayı = 23
```

Bu sayede `cik` gibi tuhaf değişken adları yerine `çık` gibi daha anlamlı değişken adları tanımlayabileceğiz...

Python3'te bölme işlemleri tam sonuç verir.

Bildiğiniz gibi, Python2'de şöyle bir durum söz konusu idi:

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

Buradan 2.5 çıktısı alabilmek için yukarıdaki kodu şöyle yazmamız gerekiyordu:

```
>>> 5.0 / 2
2.5
```

Python3'le birlikte bu durum tamamen değişti:

```
>>> 5 / 2
2.5
```

Eğer Python2'deki çıktıyı Python3'te de almak isterseniz yukarıdaki aritmetik işlemi şöyle yapmanız gerekiyor:

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

Python3'te dile sıralı sözlükler eklenmiştir.

Bildiğiniz gibi, Python sözlüklerinde sıra diye bir kavram yoktur. Yani sözlüklerin öğelerine sıralı bir şekilde erişemeyiz. Buna bir örnek verelim:

```
>>> sozluk = {"a": 1, "b": 2, "c": 3}
>>> sozluk
{'a': 1, 'c': 3, 'b': 2}
```

Gördüğünüz gibi sözlük çıktısındaki öğe sıralaması ile sözlüğü tanımlarken belirlediğimiz öğe sıralaması aynı değil.

Ancak Python3 ile birlikte `OrderedDict()` adlı yeni bir fonksiyonumuz oldu. Bunu şöyle kullanıyoruz:

```
>>> from collections import OrderedDict
>>> sıralı_sözlük = OrderedDict([("a", 1), ("b", 2), ("c", 3)])
>>> sıralı_sözlük
OrderedDict([('a', 1), ('b', 2), ('c', 3)])
```

Gördüğünüz gibi, bildik sözlüklerin aksine `OrderedDict()` çıktısı sıralı görünüyor.



Bu özellik Python'un 2.7 sürümlerine de aktarıldığı için, bu özelliği Python 2.7'de de kullanabiliyoruz.

Python3'te karakter dizilerinin `format()` adlı yeni bir metodu bulunur.

Python2'de biçim düzenleyici olarak şöyle bir yapıdan yararlanıyorduk:

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

Python3'te ise aynı işlevi şu şekilde yerine getiriyoruz:

```
>>> "{0} ile {1} iyi bir ikilidir.".format("Python", "Django")
```

Eğer isterseniz köşeli parantezler içindeki sayıları da atabilirsiniz:

```
>>> "{} ile {} iyi bir ikilidir.".format("Python", "Django")
```

Bu sayıları kullanarak öğelerin sırasını da değiştirebilirsiniz:

```
>>> "{1} ile {0} iyi bir ikilidir.".format("Python", "Django")
```

`format()` fonksiyonunun çok daha gelişmiş yetenekleri vardır, ancak bu fonksiyonun temel kullanımı yukarıda gösterdiğimiz gibidir.

Python2'deki `%s` yapısını yine kullanmaya devam edebiliriz. Ancak bu yapının ileriki bir Python sürümünde kullanımdan kaldırılma ihtimali var. Ayrıca yeni **`format()`** fonksiyonu eski **`%s`** yapısına göre çok daha esnek ve yeteneklidir.



Bu özellik Python'un 2.7 sürümlerine de aktarıldığı için, bu özelliği Python 2.7'de de kullanabiliyoruz.

Python3'te Tkinter modülünün adı tkinter olarak değiştirildi.

Python2'de Tkinter modülünü içe aktarmak için şu komutu kullanıyorduk:

```
>>> from Tkinter import *
```

Python3'te bu modülün adı tkinter olarak değiştirildiği için artık bu modülü şöyle içe aktaracağız:

```
>>> from tkinter import *
```

Python3'e ttk modülü eklendi.

Tkinter'e ilişkin en büyük eleştiri, bu arayüz takımıyla üretilen programların çok çirkin görünmesi idi. Ancak Python3'le birlikte gelen (ve Python 2.7'ye de aktarılan) **`ttk`** modülü sayesinde eskisine kıyasla çok daha alımlı Tkinter programları yazmak mümkün olacaktır. Ayrıca bu modülle birlikte Tkinter'deki pencere aracı (*widget*) sayısında da artış oldu.

Python3'te tkMessageBox modülünün adı değişti.

Python2'de **`tkMessageBox`** modülünü şu şekilde içe aktarıyorduk:

```
>>> from tkMessageBox import *
```

Python3'te ise aynı modülü şu şekilde içe aktarıyoruz:

```
>>> from tkinter.messagebox import *
```

Böylece Python3'le birlikte gelen temel yenilikleri bir çırpıda sıralamış olduk. Elbette Python3'le gelen başka yenilikler de var. Ancak yukarıda sıraladığımız özellikleri öğrenmeniz Python2'den Python3'e geçişinizin olabildiğince sancısız olmasını sağlamaya yetecektir.

En başta da söylediğimiz gibi, Python'un 3.x sürümleri 2.x sürümlerine kıyasla büyük farklılıklar içeriyor. Bu sebepten, Python2 ile yazılan programlar Python3 altında çalışmayacak. Ancak sürümler arasında büyük farklar olsa da nihayetinde dil yine aynı dil ve Python3'le gelen farklılıklar programcılara büyük zorluklar getirecek cinsten değil.

Şu anda Python3'ün en büyük sorunu, üçüncü şahıs modülleri açısından yetersiz olması. Python'la işe yarar programlar yazmamızı sağlayan pek çok üçüncü şahıs modül, henüz Python3'e uyumlu değil. Üstelik GNU/Linux dağıtımlarının ezici çoğunluğu da henüz öntanımlı olarak Python'un 2.x sürümlerini kullanıyor. Bu ve buna benzer sebeplerden ötürü, şu an için mantıklı olan Python 2.x ile devam edip, bir yandan da Python 3.x ile ilgili gelişmeleri takip etmektir.

KISIM III

ARAYÜZ

GRAFİK ARAYÜZ TASARIMI / TEMEL BİLGİLER

1

Bir programlama dilini ne kadar iyi bilirseniz bilin, bu programlama diliyle ne kadar güçlü programlar yazarsanız yazın, eğer programınız kullanışlı bir arayüze sahip değilse; programınızın kullanıcı düzeyinde pek ilgi çekmeyeceği kesin...

Belki bilgisayar kurtları komut satırından çalışan programları kullanmaktan rahatsız olmayacaktır. Hatta sizi takdir de edecektir, ama son kullanıcı denen kesime hitap etmediğiniz sürece dikkati çekmeniz güç. Bu dikkati çekme meselesi de çoğunlukla kullanıcıya pırıltılı arayüzler sunularak aşılabiliyor. Diyelim ki komut satırı üzerinden harika işler yapan muhteşem bir program yazdınız ve programınızı kullanıma sundunuz. Son kullanıcı, programınızı kurup çalıştırmayı denediğinde vereceği ilk tepki: "*Ama bu program çalışmıyor!*" olacaktır. Kızsak da kızmasak da durum bundan ibarettir.

Madem arayüz denen şey bu kadar önemli. Şu halde biz de bu bölümde Python'da nasıl grafik arayüzler tasarlayabileceğimizi inceleyelim.

Python'da arayüz tasarımı için birden fazla seçeneğimiz var. Bunları bir çırpıda sıralayacak olursak, seçeneklerimizin şunlar olduğunu görürüz:

- PyGTK (*http://www.pygtk.org/*)
- PyQt (*http://www.riverbankcomputing.co.uk/news*)
- wxPython (*http://www.wxpython.org/*)
- Tkinter (*http://wiki.python.org/moin/TkInter*)

Arayüz tasarlamak üzere geliştirilmiş yukarıdaki alternatifler içinde biz Python tarafından resmi olarak desteklenen Tkinter'i kullanacağız. Peki, neden ötekiler değil de Tkinter?

Öncelikle Tkinter'in öteki seçeneklere karşı en büyük üstünlüğü Python dağıtımı ile birlikte gelmesidir. Dolayısıyla Tkinter Python tarafından resmi olarak desteklenen bir arayüz tasarım takımıdır. Tkinter hem Windows'ta hem de GNU/Linux üzerinde rahatlıkla çalışabilir. Aslında artık günümüzde öteki arayüz takımları da birden fazla platform üzerinde çalışabilmektedir. Ama dağıtımla birlikte geldiği için Tkinter ötekilere göre çok daha kolay ve rahat bir şekilde kurulup kullanıma hazır hale getirilebilir. Eğer Windows kullanıyorsanız, Tkinter sizde zaten Python'la beraber sisteminize kurulmuştur. Eğer GNU/Linux kullanıyorsanız, kullandığınız dağıtımın paket yöneticisini kullanarak `tkinter` paketini sisteminize kurabilirsiniz. Farklı GNU/Linux dağıtımlarında bu paketin adı farklı olabilir. Paket yöneticiniz içinde `tk` şeklinde arama yaparsanız muhtemelen doğru paketi bulabilirsiniz. Paket adı **python-tk** veya **python2.x-tk** gibi bir şey olacaktır.

Tkinter oldukça sağlam, kararlı ve oturmuş bir arayüz takımıdır. Üstelik kullanımı da oldukça basittir. Öteki seçeneklere göre çok daha temiz ve sade bir sözdizimi vardır. Tkinter'in tek eksiği, bu arayüz takımıyla yapılan programların biraz çirkin görünmesidir. Ama mesela, resim düzenleme yazılımlarını kullanarak Tkinter'in görünümünü düzeltmek o kadar da zor bir iş değildir. Ben şahsen, Python programcılarının Tkinter'i kullanmayacak bile olsalar, en azından temelini bilmeleri gerektiğini düşünüyorum. Tabii ki sizler Tkinter'i öğrendikten sonra öteki seçenekleri de incelemek isteyebilirsiniz.

Aslında Tkinter Python içinde bir modüldür. Lafı daha fazla uzatmadan işe koyulalım.

Pencere Oluşturmak

Arayüz denilen şey tabii ki penceresiz olmaz. Dolayısıyla arayüz programlamanın ilk adımı çalışan bir pencere hazırlamak olacaktır.

Başta söylediğimiz gibi, arayüz tasarlarken Tkinter modülünden faydalanacağız. Daha önceki yazılardan, Python'da nasıl modül import edildiğini hatırlıyorsunuz.

Şimdi hemen Python'un etkileşimli kabuğunda şu komutu veriyoruz:

```
from Tkinter import *
```

Eğer hiçbir şey olmadan alt satıra geçildiyse sorun yok. Demek ki sizde Tkinter modülü yüklü. Ama eğer bu komutu verdiğinizde alt satıra düşmek yerine bir hata mesajı alıyorsanız sebebi; gerekli modülün, yani Tkinter'in sizde yüklü olmamasıdır. Eğer hal böyleyse yapacağımız işlem çok basit: Kullandığımız GNU/Linux dağıtımının paket yöneticisinden Tkinter modülünü içeren paketi gidip kuracağız. Benim şu anda kullandığım dağıtım olan **Ubuntu**'da bu paketin adı **python2.7-tk**. Sizde de paketin adı buna benzer bir şey olmalı. Mesela; **Pardus**'ta Tkinter modülünü kullanmak için kurmanız gereken paketin adı **python-tk**. Dediğim gibi, eğer Windows kullanıyorsanız, sizde Tkinter modülü zaten yüklüdür. Windows kullanıcılarının Tkinter modülünü kullanabilmesi için ayrı bir şey yüklemesine gerek yok.

Modülle ilgili kısmı sağ salim atlattıysanız, artık şu komutu verebilirsiniz:

```
Tk()
```

Eğer buraya kadar herhangi bir hata yapmadıysak, ekrana pırıl pırıl bir arayüz penceresi zıpladığını görmemiz gerekiyor. Çok güzel, değil mi? Burada komutumuzu sadece **`Tk()`** şeklinde kullanabilmemiz, yukarıda Tkinter modülünü içe aktarma şeklimizden kaynaklanıyor. Yani Tkinter modülünü **`from Tkinter import *`** şeklinde içe aktardığımız için komutu **`Tk()`** şeklinde yazmamız yeterli oluyor. Eğer Tkinter modülünü **`import Tkinter`** şeklinde içe aktarsaydık, sadece **`Tk()`** dememiz yeterli olmayacaktı. O zaman **`Tk()`** yerine **`Tkinter.Tk()`** dememiz gerekirdi. İsterseniz modülü **`import Tkinter`** şeklinde içe aktarıp, sadece **`Tk()`** yazdığınızda ne tür bir hata aldığınızı kontrol edebilirsiniz. Neyse, konuyu daha fazla dağıtmadan yolumuza devam edelim…

Gördüğünüz gibi **`Tk()`** komutuyla ortaya çıkan pencere, bir pencerenin sahip olması gereken bütün temel özelliklere sahip. Yani pencerenin sağ üst köşesinde pencereyi kapatmaya yarayan bir çarpı işareti, onun solunda pencereyi büyütüp küçültmemizi sağlayan karecik ve en solda da pencereyi görev çubuğuna indirmemizi sağlayan işaret bütün işlevselliğiyle karşımızda duruyor. Ayrıca fark edeceğiniz gibi bu pencereyi istediğimiz gibi boyutlandırmamız da mümkün.

Bu komutları hangi platformda verdiğinize bağlı olarak, pencerenin görünüşü farklı olacaktır. Yani bu komutları Windows'ta verdiyseniz, Windows'un renk ve şekil şemasına uygun bir pencere oluşacaktır. Eğer Gnome kullanıyorsanız, pencerenin şekli şemaili, Gnome'nin renk ve şekil şemasına uygun olarak KDE'dekinden farklı olacaktır...

Yukarıda verdiğimiz `Tk()` komutuyla aslında Tkinter modülü içindeki `Tk` adlı sınıfı çağırmış olduk. Bu sınıfın neye benzediğini merak eden arkadaşlar **/usr/lib/python2.7/lib-tk/** klasörü içindeki **Tkinter.py** modülü içinde `class Tk` satırını arayarak bunun nasıl bir şey olduğuna bakabilir. Bu arada herhangi bir hayal kırıklığı yaşamak istemiyorsanız, yukarıdaki `from Tkinter import *` ve `Tk()` komutlarını verirken büyük-küçük harfe dikkat etmenizi tavsiye ederim. Çünkü "*Tkinter*" ile "*tkinter*" aynı şeyler değildir...

Eğer bir arayüz oluşturacaksak sürekli komut satırında çalışamayız: Mutlaka yukarıda verdiğimiz komutları bir yere kaydetmemiz gerekir. Bunun için hemen boş bir belge açıp içine şu satırları ekleyelim:

```
from Tkinter import *
Tk()
```

Eğer bu dosyayı bu şekilde kaydeder ve çalıştırmayı denersek açılmasını beklediğimiz pencere açılmayacaktır. Burada komut satırından farklı olarak ilave bir satır daha eklememiz gerekiyor. Yani kodumuzun görünümü şu şekilde olmalı:

```
from Tkinter import *
Tk()
mainloop()
```

Belgemizin içine yukarıdaki kodları yazdıktan sonra bunu **.py** uzantılı bir dosya olarak kaydediyoruz ve normal Python dosyalarını nasıl çalıştırıyorsak öyle çalıştırıyoruz.

İsterseniz yukarıdaki kodu biraz derleyip toparlayalım. Çünkü bu şekilde kodlarımız çok çaresiz görünüyor. Yukarıdaki kodlar aslında bir program içinde şu şekilde görünecektir:

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

```
from Tkinter import *
pencere = Tk()
mainloop()
```

Bu kodlar içinde gördüğümüz tek yenilik en sondaki **`mainloop()`** satırı. Bu satır program içinde bir döngü oluşturarak, kendisinden önce gelen kodlarla belirlenen özelliklere göre bir pencere oluşturulmasını sağlıyor. Bu satırla oluşturulan döngü sayesinde de oluşturulan pencere (aksi belirtilmedikçe veya kullanıcı pencereyi kapatmadıkça) ekranda hep açık kalabiliyor. Eğer bu son satırı yazmazsak, penceremiz yine de oluşur, ama biz pencereyi ekranda göremeyiz! Bu arada buraya küçük bir not düşelim. Eğer yazdığınız programı Python'la birlikte gelen bir metin düzenleyici olan IDLE içinden çalıştırıyorsanız, yazdığınız programın son satırına **`mainloop()`** ifadesini yerleştirmeseniz bile programınız hatasız bir şekilde çalışacaktır. Neden? Çünkü IDLE da Tkinter ile yazılmış bir uygulamadır. Dolayısıyla IDLE'nin de kendisine ait bir **`mainloop()`**, satırı vardır. Yani siz kendi programınız içine **`mainloop()`**'u koymasanız bile IDLE'nin kendi **`mainloop()`**'u sizin programınızın da çalışmasını sağlayacaktır. Ancak mainloop() satırını koymadığınız halde programınız IDLE içinden çalışsa da başka yerde çalışmayacaktır. O yüzden bu **`mainloop()`** satırını mutlaka yerine koymalıyız...

Dilerseniz kodlarımızın derli toplu hale getirilmiş biçimine şöyle bir bakalım.

İlk satırı anlatmaya gerek yok. Bunun ne olduğunu zaten biliyorsunuz. Ayrıca zaten bildiğiniz gibi, bu satır sadece GNU/Linux kullanıcılarına yöneliktir. Windows kullanıcıları bu satırı yazmasalar da olur.

İkinci satırımız, yine bildiğiniz gibi, programımızı UTF-8 ile kodlamamızı sağlıyor. Windows kullanıcılarının **UTF-8** yerine **cp1254** kullanması gerekebilir...

Üçüncü satırda Tkinter modülünü programımızın içine aktarıyoruz. Yani import ediyoruz.

Sonraki satırda, daha önce komut satırında kullandığımız **`Tk()`** sınıfını bir değişkene atadık. Nesne Tabanlı Programlama konusundan hatırlayacağınız gibi, bir sınıfı değişkene atama işlemine Python'da instantiation, yani örnekleme adı veriliyordu. İşte biz de burada aslında Tkinter modülü içindeki **`Tk()`** sınıfını örneklemiş olduk...

Son satırdaki **`mainloop()`** ise yazdığımız pencerenin ekranda görünmesini sağlıyor.

Daha önceki derslerimizde Nesne Tabanlı Programlama konusunu işlerken, sınıflı yapıların genellikle arayüz programlamada tercih edildiğini söylemiştik. İsterseniz yukarıda yazdığımız minik kodun sınıflar kullanılarak nasıl yazılabileceğine bakalım:

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

from Tkinter import *

class Uygulama(object):
    def __init__(self):
        pass

pencere = Tk()
mainloop()
```

Tabii yukarıdaki sınıf örneğinin pek bir anlamı yok. Yaptığı tek şey, ekrana boş bir pencere getirmektir. Üstelik yukarıdaki örnek, kod yapısı olarak mükemmel olmaktan uzak bir iskeletten ibarettir. Bütün eksikliklerine rağmen yukarıdaki örnek bize bir Tkinter programının sınıflı bir yapı içinde temel olarak nasıl görüneceğini gösteriyor. Biz sayfalar ilerledikçe bu sınıfımızı daha iyi ve standart bir hale getireceğiz. Bu arada sınıfımızı oluştururken, Python'un yeni sınıf yapısına göre **`object`** adlı sınıfı miras aldığımıza dikkat edin.

Şimdiye kadar Python'da Tkinter yardımıyla boş bir pencere oluşturmayı öğrendik. Bu heyecan içinde bu kodları pek çok kez çalıştırıp pek çok defa boş bir pencere oluşmasını izlemiş olabilirsiniz. Ama bir süre sonra boş pencere sizi sıkmaya başlayacaktır. O yüzden, gelin isterseniz şimdi de bu boş pencereyi nasıl dolduracağımızı öğrenelim. Hemen yeni bir dosya açarak içine şu satırları ekliyoruz:

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

from Tkinter import *

pencere = Tk()
```

İlk iki satırın ne olduğunu zaten biliyorsunuz, o yüzden açıklamaya gerek yok. Ondan sonra gelen satırda da bildiğiniz gibi Tkinter modülünü çağırdık. Bir alt satırda ise yaptığımız şey daha önce gördüğümüz **`Tk()`** komutuna bir

ad vermekten, yani daha teknik bir dille ifade etmek gerekirse **`Tk()`** sınıfını örneklemekten ibaret. Biz örnekleme (instantiation) işlemi için burada pencere adını tercih ettik. Tabii ki siz isterseniz başka bir isim de kullanabilirsiniz. Hatta hiç isim vermeseniz de olur. Ama kullanım kolaylığı ve performans açısından isimlendirmenin zararını değil, yararını görürsünüz. Şimdi kodlamaya kaldığımız yerden devam ediyoruz:

```
etiket = Label(text="Elveda Zalim Dünya!")
etiket.pack()

mainloop()
```

Burada **`etiket`** adlı yeni bir değişken oluşturduk. Her zamanki gibi bu **`etiket`** adı zorunlu bir isimlendirme değil. İstediğiniz ismi kullanmakta serbestsiniz. Ancak değişkenin ismi önemli olmasa da, içeriği önemlidir. **`pencere = Tk()`** komutu yardımıyla en başta oluşturduğumuz pencereyi bir kutu olarak düşünürsek, **`etiket`** adıyla tanımladığımız değişken de bu kutu üzerine yapıştırılacak bir etiket olarak düşünülebilir. Bahsettiğimiz bu kutunun üzerine etiketini yapıştırmak için **`Label`** adlı özel bir işlevden faydalanıyoruz. Daha sonra bu **`Label`** ifadesine parametre olarak bir metin işliyoruz. Metnimiz "*Elveda Zalim Dünya!*". Metnimizin adı ise **`text`**. **`Label`** ifadesinde olduğu gibi **`text`** ifadesi de aklımıza kazımamız gereken ifadelerden birisi. Bunu bu şekilde öğrenmemiz gerekiyor. Kendi kendinize bazı denemeler yaparak bu satırda neleri değiştirip neleri değiştiremeyeceğinizi incelemenizi tavsiye ederim...

Bir alt satırda **`etiket.pack()`** ifadesini görüyoruz. Bu satır, yukarıdaki satırın işlevli hale gelmesini sağlıyor. Yani kutu üzerine etiketi yapıştırdıktan sonra bunu alıp kargoya vermemize yarıyor. Eğer bu satırı kullanmazsak bir güzel hazırlayıp etiketlediğimiz kutu kargoya verilmemiş olacağı için kullanıcıya ulaşmayacaktır.

En sondaki **`mainloop()`** ifadesini ise zaten tanıyorsunuz: Yukarıda yazdığımız bütün kodların döngü halinde işletilerek bir pencere şeklinde sunulmasını sağlıyor.

Kodlarımızın son hali şöyle olmalı:

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

from Tkinter import *
```

```
pencere = Tk()

etiket = Label(text="Elveda Dünya!")
etiket.pack()

mainloop()
```

Gördüğünüz gibi yine gayet şık, içinde "*Elveda Dünya!*" yazan bir pencere elde ettik. Alıştırma olsun diye, yukarıda yazdığımız kodlara çok benzeyen başka bir örnek daha yapalım:

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

from Tkinter import *

pencere = Tk()

etiket = Label(text = "Resimler klasörünü silmek \
istediğinizden emin misiniz?")

etiket.pack()

mainloop()
```

Şimdi isterseniz yukarıdaki kodları sınıflı yapı içinde nasıl gösterebileceğimize bakalım:

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

from Tkinter import *

class Uygulama(object):
    def __init__(self):
        self.etiket = Label(text="Dosyayı silmek \
istediğinize emin misiniz?")

        self.etiket.pack()

pencere = Tk()
uyg = Uygulama()
mainloop()
```

Sınıfsız yapıdaki kodların sınıflı yapı içinde nasıl değişikliklere uğradığına dikkat edin. **`self`**'leri nasıl kullandığımızı inceleyin. Burada **`pencere = Tk()`** satırıyla

Tk() sınıfını örnekledikten sonra **uyg = Uygulama()** satırıyla da, yukarıdaki kodların içinde yazdığımız **Uygulama** adlı sınıfımızı örnekliyoruz. En son satıra da **mainloop()** ifadesini yerleştirmeyi unutmuyoruz. Aksi halde oluşturduğumuz pencere ekranda görünmeyecektir...

Yukarıdaki sınıflı kodu isterseniz şu şekilde de yazabilirsiniz:

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

from Tkinter import *

class Uygulama(object):
    def __init__(self):
        self.araclar()

    def araclar(self):
        self.etiket = Label(text="Dosyayı silmek \
istediğinize emin misiniz?")

        self.etiket.pack()

pencere = Tk()
uyg = Uygulama()
mainloop()
```

Gördüğünüz gibi burada **etiket** öğesini ayrı bir fonksiyon olarak belirledik. Programınız büyüyüp birbirinden farklı öğeler içermeye başladığı zaman bu öğeleri yukarıdaki şekilde farklı bölümlere ayırmak, kodlarınızı daha düzenli bir hale getirecek, ileride kodlarınızın bakımını yapmayı kolaylaştıracaktır.

Eğer bu sınıflı kodları anlamakta zorluk çekiyorsanız, Nesne Tabanlı Programlama konusunu gözden geçirebilirsiniz. Bu arada hatırlatalım, Tkinter'i kullanmak için sınıflı yapıları bilmek zorunda değilsiniz. Ben de zaten bu sayfalarda sınıflı ve sınıfsız kullanımları bir arada vereceğim. Eğer sınıflı yapıyı anlamazsanız, sadece sınıfsız örneklerle de ilerleyebilirsiniz. Ama yine de benim size tavsiyem, sınıfları öğrenmeniz yönünde olacaktır. Çünkü internet üzerinde belli bir seviyeden sonra bu sınıflı yapılar hep karşınıza çıkacaktır. Yani siz kullanmasanız da, sınıfları kullanmayı bilmeniz sizin için faydalı olacaktır. Bu arada dikkat ederseniz, yukarıda verdiğimiz sınıflı son iki kod, daha önce verdiğimiz sınıflı koddan daha yetkin. Dediğimiz gibi yeni özellikler öğrendikçe, kodlarımızı adım adım standartlara daha yakın hale getireceğiz.

Bu derste Tkinter'i kullanarak basit bir pencereyi nasıl oluşturabileceğimizi öğrendik. Örneklerimizi hem sınıflı hem de sınıfsız kodlarla gösterdik. Tabii ki sınıflı kodlar bu tür küçük örneklerde çok lüzumlu değildir. Ama büyük bir projede çalışırken sınıflı yapı size esneklik ve vakit kazandıracak, kodlarınızın bakımını daha kolay yapmanızı sağlayacaktır.

Bu açıklamaları da yaptığımıza göre başka bir bölüme geçebiliriz.

Pencere Başlığı

Bildiğiniz gibi, gündelik yaşamda karşımıza çıkan pencerelerde genellikle bir başlık olur. Mesela; bilgisayarınız size bir hata mesajı gösterirken pencerenin tepesinde "*Hata*" ya da "*Error*" yazdığını görürsünüz. Eğer istersek biz de oluşturduğumuz pencerelerin tepesine böyle ifadeler yerleştirebiliriz. İşte bu işi yapmak, yani penceremizin başlığını değiştirmek için `title()` adlı metottan yararlanmamız gerekiyor. Yani bunun için yazdığımız kodlara şuna benzer bir satır eklemeliyiz:

```
pencere.title("Başlık")
```

Hemen bu özelliği bir örnekle gösterelim ki bu bilgi kafamızda somutlaşsın:

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

from Tkinter import *

pencere = Tk()

baslik = pencere.title("Hiç bir işe yaramayan \
bir pencereyim ben...")

etiket= Label(text="...ama en azından bir başlığım var.")
etiket.pack()

mainloop()
```

Gördüğünüz gibi pencereye başlık eklemek son derece kolay. Tek bir satırla işi halledebiliyoruz. Üstelik herhangi bir **paketleme** işlemi yapmamız da gerekmiyor.

Dilerseniz şimdi yukarıdaki kodları sınıflı yapı içinde gösterelim:

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

from Tkinter import *

class Uygulama(object):
    def __init__(self):
        self.etiket = Label(text="Sabit diskiniz siliniyor...")
        self.etiket.pack()

        self.baslik = pencere.title("Çok Önemli Uyarı!")

pencere = Tk()
uyg = Uygulama()
mainloop()
```

Bu kodlar içindeki tek yenilik, **`self.baslik = ...`** satırıdır. Geri kalan kısımları zaten önceki örneklerden de biliyorsunuz.

Pencerelerle ilgili daha karmaşık özelliklere geçmeden önce isterseniz burada bir durup şimdiye kadar öğrendiklerimizi çok kısa bir şekilde özetleyelim:

Tkinter'le arayüz tasarımı yapabilmek için öncelikle **`from Tkinter import \*`** diyerek gerekli modülü içe aktarmamız gerekiyor.

Ardından **`Tk()`** yardımıyla boş bir pencere oluşturuyoruz. Kullanım kolaylığı ve performans açısından **`Tk()`** ifadesini isimlendirerek bir değişkene atamayı (yani **`Tk()`** sınıfını örneklemeyi) tercih edebilirsiniz. Örneğin; **`uygulama = Tk()`**. Performans açısından sınıflarımızı örneklemek gerekir, çünkü Nesne Tabanlı Programlama adlı dersten hatırlayacağınız gibi, örneklenmeyen sınıflar **çöp toplama** (*garbage collection*) adı verilen bir sürece tabi tutulacaktır.

Bundan sonra pencere içine yazmak istediğimiz ifadeyi içeren bir etiket hazırlamamız gerekiyor. Bu etiketi alıp **`Tk()`** isimli kutunun üzerine yapıştıracağız. Bu iş için bize **`Label()`** adlı araç yardımcı olacak. Bu fonksiyonun parametresi olarak yazacağımız **`text`** adlı değişkene istediğimiz metni girebiliriz. Mesela; **`oyun = Label(text="Bilgisayarınıza virüs bulaşmış!")`**

Etiketimizi hazırladıktan sonra paketleyip kargoya vermemiz gerekiyor. Grafik arayüzün kullanıcıya ulaşması için bu gerekli. Bunun için **`xxx.pack()`** ifadesini

kullanacağız. Buradaki **xxx** yerine, bir üst satırda hazırladığımız etiket için kullandığımız ismi yazacağız. Mesela; **oyun.pack()**

Bütün bu yazdığımız kodların işlevli hale gelebilmesi için ise en sona mainloop() ifadesini yerleştirmemiz gerekiyor.

Eğer hazırladığımız pencerenin bir de başlığı olsun istiyorsak **title()** adlı metottan yararlanıyoruz.

Şu ana kadar yapabildiğimiz tek şey bir pencere oluşturup içine bir metin eklemek ve pencerede basitçe bir başlık oluşturmak. Ama tabii ki bir süre sonra bu da bizi sıkacaktır. Hem sadece bu özellikleri öğrenerek heyecan verici arayüzler hazırlamamız pek mümkün değil. En fazla, arkadaşlarımıza ufak tefek şakalar yapabiliriz bu öğrendiklerimizle!

Yeri gelmişken, hazırladığımız programı, simgesi üzerine çift tıklayarak nasıl çalıştıracağımızı görelim şimdi. Gerçi daha önceki bölümlerden hatırlıyoruz bu işin nasıl yapılacağını, ama biz yine de tekrar hatırlatalım:

GNU/Linux kullanıcıları:

.py uzantılı dosyamızı hazırlarken ilk satıra **#!/usr/bin/env python** ifadesini mutlaka yazıyoruz.

Kaydettiğimiz **.py** uzantılı dosyaya sağ tıklayarak "**özellikler**"e giriyoruz.

Burada izinler sekmesinde **çalıştırılabilir** ifadesinin yanındaki kutucuğu işaretliyoruz.

Tamam diyerek çıkıyoruz.

Windows kullanıcıları:

Windows kullanıcıları **.py** uzantılı dosyaya çift tıklayarak programı çalıştırabilir. Ancak bu şekilde arkada bir de siyah DOS ekranı açılacaktır.

O DOS ekranının açılmaması için, kodlarımızı barındıran dosyamızı **.py** uzantısı ile değil **.pyw** uzantısı ile kaydediyoruz.

Dosya uzantısını **.pyw** olarak belirledikten sonra dosya üzerine çift tıklayarak Tkinter programınızı çalıştırabilirsiniz.

Windows kullanıcıları dosya kodlaması için **#-*-coding:cp1254** veya **#-*-coding:utf-8 -*-** satırlarını kullanabilir. Böylece artık programımızın simgesi üzerine çift tıklayarak arayüzümüzü çalıştırabiliriz.

Renkler

Dikkat ettiyseniz şimdiye dek oluşturduğumuz pencerelerde yazdığımız yazılar hep siyah renkte. Tabii ki, siz oluşturduğunuz pencereler içindeki metinlerin her daim siyah olmasını istemiyor olabilirsiniz. Öntanımlı renkleri değiştirmek sizin en doğal hakkınız. Tkinter'le çalışırken renklerle oynamak oldukça basittir. Renk işlemleri Tkinter'de birtakım ifadeler vasıtasıyla hallediliyor. Python dilinde bu ifadelere **seçenek** (*option*) adı veriliyor. Mesela; daha önce öğrendiğimiz ve etiket içine metin yerleştirmemizi sağlayan text ifadesi de bir seçenektir. Şimdi öğreneceğimiz seçeneklerin kullanımı da tıpkı bu **`text`** seçeneğinin kullanımına benzer.

fg Seçeneği

Diyelim ki pencere içine yazdığımız bir metni, daha fazla dikkat çekmesi için kırmızı renkle yazmak istiyoruz. İşte bu işlem için kullanmamız gereken şey fg seçeneği. Bu ifade İngilizce'deki **foreground** (*önplan, önalan*) kelimesinin kısaltması oluyor. Hemen bir örnek verelim:

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

from Tkinter import *

pencere = Tk()
pencere.title("Hata!")

etiket = Label(text = "Hata: Bilinmeyen Hata 404", fg="red")
etiket.pack()

mainloop()
```

Gördüğünüz gibi yaptığımız tek şey, **`fg`** seçeneğini etiketin içine yerleştirmek. Burada kırmızı renk için kullandığımız kelime, İngilizce'de **kırmızı** anlamına gelen **red** sözcüğü... Yani insan dilinde kullanılan renkleri doğrudan Tkinter'de de kullanabiliyoruz. Tabii bu noktada biraz İngilizce bilmek işinizi bir hayli kolaylaştıracaktır.

Tkinter'de kullanabileceğimiz, İngilizce renk adlarından birkaç tanesini hemen sıralayalım:

red = kırmızı

white = beyaz

black = siyah

yellow = sarı

blue = mavi

brown = kahverengi

green = yeşil

pink = pembe

Tkinter'de kullanılabilecek renk adları bunlarla sınırlı değil. Eğer mevcut renk listesini görmek isterseniz, şu adrese bir bakabilirsiniz: *http://www.tcl.tk/man/tcl8.3/TkCmd/colors.htm*. Tabii o listede sıralanan renkler arasından istediğinizi bulup çıkarmak pek kolay değil. Aynı zamanda renklerin kendisini de gösteren bir liste herhalde daha kullanışlı olurdu bizim için. Neyse ki çaresiz değiliz. Tkinter bize, renk adlarının yanısıra renk kodlarını kullanarak da yazılarımızı renklendirebilme imkanı veriyor. Renk kodlarını bulmak için şu adresten yararlanabilirsiniz: *http://html-color-codes.info* Buranın avantajı, renklerin kendisini de doğrudan görebiliyor olmamız. Mesela; orada beğendiğimiz herhangi bir rengin üzerine tıkladığımızda, alttaki kutucukta o rengin kodu görünecektir. Diyelim ki renk paletinden bir renk seçip üzerine tıkladık ve alttaki kutucukta şu kod belirdi: **`#610B0B`**. Bu kodu kopyalayıp kendi yazdığımız program içinde gerekli yere yapıştırabiliriz:

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

from Tkinter import *

pencere = Tk()
pencere.title("Hata!")

etiket = Label(text = "Hata: Bilinmeyen Hata 404", fg="#610B0B")
etiket.pack()

mainloop()
```

Doğrudan renk adları yerine renk kodlarını kullanmak daha kesin sonuç verecektir. Ayrıca renk kodları size daha geniş bir renk yelpazesi sunacağı için epey işinize yarayacaktır.

Son olarak, yukarıdaki kodu sınıflı yapı içinde kullanalım:

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

from Tkinter import *

class Uygulama(object):
    def __init__(self):
        self.etiket = Label(text="Hello",fg = "red")
        self.etiket.pack()

pencere = Tk()
uyg = Uygulama()
mainloop()
```

Veya öğeleri ayrı bir fonksiyon içinde belirtmek istersek...

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

from Tkinter import *

class Uygulama(object):
    def __init__(self):
        self.araclar()

    def araclar(self):
        self.etiket = Label(text="Hata: Bilinmeyen Hata 404",
                            fg="#610B0B")
        self.etiket.pack()

pencere = Tk()
uyg = Uygulama()
mainloop()
```

Gördüğünüz gibi, bir önceki sınıflı örnekten pek farkı yok. Tek yenilik **`fg`** seçeneğinin de kodlar arasına eklenmiş olması.

BG SEÇENEĞİ

Bu da İngilizce'deki **background** (*arkaplan, arkaalan*) kelimesinin kısaltması. Adından da anlaşılacağı gibi pencereye yazdığımız metnin arkaplan rengini değiştirmeye yarıyor. Kullanımı şöyle:

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

from Tkinter import *

pencere = Tk()
pencere.title("Hata!")

etiket = Label(text = "Hata: Bilinmeyen Hata 404",
               bg="blue")
etiket.pack()

mainloop()
```

Yukarıda verdiğimiz renkleri bu seçenek için de kullanabilirsiniz. Ayrıca bu seçenek bir önceki `fg` seçeneğiyle birlikte de kullanılabilir:

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

from Tkinter import *

pencere = Tk()
pencere.title("Hata!")

etiket = Label(text = "Hata monitörle sandalyenin tam arasında!",
               fg="white",
               bg="black")
etiket.pack()

mainloop()
```

Son örnekte renk kodları yerine renk adlarını kullandığımıza dikkat edin.

Yazı Tipleri (Fonts)

Tkinter, bize renklerin yanısıra yazı tipleriyle de oynama imkânı veriyor. Hazırladığımız pencerelerdeki yazı tiplerini değiştirmek için, tıpkı renklerde olduğu gibi, yine bazı seçeneklerden faydalanacağız. Burada kullanacağımız seçeneğin adı font. Bunu kullanmak için kodlarımız arasına şuna benzer bir seçenek eklememiz gerekiyor:

```
font= "Helvetica 14 bold"
```

Burada tahmin edebileceğiniz gibi, "*Helvetica*" yazı tipini; "*14*" yazı boyutunu; "*bold*" ise yazının kalın olacağını gösteriyor. Örnek bir kullanım şöyledir:

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

from Tkinter import *
pencere = Tk()
etiket = Label(text="Merhaba Dostlar!", font="Times 15 italic")
etiket.pack()
mainloop()
```

Burada yazı tipi olarak "*Times*"; yazı boyutu olarak "*15*"; yazı biçimi olarak ise "*italic*", yani "*yatık*" biçim seçtik. **`font`** seçeneği ile birlikte kullanabileceğimiz ifadeler şunlardır:

italic = yatık

underline = altı çizili

bold = kalın

overstrike = kelimenin üstü çizili

Yazı tipi olarak neleri kullanabileceğinizi, daha doğrusu kendi sisteminizde hangi yazıtiplerinin kurulu olduğunu ise şöyle öğrenebilirsiniz:

```
from Tkinter import *
import tkFont

pencere = Tk()
yazitipleri = list(tkFont.families() )
yazitipleri.sort()
for i in yazitipleri:
    print i
```

Bu listede birden fazla kelimeden oluşan yazı tiplerini gösterirken kelimeleri birleşik yazmamız gerekiyor. Mesela "DejaVu Sans"ı seçmek için "DejaVuSans" yazmamız lazım...

Metin Biçimlendirme

Daha önceki yazılarımızda öğrendiğimiz metin biçimlemeye yarayan işaretleri Tkinter'de de kullanabiliyoruz. Yani mesela şöyle bir şey yapabiliyoruz:

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

from Tkinter import *
pencere = Tk()
etiket = Label(text="Merhaba Dostlar!\n\tNasılsınız?",
               font="DejaVuSans 15 italic")
etiket.pack()
mainloop()
```

Gördüğünüz gibi, daha önceki yazılarımızda öğrendiğimiz **\n** ve **\t** kaçış dizilerini burada da kullanabiliyoruz.

İmleçler

Arayüzlerinizi oluştururken farklı imleç şekilleri kullanmak da isteyebilirsiniz. Bunun için kullanacağımız seçenek **cursor**. Mesela:

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

from Tkinter import *
pencere = Tk()
etiket = Label(text="Deneme 1,2,3...", cursor="bottom_side")
etiket.pack()
mainloop()
```

Burada, imleci pencere üzerine getirdiğimizde imlecin ters ok şekli aldığını görürüz. Kullanılabilecek imleç isimleri için şu sayfaya bakabilirsiniz: *http://www.dil.univ-mrs.fr/~garreta/PythonBBSG/docs/Tkinter_ref.pdf*

Burada 10. ve 11. sayfalarda hoşunuza gidebilecek imleç isimlerini seçebilirsiniz. Bunun dışında yararlanabileceğiniz başka bir kaynak da şudur: *http://www.tcl.tk/man/tcl8.4/TkCmd/cursors.htm*

Pencere Boyutu

Diyelim ki içinde sadece bir "etiket" barındıran bir pencere oluşturduk:

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

from Tkinter import *

pencere = Tk()

etiket = Label(text="Hata!")
etiket.pack()

mainloop()
```

Bu kodları çalıştırdığımızda karşımıza çıkan pencere çok küçüktür. O yüzden pencerenin tepesindeki eksi, küçük kare ve çarpı işaretleri görünmüyor. Bunları görebilmek için pencereyi elle boyutlandırmamız gerekiyor. Tabii bu her zaman arzu edilecek bir durum değil. Bu gibi durumları engelleyebilmemiz ve penceremizi istediğimiz boyutta oluşturabilmemiz için Tkinter bize gayet kullanışlı bir araç sunuyor:

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

from Tkinter import *

pencere = Tk()
pencere.geometry("100x100+15+100")

etiket = Label(text="Hata!")
etiket.pack()

mainloop()
```

Gördüğünüz gibi, burada **`pencere.geometry`** ifadesinin karşısına birtakım sayılar ekleyerek penceremizi boyutlandırıyoruz. İlk iki rakam (100x100) penceremizin 100x100 boyutunda olduğunu; son iki rakam ise (+15+100) penceremizin ekrana göre soldan sağa doğru 15. pozisyonda; yukarıdan aşağıya doğru ise 100.

pozisyonda açılacağını gösteriyor. Yani bu satır sayesinde penceremizin hem boyutunu hem de konumunu değiştirmiş oluyoruz. Bu dört rakamı değiştirerek kendi kendinize denemeler yapmanızı tavsiye ederim.

Şimdi son bir özellikten bahsedip bu konuyu kapatalım. Gördüğünüz gibi oluşturduğumuz bir pencere ek bir çabaya gerek kalmadan bir pencerenin sahip olabileceği bütün temel nitelikleri taşıyor. Buna pencerenin kullanıcı tarafından serbestçe boyutlandırılabilmesi de dahil. Ancak bazı uygulamalarda bu özellik anlamsız olacağı için (mesela hesap makinelerinde) kullanıcının pencereyi boyutlandırmasına engel olmak isteyebilirsiniz. Bu işi yapmak için şu kodu kullanıyoruz:

```
pencere.resizable(width=FALSE, height=FALSE)
```

Örnek bir uygulama şöyle olacaktır:

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

from Tkinter import *

pencere = Tk()
pencere.resizable(width=FALSE, height=FALSE)

etiket = Label(text="Deneme 1,2,3...", cursor="bottom_side")
etiket.pack()

mainloop()
```

Gördüğünüz gibi bu kodlar çalıştırıldığında ortaya çıkan pencere hiçbir şekilde boyutlandırma kabul etmiyor...

Şimdi isterseniz yukarıda verdiğimiz kodları sınıflı yapı içinde nasıl kullanacağımızı görelim:

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

from Tkinter import *

class Uygulama(object):
    def __init__(self):
        self.etiket = Label(text="Hata!")
        self.etiket.pack()
```

```
pencere = Tk()
pencere.resizable(width=FALSE, height=FALSE)
uyg = Uygulama()
mainloop()
```

Böylelikle Arayüz Tasarımı konusunun Temel Bilgiler kısmını bitirmiş oluyoruz. Bir sonraki bölümde Tkinter'de Pencere Araçlarını (*widgets*) kullanmayı öğreneceğiz. Ama isterseniz yeni bölüme geçmeden önce şimdiye kadar öğrendiğimiz hemen hemen bütün konuları kapsayan bir örnek yapalım.

Tekrar

Şimdiye kadar Tkinter'in özelliklerine ve nasıl kullanılacağına ilişkin pek çok konu işledik. Yani artık temel olarak Tkinter'in neye benzediğini ve bununla neler yapabileceğimizi az çok biliyoruz. Temel Bilgiler kısmını geride bırakıp yeni bir bölüme geçmeden önce, şimdiye kadar öğrendiklerimizi bir araya getiren bir örnek yapalım. Böylece öğrendiklerimizi test etme imkanı bulmuş olacağız. Bu yazıda vereceğim örneği olabildiğince ayrıntılı bir şekilde açıklayacağım. Eğer aklınıza takılan bir yer olursa bana nasıl ulaşacağınızı biliyorsunuz...

Şimdi örneğimize geçelim. Amacımız; 1 ile 100 arasında 6 tane rastgele sayı üretmek. Yalnız bu 6 sayının her biri benzersiz olacak. Yani 6 sayı içinde her sayı tek bir defa geçecek. Önce kodlarımızın tamamını verelim:

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

from Tkinter import *
import random

liste = []

pencere = Tk()
pencere.geometry("200x50+600+460")

etiket = Label(fg="white", bg="#61380B",
               font="Helvetica 12 bold")
etiket.pack()

for i in range(6):
    while len(liste) != 6:
        a = random.randint(1,100)
```

```
        if a not in liste:
            liste.append(a)
etiket["text"] = liste
mainloop()
```

Bu kodları her çalıştırışımızda, 1 ile 100 arasında, birbirinden farklı altı adet rastgele sayı ekranda görünecektir. Tkinter'e ilişkin kodlara geçmeden önce, isterseniz yukarıda kullandığımız, rastgele sayı üretmemizi sağlayan Python kodunu biraz açıklayalım:

Rastgele sayılar üreteceğimiz için öncelikle Python'un **`random`** adlı modülünü içe aktarıyoruz. Bu modül bu tür rastgele sayı üretme işlemlerinde kullanılır.

`random` modülü içindeki **`randint()`** metodu bize, örneğin; 1'den 100'e kadar rastgele sayılar üretme imkanı sağlar. Ancak **`random.randint(1, 100)`** dediğimiz zaman, bu kod 1'den 100'e kadar rastgele tek bir sayı üretecektir. Bizim amacımız altı adet rastgele sayı üretmek.

Amacımız altı adet sayı üretmek olduğu için öncelikle **`for i in range(6):`** satırı yardımıyla 6 kez işletilecek bir döngü oluşturuyoruz. (Bunun mantığıyla ilgili aklınızda şüpheler varsa, **`for i in range(6): print "Merhaba Dünya!"`** komutunu vererek şüphelerinizi giderebilirsiniz...) Bu döngü nihai olarak **`random.randint()`** fonksiyonunu altı kez çalıştıracaktır. Böylelikle elimizde altı adet sayı olmuş olacak. Ancak üretilen bu altı sayının hepsi birbirinden farklı olmayabilir. Yani bazı sayılar birden fazla üretilebilir. Dolayısıyla "1, 5, 5, 7, 56, 64" gibi bir diziyle baş başa kalabiliriz. Ama bizim amacımız altı sayının hepsinin birbirinden farklı olması. O yüzden bunu sağlayacak bir önlem almamız gerekiyor.

Bu bahsettiğimiz önlemi **`if a not in liste: liste.append(a)`** satırlarıyla alıyoruz. Aradaki **`while`** döngüsü başka bir işe yarıyor. Onu birazdan açıklayacağız.

`if a not in liste: liste.append(a)` satırlarıyla, "**`Eğer random.randint(1, 100)`** *komutu ile üretilen sayı liste içinde yer almıyorsa bu sayıyı listeye ekle!*" emrini veriyoruz. Yani **`random.randint`** ile her sayı üretilişinde Python'un bu sayıyı listeye eklemeden önce, listede bu sayının zaten var olup olmadığını kontrol etmesini istiyoruz. Eğer üretilen sayı listede yoksa, **`liste.append()`** komutu sayesinde sayımız listeye eklenecek, yok eğer üretilen sayı listede zaten varsa, o sayı listeye alınmayacaktır. Ancak burada bir sorun çıkıyor karşımıza. Eğer üretilen sayı

listede zaten varsa, yukarıda yazdığımız komut gereği o sayı listeye alınmayacak, bu durumda listede 6'dan az sayı kalacaktır. Bizim, üretilen sayıların aynı olması durumunda **random.randint** komutunun altıyı tamamlayana kadar tekrar tekrar sayı üretmesini sağlamamız gerekiyor.

İşte bu işlemi yukarıdaki **while** döngüsü yardımıyla yapacağız. **while len(liste) != 6:** satırı ile liste uzunluğunun 6'ya eşit olup olup olmadığını denetliyoruz. Yani Python'a şu emri veriyoruz. "*liste'nin uzunluğu 6'ya eşit olmadığı sürece* **random.randint(1,100)** *komutunu işletmeye devam et!*". Böylelikle listedeki tekrar eden sayılar yüzünden liste içeriği 6'dan küçük olduğunda, bu **while** döngüsü sayesinde Python listeyi altıya tamamlayana kadar **random.randint** komutunu çalıştırmaya devam edecektir... **if a not in liste:** satırı nedeniyle zaten listede var olan sayıları listeye ekleyemeyeceği için, benzersiz bir sayı bulana kadar uğraşacak ve neticede bize 6 adet, birbirinden farklı sayı verecektir...

Kodların Tkinter ile ilgili kısmına gelelim… Esasında yukarıdaki kodlar içinde geçen her satırı anlayabilecek durumdayız. Şimdiye kadar görmediğimiz tek biçim, **etiket["text"] =** liste satırı. Bu biçimle ilgili olarak şunu söyleyebiliriz: Tkinter'deki etiket, düğme vb. öğelerin niteliklerini, yukarıda görüldüğü şekilde, tıpkı bir sözlüğün öğelerini değiştirirmiş gibi değiştirebiliriz. Yukarıdaki yapının, sözlük öğelerini değiştirme konusunu işlerken gördüğümüz sözdiziminin aynısı olduğuna dikkat edin. Mesela; bu yapıyı kullanarak şöyle bir şey yapabiliriz. Diyelim ki programımız içinde aşağıdaki gibi bir etiket tanımladık:

```
etiket = Label(fg="white", bg="#61380B",
               font="Helvetica 12 bold")
etiket.pack()
```

Etiketin **text** seçeneğinin bulunmadığına dikkat edin. İstersek yukarıdaki satırda **text** seçeneğini **text = ""** şeklinde boş olarak da tanımlayabiliriz. Bu etiketi bu şekilde tanımladıktan sonra, programımız içinde başka bir yerde bu etiketin **text** seçeneğini, öteki özellikleriyle birlikte tek tek değiştirebiliriz. Şöyle ki:

```
etiket["text"] = "www.istihza.com"
etiket["fg"] = "red"
etiket["bg"] = "black"
etiket["font"] = "Verdana 14 italic"
```

Bu yapının Python'daki sözlüklere ne kadar benzediğine bir kez daha dikkatinizi çekmek isterim. Bu yüzden bu yapıyı öğrenmek sizin için zor olmasa gerek.

Şimdi isterseniz yukarıda verdiğimiz örneğin sınıflı yapı içinde nasıl görüneceğine bir bakalım:

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

from Tkinter import *
import random

liste = []

class Uygulama(object):
    def __init__(self):
        self.araclar()
        self.kodlar()

    def kodlar(self):
        for i in range(6):
            while len(liste) != 6:
                a = random.randint(1,100)
                if a not in liste:
                    liste.append(a)
        self.etiket["text"] = liste

    def araclar(self):
        self.etiket = Label(fg="white", bg="#61380B",
                            font="Helvetica 12 bold")
        self.etiket.pack()

pencere = Tk()
pencere.geometry("150x50+600+460")
uyg = Uygulama()
mainloop()
```

Gördüğünüz gibi, sınıflı yapı içinde, kodları ufak bölümlere ayırarak incelenmelerini kolaylaştırmış olduk. Artık örneğimizi verip açıklamalarımızı da yaptığımıza göre, gönül rahatlığıyla bir sonraki bölüme geçebiliriz.

PENCERE ARAÇLARI (WIDGETS): 1.BÖLÜM

2

Arayüz uygulamaları tabii ki sadece pencerelerden ibaret değildir. Arayüzler üzerinde, istediğimiz işlemleri yapabilmemiz için yerleştirilmiş birtakım menüler, yazılar, düğmeler, kutucuklar ve buna benzer araçlar da bulunur. İşte herhangi bir programın arayüzü üzerinde bulunan düğmeler, etiketler, sağa-sola, yukarı-aşağı kayan çubuklar, kutular, kutucuklar, menüler, vb. hepsi birden pencere araçlarını, yani widget'leri, oluşturuyor.

Aslında şimdiye kadar bu pencere araçlarından bir tanesini gördük. Bildiğimiz bir pencere aracı olarak elimizde şimdilik "Label" bulunuyor. Gelin bu "Label" adlı pencere aracına biraz daha yakından bakalım:

Label Pencere Aracı

Daha önce de söylediğimiz gibi bu kelime İngilizce'de "etiket" anlamına geliyor. Bu araç, anlamına uygun olarak pencerelerin üzerine etiket misali öğeler yapıştırmamızı sağlıyor.

Hatırlayacağınız gibi bu aracı **`Label()`** şeklinde kullanıyorduk. İsterseniz bununla ilgili olarak hemen basit bir örnek verelim:

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

from Tkinter import *

pencere = Tk()
```

```
etiket = Label(text = "Hata: Bellek Read Olamadı!")
etiket.pack()

mainloop()
```

Gördüğünüz gibi yukarıdaki kullanımda "Label" aracı bir metnin pencere üzerinde görüntülenmesini sağlıyor.

Button Pencere Aracı

Bu araç yardımıyla pencerelerimize, tıklandıklarında belli bir işlevi yerine getiren düğmeler ekleyebileceğiz. Hemen bir örnek verelim:

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

from Tkinter import *

pencere = Tk()
dugme = Button(text="TAMAM", command = pencere.quit)
dugme.pack()

mainloop()
```

Dikkat ederseniz, **`Button()`** aracının kullanımı daha önce gördüğümüz **`Label`** aracının kullanımına çok benziyor. Burada da parantez içinde bazı parametreler kullandık. **`text`** parametresini zaten biliyoruz: Kullanıcıya göstermek istediğimiz metni bu **`text`** parametresi yardımıyla belirliyoruz. Aynı parantez içinde gördüğümüz **`command`** parametresi ise, düğme üzerine tıklandığında işletilecek komutu gösteriyor. Biz burada **`pencere.quit`** komutunu vererek, düğmeye tıklandığında pencerenin kapatılmasını istedik. Dolayısıyla bir satır içinde üç yeni özellik görmüş oluyoruz:

- **`Button`**: Kullanacağımız pencere aracı.
- **`command`**: Oluşturduğumuz düğmeye tıklandığında çalıştırılacak komut
- **`xxx.quit`**: Düğmeye tıklandığında pencerenin kapatılmasını sağlayan komut.

Daha sonra gelen satırdaki ifade size tanıdık geliyor olmalı: **`dugme.pack()`**. Tıpkı daha önce **`etiket.pack()`** ifadesinde gördüğümüz gibi bu ifade de hazırladığımız pencere aracının kargoya verilmek üzere **paketlenmesini** sağlıyor.

En son satırdaki **`mainloop()`** ifadesinin ne işe yaradığını artık söylemeye bile gerek yok.

Yukarıda bahsettiğimiz command parametresi çok güzel işler yapmanızı sağlayabilir. Mesela, diyelim ki; "*Oluştur*" düğmesine basınca, bilgisayarda yeni bir dosya oluşturan bir arayüz tasarlamak istiyoruz. O halde hemen bu düşüncemizi tatbik sahasına koyalım:

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

from Tkinter import *

pencere = Tk()

def olustur():
 dosya = open("deneme.txt", "w")

dugme = Button(text = "oluştur", command=olustur)
dugme.pack()

mainloop()
```

Gördüğünüz gibi Tkinter modülünü çağırıp **`pencere = Tk()`** şeklinde penceremizi oluşturduktan sonra bir fonksiyon yazdık. Tkinter dışından bir komut çalıştırmak istediğimizde bu şekilde bir fonksiyon tanımlamamız gerekir.

Daha sonra **`Button()`** pencere aracı yardımıyla, pencereye yerleştireceğimiz düğmeyi meydana getirdik. Burada command parametresine biraz önce oluşturduğumuz fonksiyonu atayarak düğmeye basıldığında, yeni bir dosya oluşturulmasına zemin hazırladık. **`text`** parametresi yardımıyla da düğmemizin adını "*Oluştur*" olarak belirledik. Ondan sonraki satırları ise zaten artık ezbere biliyoruz.

Eğer bu kodları yazarken, yukarıdaki gibi bir fonksiyon oluşturmadan;

```
dugme = Button(text = "oluştur", command=open("deneme.txt","w"))
```

gibi bir satır oluşturursanız, *deneme.txt* adlı dosya düğmeye henüz basmadan oluşacaktır...

İsterseniz bu arayüze bir de "*Çıkış*" butonu ekleyebiliriz:

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

from Tkinter import *

pencere = Tk()

def olustur():
 dosya = open("deneme.txt", "w")

dugme = Button(text = "oluştur", command=olustur)
dugme.pack()

dugme2 = Button(text = "çıkış", command=pencere.quit)
dugme2.pack()

mainloop()
```

Burada yaptığımız şey, ikinci bir düğme oluşturmaktan ibaret. Oluşturduğumuz **`dugme2`** adlı değişken için de **`dugme2.pack`** ifadesini kullanmayı unutmuyoruz.

Düğmelerin pencere üstünde böyle alt alta görünmesini istemiyor olabilirsiniz. Herhalde yan yana duran düğmeler daha zarif görünecektir. Bunun için kodumuza şu eklemeleri yapmamız gerekiyor:

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

from Tkinter import *

pencere = Tk()

def olustur():
 dosya = open("deneme.txt", "w")

dugme = Button(text = "oluştur", command=olustur)
dugme.pack(side=LEFT)

dugme2 = Button(text = "çıkış", command=pencere.quit)
dugme2.pack(side=RIGHT)

mainloop()
```

Paketleme aşamasında düğmelerden birine **`side=LEFT`** parametresini ekleyerek o düğmeyi sola; öbürüne de **`side=RIGHT`** parametresini ekleyerek sağa yaslıyoruz.

İsterseniz gelin şu son yaptığımız örneği sınıflı yapı içinde göstererek sınıflarla ufak bir pratik daha yapmış olalım:

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

from Tkinter import *

class Uygulama(object):
 def __init__(self):
  self.dugme = Button(text="oluştur",
       command=self.olustur)

  self.dugme.pack(side=LEFT)

  self.dugme2 = Button(text="çıkış",
        command=pencere.quit)

  self.dugme2.pack(side=RIGHT)

 def olustur(self):
  self.dosya = open("deneme.txt","w")

pencere = Tk()
uyg = Uygulama()
mainloop()
```

Hatırlarsanız önceki derslerimizden birinde 1-100 arası sayılar içinden rastgele altı farklı sayı üreten bir uygulama yazmıştık. Artık **`Button`** adlı pencere aracını öğrendiğimize göre o uygulamayı birazcık daha geliştirebiliriz. Mesela; o uygulamaya bir düğme ekleyebilir, o düğmeye basıldıkça yeni sayıların üretilmesini sağlayabiliriz:

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

from Tkinter import *
import random

pencere = Tk()
pencere.geometry("300x50+600+460")

def kodlar():
 liste = []
 for i in range(6):
  while len(liste) != 6:
```

```
    a = random.randint(1,100)
    if a not in liste:
     liste.append(a)
 etiket["text"] = liste

etiket = Label(text="Sayı üretmek için düğmeye basın",
       fg="white",
       bg="#61380B",
       font="Helvetica 12 bold")

etiket.pack()

dugme = Button(text="Yeniden",command=kodlar)
dugme.pack()

mainloop()
```

Burada, programda ana işlevi gören kodları **`kodlar()`** adlı bir fonksiyon içine aldık. Çünkü **`Button`** pencere aracı içindeki **`command`** seçeneğine atayabilmemiz için elimizde bir fonksiyon olması gerekiyor. Düğmeye bastığımızda Tkinter, fonksiyon içindeki bu kodları işletecek. Düğmeye her basıldığında liste öğelerinin yenilenebilmesi için **`liste=[]`** satırını fonksiyon içine alıyoruz. Böylece düğmeye ilk kez basıldığında önce liste öğesinin içi boşaltılacak, ardından altı adet sayı üretilip etikete yazdırılacaktır. Böylelikle düğmeye her basışımız yeni bir sayı dizisi oluşturacaktır. Eğer **`liste=[]`** satırını fonksiyon dışında tanımlarsak, düğmeye ilk basışımızda altı adet sayı üretilecektir, ama ikinci kez düğmeye bastığımızda, liste hala eski öğeleri tutuyor olacağı için etiket yenilenmeyecek. Düğmeye her basışımızda eski sayılar etikete yazdırılmaya devam edecektir. Bizim istediğimiz şey ise, düğmeye her basışta yeni bir sayı setinin etikete yazdırılması. Bunun için **`liste=[]`** satırını fonksiyon içine almamız gerekiyor.

Programımızın işlevini yerine getirebilmesi için kullanıcının bir düğmeye basması gerekiyor. Bu yüzden kullanıcıya ne yapması gerektiğini söylesek iyi olur. Yani kullanıcımızı düğmeye basmaya yönlendirmemiz lazım. Bu amaçla etiketin **`text`** seçeneğine "*Sayı üretmek için düğmeye basın*" şeklinde bir uyarı yerleştirdik. Programımız ilk kez çalıştırıldığında kullanıcı öncelikle bu uyarıyı görecek, böylelikle programı kullanabilmek için ne yapması gerektiğini bilecektir. Belki bu küçük programda çok belirgin olmayabilir, ama büyük programlarda, etiketler üzerine yazdığımız yönergelerin açık ve anlaşılır olması çok önemlidir. Aksi halde kullanıcı daha programı deneyemeden bir kenara atabilir. Tabii biz böyle olmasını istemeyiz, değil mi?

Şimdi de bu kodları sınıflı yapıya çevirelim:

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

from Tkinter import *
import random

class Uygulama(object):
 def __init__(self):
  self.araclar()

 def kodlar(self):
  self.liste = []
  for i in range(6):
   while len(self.liste) != 6:
    a = random.randint(1, 100)
    if a not in self.liste:
     self.liste.append(a)
  self.etiket["text"] = self.liste

 def araclar(self):
  self.etiket = Label(text="Sayı üretmek için düğmeye basın",
       fg="white",
       bg="#61380B",
       font="Helvetica 12 bold")

  self.etiket.pack()

  self.dugme = Button(text="Yeniden", command = self.kodlar)
  self.dugme.pack()

pencere = Tk()
pencere.geometry("300x50+600+460")
uyg = Uygulama()
mainloop()
```

Bu kodlar da önceki gördüklerimizden çok farklı değil. Artık sınıflı bir Tkinter kodunun neye benzediğini gayet iyi biliyoruz.

Entry Pencere Aracı

Bu araç yardımıyla kullanıcının metin girebileceği tek satırlık bir alan oluşturacağız. Bu pencere aracının kullanımı da diğer araçların kullanımına benzer. Hemen bir örnek verelim:

```
from Tkinter import *

pencere = Tk()

giris = Entry()
giris.pack()

mainloop()
```

Her zamanki gibi ilk önce Tkinter modülünü çağırarak işe başladık. Hemen ardından da **pencere** adını verdiğimiz boş bir pencere oluşturduk. Dediğimiz gibi, **Entry()** aracının kullanımı daha önce sözünü ettiğimiz pencere araçlarının kullanımına çok benzer. Dolayısıyla **giris** adını verdiğimiz **Entry()** aracını rahatlıkla oluşturabiliyoruz. Bundan sonra yapmamız gereken şey; tabii ki bu pencere aracını paketlemek. Onu da **giris.pack()** satırı yardımıyla hallediyoruz. Son darbeyi ise **mainloop** komutuyla vuruyoruz.

Gördüğünüz gibi, kullanıcıya tek satırlık metin girme imkânı veren bir arayüz oluşturmuş olduk. İsterseniz şimdi bu pencereye birkaç düğme ekleyerek daha işlevli bir hale getirelim arayüzümüzü. Mesela pencere üzerinde programı kapatmaya yarayan bir düğme ve kullanıcının yazdığı metni silen bir düğme bulunsun:

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

from Tkinter import *

pencere = Tk()

def sil():
 giris.delete(0, END)

giris = Entry()
giris.pack()

dugme1 = Button(text = "KAPAT", command = pencere.quit)
dugme1.pack(side = LEFT)
```

```
dugme2 = Button(text = "SİL", command = sil)
dugme2.pack(side = RIGHT)

mainloop()
```

Bu kodlar içinde bize yabancı olan tek ifade **giris.delete(0, END)**. Bu komut, görünüşünden de anlaşılacağı gibi giris değişkeninin içeriğini silmeye yarıyor. Parantez içindeki **0**, **END** ifadesi, metin kutusuna girilen kelimenin en başından en sonuna kadar bütün harflerin silinmesi emrini veriyor. Eğer **0**, **END** yerine, mesela **2**, **4** gibi bir ifade koysaydık; girilen kelimenin 2. harfinden itibaren 4. harfine kadar olan kısmın silinmesi emrini vermiş olacaktık. Son bir alıştırma yaparak bu aracı da tamamlayalım:

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

from Tkinter import *

pencere = Tk()

def olustur():
 dosya = open("deneme.txt","w")
 metin = giris.get()
 dosya.write(metin)

giris = Entry()
giris.pack()

dugme = Button(text = "OLUŞTUR", command = olustur)
dugme.pack(side=LEFT)

dugme2 = Button(text = "ÇIK", command = pencere.quit)
dugme2.pack(side=RIGHT)

mainloop()
```

Burada da bize yabancı olan tek ifade **giris.get()**. Bu ifade **Entry()** pencere aracı ile kullanıcıdan aldığımız metni elimizde tutup saklamamızı sağlıyor. Burada bu ifadeyi önce **metin** adlı bir değişkene atadık. Sonra da bu metin değişkeninin içeriğini **dosya.write(metin)** komutunun yardımıyla boş bir dosyaya aktardık. Metin kutusunun içeriğini barındıran **deneme.txt** isimli dosya **/home** klasörünüzün altında veya masaüstünde oluşmuş olmalı.

Bir örnek daha yapalım...

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

from Tkinter import *
import random

pencere = Tk()

def bas():
 a = random.randint(1, 100)
 giris.delete(0, END)
 giris.insert(0, a)

giris = Entry(width=10)
giris.pack()

dugme = Button(text="bas", command=bas,
      width=2, height=0)
dugme.pack()

mainloop()
```

Gördüğünüz gibi, bu uygulama 1 ile 100 arasında rastgele sayılar seçiyor. Aslında yaptığımız işlem çok basit:

Öncelikle Python'un **`random`** adlı modülünü çağırdık. Daha önce de gördüğümüz gibi, rastgele sayılar seçerken bize bu modül yardımcı olacak. Ardından da bu rastgele sayıları oluşturup ekrana yazdırmamızı sağlayacak fonksiyonu oluşturuyoruz. Bu fonksiyonda öncelikle rastgele sayıların hangi aralıkta olacağını belirleyip bunu "*a*" adlı bir değişkene atıyoruz. Böylece rastgele sayıları ekrana yazdırmak için gereken altyapıyı oluşturmuş olduk. Şimdi bu noktada eğer bir önlem almazsak ekrana basılacak sayılar yan yana sıralanacaktır. Yani mesela diyelim ki ilk rastgele sayımız 3 olsun. Bu 3 sayısı ekrana yazıldıktan sonra ikinci rastgele sayı ekrana gelmeden önce bu ilk sayının ekrandan silinmesi gerekiyor. Bütün sayılar yan yana ekrana dizilmemeli. Bunun için kodumuza şu satırı ekliyoruz:

```
giris.delete(0, END)
```

Bu satır sayesinde, ilk sayı ekrana basıldıktan sonra ekranın 0. konumundan sonuncu konumuna kadar bütün her şeyi siliyoruz. Böylelikle ikinci gelecek sayı

için yer açmış oluyoruz. İsterseniz yukarıdaki kodları bu satır olmadan çalıştırmayı bir deneyebilirsiniz. Ondan sonra gelen satırın ne işe yaradığını anlamışsınızdır: **a** değişkenini 0. konuma yerleştiriyoruz. Fonksiyonumuzu böylece tamamlamış olduk. Daha sonra normal bir şekilde **Entry** ve **Button** adlı pencere araçları yardımıyla düğmelerimizi ve metin alanımızı oluşturuyoruz. Burada bazı yeni seçenekler dikkatiniz çekmiş olmalı: Bunlar; **width** ve **height** adlı seçenekler... **width** seçeneği yardımıyla bir pencere aracının genişliğini; "**height**" seçeneği yardımıyla ise o aracın yüksekliğini belirliyoruz.

Frame()

Frame() Tkinter içinde bulunan sınıflardan biridir. Bu sınıf aynı zamanda bir **pencere aracı** (*widget*) olarak da kullanılabilir. Bir pencere aracı olarak kullanıldığında bu sınıfın görevi, pencere içindeki öbür pencere araçlarını pencere içinde rahatça konumlandırmayı sağlamaktır. Bu araçtan, ileride **geometri yöneticilerini** (*geometry managers*) işlerken daha ayrıntılı olarak bahsedeceğiz. Bu derste **Frame()** pencere aracının temel olarak nasıl kullanıldığına dair birkaç örnek vermekle yetineceğiz. Şu örneğe bir bakalım:

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

from Tkinter import *

pencere = Tk()

etiket = Label(text="Aşağıdaki kutucuğa e.posta adresinizi yazınız!")
etiket.pack()

giris = Entry()
giris.pack()

dugme = Button(text="Gönder", command=pencere.destroy)
dugme.pack()

mainloop()
```

Burada gördüğünüz gibi "*Gönder*" düğmesi hemen üstündeki kutucuğa çok yakın duruyor. Bu arayüzün daha iyi görünmesi için bu iki pencere aracının arasını biraz açmak isteyebiliriz.

İşte **Frame()** aracı böyle bir ihtiyaç halinde işimize yarayabilir:

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

from Tkinter import *

pencere = Tk()

etiket = Label(text="Aşağıdaki kutucuğa e.posta adresinizi yazınız!")
etiket.pack()

giris = Entry()
giris.pack()

cerceve = Frame()
cerceve.pack()

dugme = Button(text="Gönder", command=pencere.destroy)
dugme.pack()

mainloop()
```

Gördüğünüz gibi **Frame()** aracını **giris** ve **dugme** adlı pencere araçlarının arasına yerleştirdik. Bu kodları çalıştırdığımızda, düğmenin azıcık da olsa aşağıya kaydığını göreceksiniz. Eğer kayma oranını artırmak isterseniz, **pad** seçeneğinden yararlanabilirsiniz. Bu seçenek, bir pencere aracının, öteki pencere araçları ile arasını açmaya yarıyor. **pad** seçeneği iki şekilde kullanılabilir: **padx** ve **pady**. Bu iki seçenekten **padx**, **x** düzlemi üzerinde işlem yaparken, **pady** ise **y** düzlemi üzerinde işlem yapıyor. Yani:

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

from Tkinter import *

pencere = Tk()

etiket = Label(text="Aşağıdaki kutucuğa e.posta adresinizi
yazınız!")
etiket.pack()

giris = Entry()
giris.pack()

cerceve = Frame()
cerceve.pack(pady=5)
```

```
dugme = Button(text="Gönder", command=pencere.destroy)
dugme.pack()

mainloop()
```

Burada sadece **pady** seçeneğini kullanmamız yeterli olacaktır. Çünkü bizim bu örnekte istediğimiz şey, iki pencere aracının arasını y düzlemi üzerinde açmak. Yani yukarıdan aşağıya doğru... Eğer **x** düzlemi üzerinde işlem yapmak isteseydik (yani soldan sağa doğru), o zaman padx seçeneğini de kullanacaktık. İsterseniz bu **padx** ve **pady** seçeneklerine bazı sayılar vererek birtakım denemeler yapabilirsiniz. Bu sayede bu seçeneklerin ne işe yaradığı daha net anlaşılacaktır. Ayrıca dikkat ederseniz, bu **padx** ve **pady** seçenekleri, daha önce gördüğümüz side seçeneğine kullanım olarak çok benzer. Hem **padx** ve **pady**, hem de **side** seçenekleri, **pack** aracının birer özelliğidir.

İsterseniz yukarıdaki kodların sınıflı yapı içinde nasıl görüneceğine de bir bakalım...

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

from Tkinter import *

class Uygulama(object):
 def __init__(self):
  self.guiPenAr()

 def guiPenAr(self):
  self.etiket = Label(text="Aşağıdaki kutucuğa e.posta \
adresinizi yazınız!")

  self.etiket.pack()

  self.giris = Entry()
  self.giris.pack()

  self.cerceve = Frame()
  self.cerceve.pack(pady=5)

  self.dugme = Button(text="Gönder",
       command=pencere.destroy)
  self.dugme.pack()

pencere = Tk()
uyg = Uygulama()
mainloop()
```

Şimdilik bu `Frame()` pencere aracını burada bırakıyoruz. Bir sonraki bölümde bu araçtan ayrıntılı olarak bahsedeceğiz.

Böylelikle Pencere Araçları konusunun ilk bölümünü bitirmiş olduk. Şimdi başka bir pencere aracı grubuna geçmeden önce çok önemli bir konuya değineceğiz: Tkinter'de **Geometri Yöneticileri** (*Geometry Managers*).

GEOMETRİ YÖNETİCİLERİ

3

Şimdiye kadar yaptığımız örneklerden de gördüğünüz gibi bir pencere aracını kullanıcılara gösterebilmemiz için bu aracı tanımlamak yeterli olmuyor. Aracı tanımlamanın yanısıra özel bir metodu da kullanmamız gerekiyor. Şu ana dek Tkinter ile hazırladığımız örneklerde pencere araçlarını kullanıcıya gösterebilmek için **`pack()`** adlı özel bir metottan yararlandık. Bu metodun kullanımına şöyle bir örnek verebiliriz:

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

from Tkinter import *

pencere = Tk()
pencere.geometry("200x100+50+100")

dolar = Label(text="Dolar",
     fg = "white",
     bg = "red",
     font="Verdana 13 bold")
dolar.pack()

avro = Label(text="Avro",
    fg = "white",
    bg = "blue",
    font="Verdana 13 bold")
avro.pack()

lira = Label(text="Lira",
```

```
    fg = "white",
    bg = "green",
    font="Verdana 13 bold")
lira.pack()

mainloop()
```

Yukarıdaki pencere araçlarını gösterebilmek için kullandığımız **`pack()`** adlı metoda Tkinter'de **geometri yöneticisi** (*geometry manager*) adı veriliyor. Anladığınız gibi geometri yöneticilerinin temel işlevi Tkinter ile oluşturulan pencere araçlarını (yani widget'leri) pencere üzerinde görünür hale getirmektir. Tkinter'de esasen üç farklı geometri yöneticisi bulunur. Bu geometri yöneticilerini şöyle sıralayabiliriz:

- **`pack()`**
- **`grid()`**
- **`place()`**

Geometri Yöneticileri bir pencere aracının konumunu belirler. Bu Geometri Yöneticileri'nin her birinin ayrı özellikleri vardır ve bunların her biri belli bir işlevi yerine getirmede daha başarılıdır. Pack ve Grid Geometri Yöneticileri genel olarak hemen her işi yapmamıza izin verir. O yüzden bu iki Yönetici gayet kullanışlıdır. Place Geometri Yöneticisi ise pencere araçlarını pencere üzerinde yerli yerine oturtma konusunda çok ustadır. O yüzden pencere araçlarını istediğimiz şekilde pencere üzerine konumlandırmak istediğimiz zaman bu işi en kolay Place Geometri Yöneticisi yardımıyla yapabiliriz.

Geometri Yöneticileri ile ilgili olarak aklımızda tutmamız gereken en önemli şey aynı pencere üzerinde birbirinden farklı Yöneticileri bir arada kullanmamaktır. Bunun ne demek olduğunu ilerde örneklerle gösterdiğimiz zaman çok daha iyi anlayacaksınız.

Bu geometri yöneticileri içinde en kolay olanı **`pack()`** adlı yöneticidir. Zaten bu sebeple şimdiye kadar gördüğümüz örneklerde de hep bu **`pack()`** metodunu kullandık. Ancak hangi geometri yöneticisini kullanırsak kullanalım, öteki yöneticilerin nasıl kullanıldığını da öğrenmemiz gerekir. Çünkü bu geometri yöneticilerinin her birinin iyi olduğu bir alan vardır. Şimdi isterseniz bu geometri yöneticilerini tek tek inceleyelim.

pack() Geometri Yöneticisi

Bir önceki bölümde de söylediğimiz gibi, **pack()** geometri yöneticisi Tkinter'deki geometri yöneticileri içinde en kolay olanıdır. Bu geometri yöneticisinin kullanımıyla ilgili basit bir örneği önceki bölümde vermiştik. İsterseniz bu örneği tekrar önümüze alalım:

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

from Tkinter import *

pencere = Tk()
pencere.geometry("200x100+50+100")

dolar = Label(text="Dolar",
     fg = "white",
     bg = "red",
     font="Verdana 13 bold")
dolar.pack()

avro = Label(text="Avro",
    fg = "white",
    bg = "blue",
    font="Verdana 13 bold")
avro.pack()

lira = Label(text="Lira",
    fg = "white",
    bg = "green",
    font="Verdana 13 bold")
lira.pack()

mainloop()
```

pack() metodunu parametresiz olarak kullandığımızda öğelerin alt alta sıralandığına dikkat edin. Ancak gördüğünüz gibi, **pack()** metodunu bu şekilde kullandığımızda etiketler pencereye orantısız olarak dağılıyor. Yani görünüş biraz özensiz duruyor. Ama eğer biz istersek bu durumu bir nebze olsun düzeltebiliriz. Bunun için ihtiyacımız olan şey, **pack()** geometri yöneticisinin **expand** adlı seçeneğidir...

expand seçeneği:

expand seçeneği, pencere araçlarının pencere üzerinde daha orantılı bir şekilde dağılmasını sağlar. Bu expand seçeneği **YES** ve **NO** olmak üzere iki farklı değer alır. Varsayılan değer **NO**'dur. Bunu yukarıdaki örnek üzerinde görelim:

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

from Tkinter import *

pencere = Tk()
pencere.geometry("200x100+50+100")

dolar = Label(text="Dolar",
     fg = "white",
     bg = "red",
     font="Verdana 13 bold")

dolar.pack(expand=YES)

avro = Label(text="Avro",
    fg = "white",
    bg = "blue",
    font="Verdana 13 bold")

avro.pack(expand=YES)

lira = Label(text="Lira",
    fg = "white",
    bg = "green",
    font="Verdana 13 bold")

lira.pack(expand=YES)

mainloop()
```

expand kelimesi İngilizce'de **genişlet**, **yay** gibi anlamlara gelir. Yukarıdaki örnekte gördüğünüz gibi **pack()** metoduna parametre olarak **expand = YES** seçeneğini vermek yoluyla öğelerin pencere üzerine dağılmasını veya yayılmasını sağlıyoruz. Eğer **expand = NO** gibi bir biçim kullanırsak, öğeler pencere üzerine yayılmayacak, yani sanki bu seçeneği hiç kullanmamışız gibi davranılacaktır.

Gördüğünüz gibi öğeler yukarıdaki şekilde birbirinden ayrı duruyor. Biz bu öğelerin birbirine değmesini de isteyebiliriz. Yani tercihimiz, tıpkı bazı ülkelerin

bayraklarında olduğu gibi, bu üç rengin şeritler halinde birbirine değecek şekilde uzanmasını sağlamak olabilir. Bu iş için kullanabileceğimiz ayrı bir seçenek var. Bu seçeneğin adı: **fill**.

fill seçeneği:

Bu fill kelimesi, İngilizce'de **doldurmak** anlamına gelir. Bu seçenek yardımıyla, araçlarımızın pencereyi doldurmasını sağlayacağız. fill seçeneği üç farklı değer alabilir: `X`, `Y` ve `BOTH`.

`X` seçeneği, pencere aracımızın, içinde bulunduğu pencereyi `x` düzlemi boyunca doldurmasını sağlar (yani soldan sağa doğru)

`Y` seçeneği, pencere aracımızın, içinde bulunduğu pencereyi `y` düzlemi boyunca doldurmasını sağlar (yani yukarıdan aşağıya doğru)

`BOTH` seçeneği, pencere aracımızın, içinde bulunduğu pencereyi `x` ve `y` düzlemi boyunca doldurmasını sağlar (both kelimesi İngilizce'de **her ikisi de** anlamına gelir)

Şimdi bu seçenekleri örnekler üzerinde görerek kafamızda somutlaştıralım:

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

from Tkinter import *

pencere = Tk()
pencere.geometry("200x100+50+100")

dolar = Label(text="Dolar",
     fg = "white",
     bg = "red",
     font="Verdana 13 bold")

dolar.pack(expand=YES, fill = X)

avro = Label(text="Avro",
    fg = "white",
    bg = "blue",
    font="Verdana 13 bold")

avro.pack(expand=YES, fill = Y)

lira = Label(text="Lira",
```

```
    fg = "white",
    bg = "green",
    font="Verdana 13 bold")

lira.pack(expand=YES, fill = BOTH)

mainloop()
```

Burada her bir öğede ayrı bir değer kullandık. Ama eğer siz isterseniz her üçü için tek bir değer kullanarak, bu değerlerin tam olarak ne yaptığını daha net görmeyi tercih edebilirsiniz.

side seçeneği:

side seçeneğini daha önce de kullanmıştık. Bu seçenek bir pencere aracını istediğimiz bir yöne yaslamamızı sağlıyor. Bu seçenek dört farklı değer alır: **TOP**, **BOTTOM**, **LEFT** ve **RIGHT**.

- **TOP**: pencere aracını üste yaslar.
- **BOTTOM**: pencere aracını dibe yaslar.
- **LEFT**: pencere aracını sola yaslar.
- **RIGHT**: pencere aracını sağa yaslar.

Ancak bu dört değer içinde en sık kullanılanlar **TOP** ve **LEFT**'tir. Eğer pencere araçlarını yan yana sıralamak istersek kısaca **LEFT** değerini kullanabiliriz. Eğer pencere araçlarını üst üste sıralamak istersek de **TOP** değerini kullanabiliriz. Şu iki örneği dikkatlice inceleyin:

LEFT ile kullanım:

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

from Tkinter import *

pencere = Tk()
dolar = Label(text="Dolar",
     fg = "white",
     bg = "red",
     font="Verdana 13 bold")

dolar.pack(side=LEFT)
```

```
avro = Label(text="Avro",
    fg = "white",
    bg = "blue",
    font="Verdana 13 bold")

avro.pack(side=LEFT)

lira = Label(text="Lira",
    fg = "white",
    bg = "green",
    font="Verdana 13 bold")

lira.pack(side=LEFT)

mainloop()
```

TOP ile kullanım:

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

from Tkinter import *

pencere = Tk()
dolar = Label(text="Dolar",
     fg = "white",
     bg = "red",
     font="Verdana 13 bold")

dolar.pack(side=TOP)

avro = Label(text="Avro",
    fg = "white",
    bg = "blue",
    font="Verdana 13 bold")

avro.pack(side=TOP)

lira = Label(text="Lira",
    fg = "white",
    bg = "green",
    font="Verdana 13 bold")

lira.pack(side=TOP)

mainloop()
```

Ayrıca istersek birkaç seçeneği bir arada kullanarak aşağıdaki şekilde bir şey de yapabiliriz:

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

from Tkinter import *

pencere = Tk()
dolar = Label(text="Dolar",
     fg = "white",
     bg = "red",
     font="Verdana 13 bold")

dolar.pack(side=TOP, expand=YES,fill=BOTH)

avro = Label(text="Avro",
    fg = "white",
    bg = "blue",
    font="Verdana 13 bold")

avro.pack(side=TOP,expand=YES, fill=BOTH)

lira = Label(text="Lira",
    fg = "white",
    bg = "green",
    font="Verdana 13 bold")

lira.pack(side=TOP,expand=YES, fill=BOTH)

mainloop()
```

Bu pencereyi elle sağa-sola veya yukarı-aşağı doğru boyutlandırın. Gördüğünüz gibi, pencereyi ne şekilde boyutlandırırsanız boyutlandırın, etiket öğeleri pencereyi doldurmaya devam ...

Elbette **`pack()`** geometri yöneticisi yalnızca etiketlerle birlikte kullanılmıyor. Öğrendiğimiz öteki pencere araçlarını da kullanarak, örneğin, şöyle bir şey yapabiliriz:

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

from Tkinter import *

pencere = Tk()
```

```
uyari = Label(pencere,text="Programdan çıkmak istediğinize \
emin misiniz?")

uyari.pack()

evet = Button(text="Evet",command=pencere.quit)
evet.pack()

hayir = Button(text="Hayır")
hayir.pack()

mainloop()
```

Burada **`pack()`** metotları içinde değişik seçeneklere farklı değerler uygulayarak istediğinize yakın bir görüntü elde edebilirsiniz. Ancak aşağıdaki gibi bir görüntü elde etmek için biraz uğraşmak gerekiyor:



Yukarıdaki gibi bir görüntü elde edebilmek için daha önce gördüğümüz **`Frame()`** adlı pencere aracından yararlanmanız gerekir. **`Frame()`** aracının nasıl kullanıldığını hatırlıyorsunuz:

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

from Tkinter import *

pencere = Tk()

yeni = Frame()
yeni.pack(side=BOTTOM)

uyari = Label(pencere,text="Programdan çıkmak istediğinize \
emin misiniz?")

uyari.pack()

evet = Button(yeni,text="Evet",command=pencere.quit)
evet.pack(side=LEFT)

hayir = Button(yeni,text="Hayır")
hayir.pack()

mainloop()
```

Burada dikkatimizi çekmesi gereken bazı çok önemli yenilikler var. Mesela yukarıda geçen şu satıra bakın:

```
uyari = Label(pencere, text="Programdan çıkmak istediğinize emin misiniz?")
```

Kalın harflerle gösterdiğimiz ifade bizim için bir yenilik sayılır. Daha önceden yazdığımız pencere araçlarında, bu pencere ifadesini kullanmamıza gerek yoktu. Çünkü yazdığımız programlarda hep sadece tek bir **taşıyıcı pencere aracı** (*container widget*) yer alıyordu. Bu taşıyıcı pencere aracı ifadesi de ne oluyor, diye sorarsanız hemen açıklayalım: Görevi başka pencere araçlarını taşımak olan pencere araçlarına Tkinter'de "taşıyıcı pencere aracı" (container widget) adı veriliyor. Mesela yazdığımız bütün programlarda **`pencere = Tk`** ifadesi ile oluşturduğumuz ana penceremiz esasında bir taşıyıcı pencere aracıdır. Yani bu pencerenin görevi pencere araçlarını tutmaktır. Aynı şekilde **`Frame()`** de bir taşıyıcı pencere aracıdır. Bu pencere aracı da, tıpkı ana penceremiz gibi, başka pencere araçlarını üzerinde taşıyabilir. Mesela yukarıdaki örnekte **`Frame()`** pencere aracı bu görevde kullanılıyor. Nasıl **`uyari`** adlı etiketi **`pencere`** adlı taşıyıcının üstüne yerleştirmişsek; **`evet`** ve **`hayir`** adlı pencere araçlarını da **`yeni`** adlı taşıyıcının üzerine yerleştiriyoruz. Bu yerleştirme işlemini, yukarıda gördüğünüz gibi pencere araçlarının ilk argümanı olarak, herhangi bir taşıyıcı adı belirtmek suretiyle gerçekleştiriyoruz. Mesela; **`evet`** adlı pencere aracını **`yeni`** adlı taşıyıcı üzerine yerleştirmişiz. Bunu **`Button(yeni, text="Evet".....")`** komutundaki ilk parametre olan **`yeni`** ifadesinden anlıyoruz. Aynı şekilde **`uyari`** adlı **`pencere`** aracını pencere adlı taşıyıcının üzerine yerleştirdiğimizi de **`Label(pencere,text="Pr...")`** komutundaki ilk parametre olan **`pencere`** ifadesinden anlıyoruz.

Peki **`Frame()`** aracını (ve pack yöneticisini) kullanarak başka neler yapabiliriz? Hemen bir örnek daha verelim:

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

from Tkinter import *

pencere = Tk()

etiket = Label(text="Aşağıdaki kutucuğa e.posta \
adresinizi yazınız!")
```

```
etiket.pack()

giris = Entry()
giris.pack()

cerceve = Frame(height=2, bd=1, relief=SUNKEN)
cerceve.pack(fill=X, pady=5, padx=5)

dugme = Button(text="Gönder", command=pencere.destroy)
dugme.pack()

mainloop()
```

Burada birkaç yeni özellik görüyoruz. Daha önce görmediğimiz özellikler **`bd`** ve **`relief`** seçenekleridir. **`bd`** İngilizce'de **kenar**, **kenarlık**, **sınır** anlamına gelen border kelimesinin kısaltması. Bu seçenek yardımıyla **`Frame()`** aracının kenarlık ölçülerini belirliyoruz. Varsayılan olarak **`Frame()`** aracının kenarlık boyutu 0'dır. Biz bunu yukarıda 1 olarak değiştirdik. Aynı satırda gördüğümüz **`height`** seçeneğini daha önce pencere boyutu ve entry konularını işlerken de görmüştük. Oradan hatırladığımıza göre, bu seçenek bir pencere aracının yüksekliğini ayarlamamızı sağlıyordu. Yukarıdaki örnekte de **`Frame()`** pencere aracının yüksekliğini 2 olarak ayarladık. **`relief`** seçeneği ise İngilizce bir kelime olarak Türkçe'de **kabartma** (*rölyef*) anlamına geliyor. Dolayısıyla bu seçenek yardımıyla pencere aracının kabartma tipini belirliyoruz. Burada altı farklı stil uygulayabiliriz:

- **`SOLID`**
- **`FLAT`**
- **`RIDGE`**
- **`SUNKEN`**
- **`GROOVE`**
- **`RAISED`**

Bu stillerin nasıl bir görüntü ortaya çıkardığını daha net görebilmek için **`height`** ve **`bd`** seçeneklerini daha yüksek değerlere ayarlamayı tercih edebilirsiniz.

Yukarıdaki örneğin sınıflı yapı içinde nasıl görüneceğine bakalım bir de:

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

from Tkinter import *

class Uygulama(object):
 def __init__(self):
  self.guiPenAr()

 def guiPenAr(self):
  self.etiket = Label(text="Aşağıdaki kutucuğa e.posta \
adresinizi yazınız!")

  self.etiket.pack()

  self.giris = Entry()
  self.giris.pack()

  self.cerceve = Frame()
  self.cerceve.pack(side=BOTTOM,
        padx=5,
        pady=5)

  self.dugme = Button(self.cerceve,
       text="Gönder",
       width=5,
       command=pencere.destroy)

  self.dugme.pack(side=LEFT,padx=5)

  self.dugme2 = Button(self.cerceve,
        text="İptal",
        width=5)

  self.dugme2.pack()

pencere = Tk()
uyg = Uygulama()
mainloop()
```

`Frame()` aracını kullanarak mesela şöyle bir şey de yapabiliriz:

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

from Tkinter import *

pencere = Tk()

etiket = Label(text="Aşağıdaki kutucuğa e.posta \
adresinizi yazınız!")
etiket.pack()

giris = Entry()
giris.pack()

cerceve = Frame()
cerceve.pack(side=BOTTOM,
    padx=5,
    pady=5)

dugme = Button(cerceve,
      text="Gönder",
      width=5,
      command=pencere.destroy)

dugme.pack(side=LEFT,padx=5)

dugme2 = Button(cerceve,
    text="İptal",
    width=5)

dugme2.pack()

mainloop()
```

Burada pencere aracı seçeneklerini nasıl kullandığımıza dikkat edin.

İsterseniz bu örneği de sınıflı olarak ifade edelim:

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

from Tkinter import *

class Uygulama(object):
 def __init__(self):
```

```
    self.guiPenAr()

  def guiPenAr(self):
    self.etiket = Label(text="Aşağıdaki kutucuğa e.posta \
adresinizi yazınız!")
    self.etiket.pack()

    self.giris = Entry()
    self.giris.pack()

    self.cerceve = Frame()
    self.cerceve.pack(side=BOTTOM,
          padx=5,
          pady=5)

    self.dugme = Button(self.cerceve,
         text="Gönder",
         width=5,
         command=pencere.destroy)

    self.dugme.pack(side=LEFT,padx=5)

    self.dugme2 = Button(self.cerceve,
          text="İptal",
          width=5)

    self.dugme2.pack()

pencere = Tk()
uyg = Uygulama()
mainloop()
```

`pack()` metodunu kullanarak çok karmaşık arayüz görüntüleri elde edebiliriz. Aslında bu metodu kullanarak yapamayacağımız şey yok gibidir. Ama bazı şeyleri bu metotla yapmaya kalkışmak bir süre sonra işkenceye dönüşebilir. O yüzden yeri geldiğinde **`pack()`** yerine başka metotları kullanmayı da bilmeliyiz. Mesela bazı işler için **`grid()`** metodu daha uygun olabilir. **`grid()`** metodunu anlatmaya geçmeden önce isterseniz **`pack()`** metodunu nerelerde kullanabileceğimizi şöyle bir sıralayalım:

- Pencere araçlarını alt alta dizmek istediğimizde,
- Pencere araçlarını yan yana dizmek istediğimizde,
- Pencere araçlarını alt alta veya yan yana dizip, bunların pencereyi (x veya y düzleminde ya da her ikisinde birden) tamamen kaplamasını istediğimizde.

Bu metodu nerelerde kullanmamızın uygun olacağını gördüğümüze göre artık başka bir geometri yöneticisini anlatmaya başlayabiliriz.

GRID() GEOMETRİ YÖNETİCİSİ

Bu geometri yöneticisi, pencere araçlarının ızgara biçiminde sıralanacağı arayüzler için uygundur. Bununla hemen basit bir örnek yapalım:

```
# -*- coding: utf-8 -*-
from Tkinter import *

dugme1 = Button(text="Ürünler")
dugme1.grid()

dugme2 = Button(text="Hizmetler")
dugme2.grid()

dugme3 = Button(text="Ulaşım")
dugme3.grid()

dugme4 = Button(text="Hakkında")
dugme4.grid()

mainloop()
```

`grid()` adlı geometri yöneticisi basitçe bu şekilde kullanılıyor. Ama tabii ki bu haliyle pek cazip görünmüyor. Buna istediğimiz şekli verebilmek için yine birtakım seçeneklerden yararlanmamız gerekecek:

row seçeneği

Bu geometri yöneticisinin en önemli iki seçeneğinden birincisi bu **`row`** seçeneğidir. Bu kelime İngilizce'de **satır** anlamına gelir. Bu seçenek yardımıyla, pencere aracının hangi satırda yer alacağını belirleyeceğiz. Bununla herhangi bir örnek yapmadan önce, isterseniz **`grid()`** geometri yöneticisinin öteki seçeneğinden de bahsedelim. Çünkü bu seçenek hep **`row`** seçeneğiyle birlikte kullanılır.

column seçeneği

column kelimesi İngilizce'de **sütun** anlamına gelir. Dolayısıyla bu seçenek yardımıyla pencere aracımızın pencere üzerinde hangi sütunda yer alacağını belirleyeceğiz. Artık bu iki seçeneği kullanarak bir örnek verebiliriz.

Aşağıdaki örneği dikkatle inceleyin:

```
# -*- coding: utf-8 -*-
from Tkinter import *

dolar = Label(text="Dolar")
dolar.grid(row=0,column=0)

avro = Label(text="Avro")
avro.grid(row=1,column=0)

lira = Label(text="Lira")
lira.grid(row=2,column=0)

dolar_g = Entry()
dolar_g.grid(row=0,column=1)

avro_g = Entry()
avro_g.grid(row=1,column=1)

lira_g = Entry()
lira_g.grid(row=2,column=1)
```

Gördüğünüz gibi bu **`grid()`** metodu yardımıyla **`pack()`** metoduna kıyasla daha **ince** ayarlamalar yapabiliyoruz. **`grid()`** metodu bize, pencere aracımızın koordinatlarını daha hassas bir şekilde belirtme imkanı sunuyor. Şimdi isterseniz ilk verdiğimiz, cazip görünmeyen örneği yeniden ele alalım. Örneğimiz şuydu:

```
# -*- coding: utf-8 -*-
from Tkinter import *

dugme1 = Button(text="Ürünler")
dugme1.grid()

dugme2 = Button(text="Hizmetler")
dugme2.grid()

dugme3 = Button(text="Ulaşım")
dugme3.grid()

dugme4 = Button(text="Hakkında")
dugme4.grid()

mainloop()
```

Şimdi yapacağımız şey, pencere üzerinde dağınık bir şekilde duran bu araçların her biri için uygun bir satır-sütun belirlemek. Hemen bakalım:

```
# -*- coding: utf-8 -*-
from Tkinter import *

dugme1 = Button(text="Ürünler")
dugme1.grid(row=0, column=0)

dugme2 = Button(text="Hizmetler")
dugme2.grid(row=0, column=1)

dugme3 = Button(text="Ulaşım")
dugme3.grid(row=1, column=0)

dugme4 = Button(text="Hakkında")
dugme4.grid(row=1,column=1)

mainloop()
```

İsterseniz daha önceden gördüğümüz **`fg`**, **`bg`** ve **`font`** gibi seçenekler yardımıyla bu düğmelerin ve üzerindeki yazıların renklerini gönlünüze göre değiştirebilirsiniz de...

Dediğimiz gibi, **`grid()`** yöneticisinin en uygun kullanım alanı ızgara biçimli arayüzlerdir. Mesela; hesap makineleri gibi… Şu örneğe bir bakalım:

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

from Tkinter import *

pencere = Tk()

#bir hesap makinesinde pencerenin boyutlandırılabilir
#olması anlamsız olacaktır. O yüzden pencereleri
#boyutlandırılamaz hale getiriyoruz.
pencere.resizable(width=FALSE,height=FALSE)

#hesap makinemizin tuş takımında yer alacak
#simgeleri ve sayıları oluşturuyoruz.
liste = [\
 "9", "8", "7",
 "6", "5", "4",
 "3", "2", "1",
```

```
 "0", "+", "-",
 "/", "*", "=",
 "C"]

#grid() yöneticisinin row (sıra) ve column (sütun)
#seçenekleri için birer başlangıç değeri belirliyoruz.
sira = 1
sutun = 0

#tuş takımındaki bütün pencere araçlarını tek bir
#for döngüsü ile oluşturuyoruz.
for i in liste:
 Button(text=i,width=4,relief=GROOVE).grid(row=sira,
    column=sutun)
 #column seçeneğinin değeri her defasında bir sayı artıyor.
 sutun += 1

 #sutun'un yani "column'un" değeri 2'yi geçtiğinde
 #column'un değerini sıfırlıyoruz. Böylece tuşlar
 #alt alta dizilebiliyor. row'un (sıra) değeri de
 #her defasında bir sayı artıyor.
 if sutun > 2:
  sutun = 0
  sira += 1

mainloop()
```

Gördüğünüz gibi yukarıda bir hesap makinesinde olması gereken temel tuşları, basit bir **`for`** döngüsü yardımıyla bir çırpıda oluşturduk. Yukarıda kodlar içinde verdiğim yorumları dikkatlice incelemenizi öneririm.

Aşağıda hesap makinemizin bitmiş halini bulabilirsiniz:

```
#!/usr/bin/env python
#-*-coding:utf-8-*-
from __future__ import division

from Tkinter import *

depo = ""
pencere = Tk()
pencere.resizable(width=FALSE,height=FALSE)
```

```
ekran = Entry(width=20)
ekran.grid(row=0, column=0, columnspan=3,ipady=4)
def hesapla(tus):
 global depo
 if tus in "0123456789":
  ekran.insert(END,tus)
  depo = depo + tus

 if tus in "+-/*":
  depo = depo + tus
  ekran.delete(0,END)

 if tus == "=":
  ekran.delete(0,END)
  #Normalde tehlikeli bir fonksiyon olan eval()'i
  #nasıl kullandığımıza dikkat edin...
  hesap = eval(depo,{"__builtins__":None},{})
  depo = str(hesap)
  ekran.insert(END,depo)

 if tus == "C":
  ekran.delete(0,END)
  depo = ""

liste = [\
 "9", "8", "7",
 "6", "5", "4",
 "3", "2", "1",
 "0", "+", "-",
 "/", "*", "=",
 "C"]

sira = 1
sutun = 0
for i in liste:
 komut = lambda x=i: hesapla(x)
 Button(text=i,
     width=5,
     relief=GROOVE,
     command=komut).grid(row=sira,
          column=sutun)
 sutun += 1

 if sutun > 2:
```

```
  sutun = 0
  sira += 1

mainloop()
```

Bu hesap makinesinde aklınıza takılan bir konu olursa bana nasıl ulaşacağınızı biliyorsunuz. Bu arada isterseniz hesap makinemizi bir de sınıflı yapı içinde görelim:

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

from __future__ import division

from Tkinter import *

class Uygulama(object):
 def __init__(self):
  self.guiPenAr()
  self.gV()

 def gV(self):
  self.depo = ""

 def guiPenAr(self):
  self.ekran = Entry(width=20)
  self.ekran.grid(row =0, column=0,
      columnspan=3, ipady=4)

  self.liste = [\
     "9", "8", "7",
     "6", "5", "4",
     "3", "2", "1",
     "0", "+", "-",
     "/", "*", "=",
     "C"]

  self.sira = 1
  self.sutun = 0

  for i in self.liste:
   self.komut = lambda x=i: self.hesapla(x)
   Button(text=i,
       width=5,
       relief=GROOVE,
```

```
        command=self.komut).grid(row=self.sira,
            column=self.sutun)
    self.sutun += 1

    if self.sutun > 2:
     self.sutun = 0
     self.sira += 1

 def hesapla(self, tus):
  self.tus = tus
  if self.tus in "0123456789":
   self.ekran.insert(END,self.tus)
   self.depo = self.depo + self.tus

  if self.tus in "+-/*":
   self.depo = self.depo + self.tus
   self.ekran.delete(0,END)

  if self.tus == "=":
   self.ekran.delete(0,END)
   self.hesap = eval(self.depo,{"__builtins__":None},{})
   self.depo = str(self.hesap)
   self.ekran.insert(END,self.depo)

  if self.tus == "C":
   self.ekran.delete(0,END)
   self.depo = ""

pencere = Tk()
pencere.resizable(width=FALSE,height=FALSE)
uyg = Uygulama()

mainloop()
```

Dikkat ederseniz, bir önceki bölümde **pack()** yöneticisini incelerken öğrendiğimiz **GROOVE**, **SUNKEN** vb. stilleri **grid()** geometri yöneticisiyle birlikte de kullanabiliyoruz.

Böylelikle bir geometri yöneticisini daha incelemiş olduk. Sıra geldi sonuncu geometri yöneticimize...

PLACE() GEOMETRİ YÖNETİCİSİ

Tahminimce bu geometri yöneticisi Tkinter'deki üç yönetici içinden en çok beğeneceğiniz yönetici olmaya adaydır... Bütün geometri yöneticileri içinde, pencere araçlarını en hassas şekilde yerleştirmemizi sağlayan yönetici **`place()`** adlı geometri yöneticisidir.

`place()` geometri yöneticisi ile çalışırken bize temel olarak iki seçenek yardımcı olacak: **`relx`** ve **`rely`**. Tahmin edebileceğiniz gibi **`relx`**, **`x`** düzlemi üzerindeki işlemler için, **`rely`** ise **`y`** düzlemi üzerindeki işlemler için kullanılacak. Hemen bir örnek görelim:

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

from Tkinter import *

pencere = Tk()

giris1 = Entry()
giris1.place(relx=0.0, rely=0.0,relheight=0.15)

dugme1 = Button(text="Düğme1")
dugme1.place(relx=0.7, rely=0.0,relheight=0.16)

dugme2 = Button(text="Düğme2")
dugme2.place(relx=0.0,rely=0.2,relwidth=1)

giris2 = Entry()
giris2.place(relx=0.0,rely=0.35,relheight=0.5,relwidth=1)

mainloop()
```

Bu örnekte **`relx`** ve **`rely`** seçeneklerinin yanısıra; **`relheight`** ve **`relwidth`** seçeneklerini de kullandık. Rahatlıkla anlayabileceğiniz gibi bu **`place()`** yöneticisi yardımıyla, pencerelerimizi istediğimiz şekilde düzenleyebiliyoruz. Ama tabii ki, gördüğünüz gibi koordinatları tam olarak vermeyi gerektirdiği için bu yöneticiyle çalışmak biraz emek istiyor. Bu yüzden bazı işler için bu yöneticiyi kullanmak pek pratik olmayabilir. Böyle durumlarda öteki yöneticilerin neler yapabileceğini de göz önünde bulundurmakta fayda var. Bir sonraki konumuz olan checkbutton pencere aracında **`place()`** adlı geometri yöneticisinden bahsetmeye devam edeceğiz.

PENCERE ARAÇLARI (WIDGETS): 2.BÖLÜM

4

Checkbutton Pencere Aracı

Checkbutton denen şey, bildiğimiz onay kutusudur. Basitçe şu şekilde oluşturuyoruz bu onay kutusunu:

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

from Tkinter import *

pencere = Tk()

onay = Checkbutton()
onay.pack()

mainloop()
```

Tabii ki muhtemelen bu onay kutumuzun, hatta kutularımızın birer adı olsun isteyeceğiz:

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

from Tkinter import *

pencere = Tk()

onay_el = Checkbutton(text="elma")
onay_el.pack()
```

```
onay_sa = Checkbutton(text="salatalık")
onay_sa.pack()

onay_do = Checkbutton(text="domates")
onay_do.pack()

onay_ka = Checkbutton(text="karnıbahar")
onay_ka.pack()

mainloop()
```

Gayet güzel. Ama gördüğünüz gibi öğeler alt alta sıralanırken hizalı değiller. Dolayısıyla göze pek hoş görünmüyorlar. Eğer istersek şöyle bir görünüm verebiliriz onay kutularımıza:

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

from Tkinter import *

pencere = Tk()

onay_el = Checkbutton(text="elma")
onay_el.pack(side=LEFT)

onay_sa = Checkbutton(text="salatalık")
onay_sa.pack(side=LEFT)

onay_do = Checkbutton(text="domates")
onay_do.pack(side=LEFT)

onay_ka = Checkbutton(text="karnıbahar")
onay_ka.pack(side=LEFT)

mainloop()
```

Öğeler böyle yan yana dizildiklerinde fena görünmüyorlar, ama biz yine de öğelerin düzgün şekilde alt alta dizilmesini isteyebiliriz. Burada geometri yöneticilerinden biri olan **`place()`** adlı olanı tercih edebiliriz. Şimdi bununla ilgili bir örnek görelim. Mesela; bir tane onay kutusu oluşturup bunu en sola yaslayalım:

```
from Tkinter import *

pencere = Tk()

onay = Checkbutton(text="Kubuntu")
```

```
onay.place(relx = 0.0, rely = 0.1)

mainloop()
```

Bir önceki bölümde öğrendiğimiz **place()** yöneticisini nasıl kullandığımızı görüyorsunuz. Bu yöneticinin **relx** ve **rely** seçenekleri yardımıyla onay kutusunun koordinatlarını belirliyoruz. Buna göre onay kutumuz **x** düzlemi üzerinde 0.0 konumunda; **y** düzlemi üzerinde ise 0.1 konumunda yer alıyor. Yani yukarıdan aşağıya 0.0; soldan sağa 0.1 konumunda bulunuyor pencere aracımız.

Hemen birkaç tane daha onay kutusu ekleyelim:

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

from Tkinter import *

pencere = Tk()

onay_pa = Checkbutton(text=”Kubuntu”)
onay_pa.place(relx = 0.0, rely = 0.1)

onay_de = Checkbutton(text=”Debian”)
onay_de.place(relx = 0.0, rely = 0.2)

onay_ub = Checkbutton(text=”Ubuntu”)
onay_ub.place(relx = 0.0, rely = 0.3)

onay_wix = Checkbutton(text=”Windows XP”)
onay_wix.place(relx = 0.0, rely = 0.4)

mainloop()
```

Dikkat ederseniz yukarıdaki bütün onay kutularının **relx** seçeneği 0.0 iken, **rely** seçenekleri birer birer artmış. Bunun nedeni, bütün onay kutularını sola yaslayıp hepsini alt alta dizmek isteyişimiz. Bu örnek **relx** ve **rely** seçeneklerinin nasıl kullanılacağı konusunda iyi bir fikir vermiş olmalı. Eğer bir kolaylık sağlayacaksa, **relx**'i sütun, **rely**'yi ise satır olarak düşünebilirsiniz.

Şimdi de mesela Kubuntu'yu Debian'ın yanına, Ubuntu'yu da Windows XP'nin yanına gelecek şekilde alt alta sıralayalım:

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

from Tkinter import *

pencere = Tk()

onay_pa = Checkbutton(text="Kubuntu")
onay_pa.place(relx = 0.0, rely = 0.1)

onay_de = Checkbutton(text="Debian")
onay_de.place(relx = 0.4, rely = 0.1)

onay_ub = Checkbutton(text="Ubuntu")
onay_ub.place(relx = 0.0, rely = 0.2)

onay_wix = Checkbutton(text="Windows XP")
onay_wix.place(relx = 0.4, rely = 0.2)

mainloop()
```

Kubuntu'yu 0.0 no'lu sütunun 0.1 no'lu satırına; Debian'ı da 0.4 no'lu sütunun 0.1 no'lu satırına yerleştirdik. Aynı şekilde Ubuntu'yu 0.0 no'lu sütunun 0.2 no'lu satırına; Windows XP'yi de 0.4 no'lu sütunun 0.2 no'lu satırına yerleştirdik.

Eğer oluşturduğumuz bu onay kutularına biraz canlılık katmak istersek, yani kutunun seçili olup olmamasına göre bazı işler yapmak istersek, kullanmamız gereken bazı özel kodlar var. Önce hemen yazmamız gereken ilk satırları yazalım:

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

from Tkinter import *

pencere = Tk()
```

Bunun hemen ardından şöyle iki satır ekleyelim:

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

from Tkinter import *
```

```
pencere = Tk()

d = IntVar()
d.set(0)
```

Burada **d=IntVar()** satırıyla yaptığımız şey **d** adlı bir değişken oluşturup, bunun değeri olarak **IntVar()** ifadesini belirlemek. Peki, bu **IntVar()** denen şey de ne oluyor?

IntVar(), İngilizce **Integer Variable** (*Sayı Değişkeni*) ifadesinin kısaltması. Bu ifade yardımıyla değişken olarak bir sayı belirlemiş oluyoruz. Bunun dışında Tkinter'de kullanacağımız, buna benzer iki ifade daha var: **StringVar()** ve **DoubleVar()**

StringVar() yardımıyla karakter dizilerini; **DoubleVar()** yardımıyla da ondalık sayıları (*kayan noktalı sayılar - floats*) depolayabiliyoruz. Buraya kadar anlattıklarımız biraz bulanık gelmiş olabilir. Ama endişeye hiç gerek yok. **d=IntVar()** satırının hemen altındaki **d.set(0)** ifadesi pek çok şeyi açıklığa kavuşturacak. O halde hemen bu satırın anlamını kavramaya çalışalım.

Aslında **d=IntVar()** ifadesi yardımıyla başladığımız işi tamamlamamızı sağlayan satır bu **d.set(0)** satırı. Bunun yardımıyla **d=IntVar()** ifadesinin değeri olarak **0**'ı seçtik. Yani bu satırı bir onay kutusu için yazdığımızı düşünürsek, onay kutusunun değerini **seçili değil** olarak belirlemiş olduk. Bu iki satırdan sonra oluşturacağımız onay kutusu karşımıza ilk çıktığında işaretlenmemiş olacaktır. Hemen görelim:

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

from Tkinter import *

pencere = Tk()

d = IntVar()
d.set(0)

btn1 = Checkbutton(text="Kubuntu", variable=d)
btn1.place(relx=0.0,rely=0.1)

mainloop()
```

Burada dikkatimizi çeken bir farklılık, **btn1** adlı onay kutusunun seçenekleri arasına **variable** adlı bir seçeneğin girmiş olması. Bu seçeneğin işlevini az çok

anlamış olmalısınız: Yukarıda belirlediğimiz **d=IntVar()** ve **d.set(0)** ifadeleri ile **Checkbutton** pencere aracı arasında bağlantı kuruyor. Yani bu seçenek yardımıyla pencerece aracına, "*değişken olarak d'yi kullan*" emrini veriyoruz.

Bu kodları çalıştırdığımızda karşımıza içinde bir onay kutusu bulunan bir pencere açılır. Bu penceredeki onay kutusu seçili değildir. İsterseniz yukarıdaki kodlar içinde yer alan **d.set(0)** ifadesini **d.set(1)** olarak değiştirip kodlarımızı öyle çalıştıralım:

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

from Tkinter import *

pencere = Tk()

d = IntVar()
d.set(1)

btn1 = Checkbutton(text="Kubuntu", variable=d)
btn1.place(relx=0.0,rely=0.1)

mainloop()
```

Gördüğünüz gibi bu defa oluşan onay kutusu seçili vaziyette. Eğer yukarıdaki kodlar içindeki **d.set()** satırını tamamen kaldırırsak, Tkinter varsayılan olarak **d.set()** için 0 değerini atayacaktır.

Gelin şimdi bu kodları biraz geliştirelim. Mesela; penceremizde bir **Entry()** aracı olsun ve onay kutusunu seçtiğimiz zaman, bu **Entry()** aracında bizim belirlediğimiz bir cümle görünsün. Bu kodlar içinde göreceğimiz get ifadesine aslında pek yabancı sayılmayız. Bunu daha önceden **Entry()** aracını kullanırken de görmüştük. Bu ifade yardımıyla bir değişkeninin değerini sonradan kullanmak üzere depolayabiliyoruz. Aşağıdaki örneği dikkatle inceleyin:

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

from Tkinter import *

pencere = Tk()

# Onay kutusu için bir değişken oluşturuyoruz
v = IntVar()
# Bu değişkene değer olarak "0" atayalım
```

```
v.set(0)

# Öbür onay kutusu için başka bir değişken daha oluşturuyoruz
z = IntVar()
# Bu değişkene de "0" değerini atıyoruz.
z.set(0)
# Bu kez bir karakter değişkeni oluşturalım
d = StringVar()
# Bu değişkenin değeri "Kubuntu" olsun.
d.set("Kubuntu")
# Bir tane daha karakter değişkeni oluşturalım
e = StringVar()
# Bunun da değeri "Debian" olsun.
e.set("Debian")

# Şimdi onay kutularını seçili hale getirdiğimizde çalışmasını
#istediğimiz komut için bir fonksiyon oluşturuyoruz:
def onay():
 # Eğer "v" değişkeninin değeri "1" ise....
 if v.get() == 1:
  # d.get() ile "d" değişkeninin değerini alıyoruz...
  giris.insert(0,"Merhaba %s kullanıcısı" %d.get())

 # Yok eğer "z" değişkeninin değeri "1" ise...
 elif z.get() == 1:
  giris.delete(0,END)
  # e.get() ile "e" değişkeninin değerini alıyoruz...
  giris.insert(0,"Merhaba %s kullanıcısı" %e.get())

 # Değişkenlerin değer "1" değil, başka bir değer ise..."
 else:
  giris.delete(0,END)

#"text" seçeneğinin değerinin "d.get()" olduğuna dikkat edin.
onay_pa = Checkbutton(text=d.get(), variable=v, command=onay)
onay_pa.place(relx = 0.0, rely = 0.1)

#"text" seçeneğinin değerinin "e.get()" olduğuna dikkat edin.
onay_de = Checkbutton(text=e.get(), variable=z, command=onay)
onay_de.place(relx= 0.0, rely= 0.2)

giris = Entry(width=24)
giris.place(relx = 0.0, rely=0.0)

mainloop()
```

Toplevel Pencere Aracı

Toplevel aracı, bizim ana pencere dışında farklı bir pencere daha açmamızı sağlar. Mesela; bir ana pencere üstündeki düğmeye bastığınızda, başka bir pencere daha açılsın istiyorsanız bu işlevi kullanmanız gerekiyor. En basit kullanımı şöyledir:

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

from Tkinter import *

pencere = Tk()
pencere2 = Toplevel()

mainloop()
```

Bu kodları çalıştırdığımızda ekranda ikinci bir pencerenin daha açıldığını görürüz. Diyelim ki bu ikinci pencerenin, bir düğmeye basıldığında açılmasını istiyoruz:

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

from Tkinter import *
pencere = Tk()

def ekle():
 pencere2 = Toplevel()

btn_pen2 = Button(text="ekle", command=ekle)
btn_pen2.pack()

mainloop()
```

Gördüğünüz gibi **pencere2**'nin açılışını bir fonksiyona bağladık. Ardından da ana pencere üzerinde **btn_pen2** adlı bir düğme oluşturduk ve bu düğmeye **command** seçeneği yardımıyla yukarıda tanımladığımız fonksiyonu atayarak, düğmeye basılınca ikinci pencerenin açılmasını sağladık.

Eğer açılan ikinci pencere üzerinde de başka düğmeler olsun istiyorsak, şu şekilde hareket etmemiz gerekir:

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

from Tkinter import *
pencere = Tk()

def ekle():
 pencere2 = Toplevel()
 btn_pen = Button(pencere2,text="çıkış",
      command=pencere2.destroy)
 btn_pen.pack()

btn_pen2 = Button(pencere,text="ekle", command=ekle)
btn_pen2.pack()

mainloop()
```

Yine gördüğünüz gibi ilk tanımladığımız **`ekle()`** fonksiyonu altında bir düğme oluşturduk. Burada yeni bir durum dikkatinizi çekmiş olmalı. **`ekle()`** fonksiyonu altında yeni tanımladığımız düğmede ilave olarak **`text`** seçeneğinden önce bir **`pencere2`** ifadesini görüyoruz. Bunu yapmamızın nedeni şu: Elimizde **`pencere`** ve **`pencere2`** adında iki adet pencere var. Böyle bir durumda oluşturulan **`pencere`** aracının hangi pencere üzerinde gösterileceğini Tkinter'e anlatmamız gerekiyor. Sadece bir pencere varken pencereleri açıkça belirtmeye gerek olmuyordu, ama eğer birden pencere var ve siz oluşturulan düğmenin hangi pencere görüntüleneceğini belirtmezseniz, örneğin bizim durumumuzda Tkinter **`ekle`** adlı düğmeye her basışta otomatik olarak ana pencere üzerinde **`çıkış`** adlı bir düğme oluşturacaktır. Anladığınız gibi, eğer aksi belirtilmemişse, pencere araçları otomatik olarak ana pencere üzerinde açılacaktır.

Diyelim ki **`a`**, **`b`**, **`c`** ve **`d`** adlarında dört adet penceremiz var. İşte hangi düğmenin hangi pencerede görüntüleneceğini belirlemek için bu özellikten faydalanacağız:

```
btn1 = Button(a, text="btn1")
btn2 = Button(b, text="btn2")
btn3 = Button(c, text="btn3")
btn4 = Button(d, text="btn4")
```

Yukarıdaki kodlarda yeni bir özellik olarak bir de **`command=pencere2.destroy`**

ifadesini görüyoruz. Aslında **destroy** komutu biraz **quit** komutuna benziyor; görevi mevcut pencereyi kapatmak. Peki, burada **pencere2.destroy** komutu yerine **pencere2.quit** komutunu kullanamaz mıyız? Tabii ki kullanabiliriz, ancak kullanırsak **çıkış** düğmesine bastığımızda sadece **pencere2** değil, doğrudan ana pencerenin kendisi de kapanacaktır. Eğer ikinci pencerenin kapanıp ana pencerenin açık kalmasını istiyorsanız, kullanmanız gereken komut destroy; yok eğer ikinci pencere kapandığında ana pencere de kapansın istiyorsanız kullanmanız gereken komut quit olmalıdır. **btn_pen2** adlı düğmeyi ise, **text** seçeneğinin hemen önüne yazdığımız **pencere** ifadesinden de anlayacağınız gibi **pencere** adlı pencerenin üzerine yerleştiriyoruz. Yukarıda bahsettiğimiz gibi burada **pencere** ifadesini kullanmasanız da olur, çünkü zaten Tkinter siz belirtmesiniz de düğmeyi otomatik olarak ana pencere üzerinde açacaktır. Tabii düğmeyi ana pencerede değil de ikincil pencereler üzerinde açmak isterseniz bunu Tkinter'e açıkça söylemeniz gerekir.

Listbox Pencere Aracı

Bu araç bize pencereler üzerinde bir liste içeren kutu hazırlama imkânı veriyor. Hemen basit bir **Listbox** kullanımı örneği görelim:

```
liste = Listbox()
liste.pack()
```

Gördüğünüz gibi bu aracın da kullanımı öteki araçlardan hiç farklı değil. İsterseniz bu aracı bir de bağlamında görelim:

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

from Tkinter import *

pencere = Tk()

liste = Listbox()
liste.pack()

mainloop()
```

Bu haliyle pencerenin tamamını kapladığı için çok belirgin olmayabilir liste kutumuz.

Ama pencereye bir de düğme eklersek liste kutusu daha rahat seçilebilir hale gelecektir:

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

from Tkinter import *

pencere = Tk()

liste = Listbox()
liste.pack()

btn = Button(text="ekle")
btn.pack()

mainloop()
```

Hatta şimdiye kadar öğrendiğimiz başka özellikleri kullanarak daha şık bir görünüm de elde edebiliriz:

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

from Tkinter import *

pencere = Tk()

liste = Listbox(bg="white")
liste.pack()

etiket = Label(text="####################",
      fg="magenta", bg="light green")
etiket.pack()

btn = Button(text="ekle",bg="orange",fg="navy")
btn.pack()

etiket2 = Label(text="####################",
    fg="magenta", bg="light green")
etiket2.pack()

mainloop()
```

Tabii ki siz pencere araçlarını ve hayal gücünüzü kullanarak çok daha çekici görünümler ve renkler elde edebilirsiniz.

Şimdi liste kutumuza bazı öğeler ekleyelim. Bunun için şu basit ifadeyi kullanacağız:

```
liste.insert(END, "öğe 1")
```

Bunu bu şekilde tek başına kullanamayız. Hemen bu parçacığı yukarıdaki kod içine yerleştirelim:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
from Tkinter import *

pencere = Tk()

liste = Listbox(bg="white")
liste.insert(END,"öğe 1")
liste.pack()

etiket = Label(text="#####################",
      fg="magenta", bg="light green")
etiket.pack()

btn = Button(text="ekle",bg="orange",fg="navy")
btn.pack()

etiket2 = Label(text="#####################",
    fg="magenta", bg="light green")
etiket2.pack()

mainloop()
```

Gördüğünüz gibi, bu parçacığı, liste kutusunu tanımladığımız satırın hemen altına ekledik. Biz liste kutumuza aynı anda birden fazla öğe de eklemek isteyebiliriz. Bunun için basitçe bir **`for`** döngüsü kullanabiliriz:

```
gnulinux_dagitimlari = ["Pardus", "Debian", "Ubuntu", "PclinuxOS",
      "TruvaLinux", "Gelecek Linux"]

for i in gnulinux_dagitimlari:
 liste.insert(END, i)
```

Hemen bu kodları da yerli yerine koyalım:

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

from Tkinter import *

pencere = Tk()

liste = Listbox(bg="white")
liste.pack()

gnulinux_dagitimlari = ["Pardus", "Debian", "Ubuntu",
      "PclinuxOS", "TruvaLinux",
      "Gelecek Linux"]

for i in gnulinux_dagitimlari:
 liste.insert(END, i)

etiket = Label(text="#################",
      fg="magenta", bg="light green")
etiket.pack()

btn = Button(text="ekle",bg="orange",fg="navy")
btn.pack()

etiket2 = Label(text="#################",
    fg="magenta", bg="light green")
etiket2.pack()

mainloop()
```

Gayet güzel bir liste kutusu oluşturduk ve listemizdeki öğeleri de rahatlıkla seçebiliyoruz. Yalnız dikkat ettiyseniz, ana pencere üzerindeki ekle düğmesi şu anda hiçbir işe yaramıyor. Tabii ki onu oraya boşu boşuna koymadık. Hemen bu düğmeye de bir işlev atayalım. Mesela; bu düğmeye basıldığında ayrı bir pencere açılsın ve kullanıcıdan girdi alarak ana penceredeki liste kutusuna eklesin. Tabii ki bu yeni açılan pencerede de bir giriş kutusu ve bir de işlemi tamamlamak için bir düğme olsun. Hemen işe koyulalım:

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

from Tkinter import *

pencere = Tk()
```

```
liste = Listbox(bg="white")
liste.pack()

gnulinux_dagitimlari = ["Pardus", "Debian", "Ubuntu",
      "PclinuxOS", "TruvaLinux",
      "Gelecek Linux"]

for i in gnulinux_dagitimlari:
 liste.insert(END, i)

def yeni():
 global giris
 pencere2 = Toplevel()
 giris = Entry(pencere2)
 giris.pack()
 btn2 = Button(pencere2, text="tamam",command=ekle)
 btn2.pack()

def ekle():
 liste.insert(END,giris.get())
 giris.delete(0,END)

etiket = Label(text="##################",
      fg="magenta", bg="light green")
etiket.pack()

btn = Button(text="ekle",bg="orange",
    fg="navy", command=yeni)
btn.pack()

etiket2 = Label(text="##################",
    fg="magenta", bg="light green")
etiket2.pack()

mainloop()
```

İsterseniz daha anlaşılır olması için parça parça ilerleyelim:

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

from Tkinter import *
pencere = Tk()
```

Bu kısmı zaten biliyoruz. Herhangi bir açıklamaya gerek yok.

```
liste = Listbox(bg="white")
liste.pack()

gnulinux_dagitimlari = ["Pardus", "Debian", "Ubuntu",
      "PclinuxOS", "TruvaLinux",
      "Gelecek Linux"]

for i in gnulinux_dagitimlari:
 liste.insert(END, i)
```

Bu kısımda bildiğiniz gibi önce **liste** adında bir liste kutusu (*Listbox*) oluşturduk. Liste kutumuzun arkaplan rengini de beyaz olarak belirledik.

Ardından da **gnulinux_dagitimlari** adında bir liste oluşturduk ve tek tek öğelerini yazdık. Hemen alt satırda bir **for** döngüsü yardımıyla **gnulinux_dagitimlari** adlı listedeki bütün öğeleri **liste** adı verdiğimiz liste kutusu içine yerleştirdik:

```
def yeni():
 global giris
 pencere2 = Toplevel()
 giris = Entry(pencere2)
 giris.pack()
 btn2 = Button(pencere2, text="tamam",command=ekle)
 btn2.pack()
```

Burada yaptığımız iş basit bir fonksiyon oluşturmaktan ibaret. Önce **yeni()** adlı bir fonksiyon oluşturuyoruz. Ardından **pencere2** adıyla ana pencereden ayrı bir pencere daha oluşturuyoruz. Bu yeni pencere, ileride **ekle** tuşuna bastığımızda açılmasını istediğimiz pencere oluyor. Alt satırda, bu yeni pencere üzerine yerleştirmek için **giris** adlı bir **Entry()** pencere aracı oluşturuyoruz. Parantez içine **pencere2** yazmayı unutmuyoruz, çünkü bu **Entry()** aracının oluşmasını istediğimiz yer ikinci pencere. Dikkat ettiyseniz fonksiyonu tanımlarken **global giris** adlı bir satır daha ekledik. Bu satırın amacı; fonksiyon içindeki **giris** adlı **Entry()** aracını fonksiyon dışında da kullanabilmek. Çünkü bu **Entry()** aracı bize daha sonra da lazım olacak. **Entry()** aracına benzer şekilde bir de **btn2** adıyla bir düğme oluşturuyoruz.

Bunu da ikinci penceremize yerleştiriyor, adını tamam koyuyor ve komut olarak aşağıda tanımlayacağımız **ekle()** fonksiyonunu seçiyoruz:

```
def ekle():
 liste.insert(END,giris.get())
 giris.delete(0,END)
```

İşte bu parçada da **ekle()** adlı bir fonksiyon oluşturduk. Burada **liste.insert** ifadesi **liste** adlı liste kutusuna ekleme yapmamızı sağlıyor. Parantez içindeki **giris.get()** ifadesi size tanıdık geliyor olmalı. Çünkü aynı ifadeyi **Entry()** pencere aracını anlatırken de görmüştük. Hatırlarsanız bu ifade sayesinde **Entry()** aracına kullanıcı tarafından girilen verileri daha sonra kullanmak amacıyla elimize tutabiliyorduk. İşte burada da bu ifade yardımıyla giris adlı **Entry()** pencere aracının içeriğini alıp **liste** adlı liste kutusu içine yerleştiriyoruz. Alt satırdaki **giris.delete(0,END)** ifadesi ise **Entry()** aracına kullanıcı tarafından giriş yapıldıktan sonra kutunun boşaltılmasını sağlıyor:

```
etiket = Label(text="##################",
      fg="magenta", bg="light green")
etiket.pack()
btn = Button(text="ekle",bg="orange",
    fg="navy", command=yeni)
btn.pack()
etiket2 = Label(text="##################",
    fg="magenta", bg="light green")
etiket2.pack()
mainloop()
```

Bu son parçada bilmediğimiz hiçbir şey yok. Normal bir şekilde "etiket" adı verdiğimiz **Label()** aracını tanımlıyoruz. Burada **Label()** aracını süsleme amacıyla nasıl kullandığımıza dikkat edin. **fg** ve **bg** seçeneklerini de önceki bölümlerden hatırlıyoruz. **fg** önplandaki rengi; **bg** ise arkaplandaki rengi seçmemizi sağlıyor. **magenta** ve **light green** ise kullanacağımız renklerin adları oluyor. Bunun altında ise basit bir **Button()** aracı tanımlıyoruz. İsmini ve renklerini belirledikten sonra da **command** seçeneği yardımıyla yukarıda tanımladığımız **yeni()** adlı fonksiyonu bu düğmeye bağlıyoruz. Bunun aşağısındaki **etiket2**'nin ise **etiket** adlı araçtan hiçbir farkı yok.

Kodlarımızı çalıştırdığımızda karşımıza gayet hoş, üstelik güzel de bir işlevi olan bir pencere çıkıyor. Burada ekle düğmesine bastığımızda karşımıza yeni bir pencere açılıyor. Bu yeni pencerede, kullanıcının giriş yapabilmesi için bir **`Entry()`** aracı, bir de işlemi tamamlayabilmesi için **`Button()`** aracı yer alıyor. Kullanıcı **`Entry()`** aracına bir veri girip **tamam** düğmesine bastığında **`Entry()`** aracına girilen veri ana penceredeki liste kutusuna ekleniyor. Ayrıca ilk veri girişinin ardından **`Entry()`** aracı içindeki alan tamamen boşaltılıyor ki kullanıcı rahatlıkla ikinci veriyi girebilsin.

Siz de burada kendinize göre değişiklikler yaparak özellikle ikinci pencereyi göze daha hoş görünecek bir hale getirebilirsiniz.

Burada dikkat ederseniz, ikinci pencerede, giriş kutusuna hiçbir şey yazmadan tamam düğmesine basarsak ana penceredeki liste kutusuna boş bir satır ekleniyor. Şimdi öyle bir kod yazalım ki, kullanıcı eğer ikinci penceredeki giriş kutusuna hiçbir şey yazmadan tamam düğmesine basarsa giriş kutusu içinde "*Veri Yok!*" yazısı belirsin ve bu yazı ana penceredeki liste kutusuna eklenmesin:

Bunun için kodlarımız içindeki **`ekle()`** fonksiyonuna iki adet if deyimi eklememiz gerekiyor:

```
def ekle():
 if not giris.get():
  giris.insert(END,"Veri Yok!")
 if giris.get() != "Veri Yok!":
  liste.insert(END,giris.get())
  giris.delete(0,END)
```

Gördüğünüz gibi "*giris*" boşken "*tamam*" tuşuna basıldığında "*Veri Yok!*" ifadesi ekrana yazdırılıyor. Ancak burada şöyle bir problem var: Eğer "*Veri Yok!*" ifadesi ekrana yazdırıldıktan sonra kullanıcı bu ifadeyi silmeden bu ifadenin yanına bir şeyler yazıp "*tamam*"a basarsa "*Veri Yok!*" ifadesiyle birlikte o yeni yazdığı şeyler de listeye eklenecektir. Bunu engellemek için kodumuzu şu hale getirebiliriz:

```
def ekle():
 if not giris.get():
  giris.insert(END,"Veri Yok!")
 if not "Veri Yok!" in giris.get():
  liste.insert(END,giris.get())
  giris.delete(0,END)
```

Yani şöyle demiş oluyoruz bu ifadelerle: *"Eğer giris adlı Entry aracı boş ise, araç içinde 'Veri Yok!' ifadesini göster. Eğer giris adlı Entry aracı içinde 'Veri Yok!' ifadesi bulunmuyorsa, liste adlı Listbox aracına giris içindeki bütün veriyi yazdır!"*

Liste kutumuza öğelerimizi ekledik. Peki, bu öğeleri silmek istersek ne yapacağız?

Niyetimiz liste kutusundan öğe silmek olduğunu göre en başta bir silme düğmesi oluşturmamız mantıklı olacaktır:

```
btn_sil = Button()
btn_sil.pack()
```

Bu düğmenin bir iş yapabilmesi için de bu düğmeye bir fonksiyon atamamız gerekir:

```
def sil():
 liste.delete(ACTIVE)
```

Burada gördüğünüz gibi, silme işlemi için **`liste.delete`** ifadesini kullanıyoruz. Parantez içindeki **`ACTIVE`** ifadesi ise liste kutusu içinden bir seçim yapıp **sil** düğmesine basınca bu seçili öğenin silinmesini sağlayacak. Yani aktif olan öğe silinecek. Bu iki parçayı öteki komutlarla birlikte bir görelim bakalım:

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

from Tkinter import *

pencere = Tk()

liste = Listbox(bg="white")
liste.pack()

gnulinux_dagitimlari = ["Pardus", "Debian", "Ubuntu",
      "PclinuxOS", "TruvaLinux",
      "Gelecek Linux"]

for i in gnulinux_dagitimlari:
 liste.insert(END, i)

def yeni():
 global giris
 pencere2 = Toplevel()
```

```
 giris = Entry(pencere2)
 giris.pack()
 btn2 = Button(pencere2, text="tamam",command=ekle)
 btn2.pack()

def ekle():
 if not giris.get():
  giris.insert(END,"Veri Yok!")
 if not "Veri Yok!" in giris.get():
  liste.insert(END,giris.get())
  giris.delete(0,END)

def sil():
 liste.delete(ACTIVE)

etiket = Label(text="#################",
      fg="magenta", bg="light green")
etiket.pack()

btn = Button(text="ekle",bg="orange",
    fg="navy", command=yeni)
btn.pack()

btn_sil = Button(text="sil",bg="orange",
     fg="navy",command=sil)
btn_sil.pack()

etiket2 = Label(text="#################",
    fg="magenta", bg="light green")
etiket2.pack()

mainloop()
```

Tabii ki, sil düğmesinin görünüşünü pencere üzerindeki öteki öğelere uydurmak için **fg** ve **bg** seçenekleri yardımıyla ufak bir renk ayarı yapmayı da unutmadık.

Böylece bir pencere aracını daha bitirmiş olduk. Gelelim sıradaki pencere aracımıza…

Menu Pencere Aracı

Adından da anlaşılacağı gibi bu araç bize pencerelerimiz üzerinde menüler hazırlama olanağı sağlıyor. "Menu" pencere aracının kullanımı öteki araçlardan birazcık daha farklıdır, ama kesinlikle zor değil. Hemen küçücük bir menü hazırlayalım:

Önce standart satırlarımızı ekliyoruz:

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

from Tkinter import *
pencere = Tk()
```

Şimdi menümüzü oluşturmaya başlayabiliriz...

```
menu = Menu(pencere)
```

Burada **menu** adlı bir Menu pencere aracı oluşturduk. Adı **menu** olmak zorunda değil, siz istediğiniz ismi kullanabilirsiniz. Ama tabii ki pencere aracımızın adı olan Menu'yu değiştiremeyiz. Buraya kadar öteki pencere araçlarından hiçbir farkı yok.

Parantez içindeki **pencere** ifadesinden de anladığımız gibi bu pencere aracını ana pencere üzerinde oluşturuyoruz. Hatırlayacağınız gibi burada **pencere** diye açık açık belirtmesek de, Tkinter pencere aracımızı otomatik olarak ana pencere üzerinde oluşturacaktır. Aslında bu satır yardımıyla ana pencerenin en üstünde, sonradan gelecek menüler için bir menü çubuğu oluşturmuş oluyoruz:

```
pencere.config(menu=menu)
dosya = Menu(menu)
```

Burada **config()** metodu yardımıyla öncelikle **menu** adlı aracı **pencere**'ye bağlıyoruz. Parantez içindeki ilk **menu** ifadesi, tıpkı öteki pencere araçlarında gördüğümüz **text** ifadesi gibi bir seçenek. İkinci menu ifadesi ise yukarıda bizim Menu aracına kendi verdiğimiz isim. Bu isim herhangi bir kelime olabilirdi. Yani en başta menünün adını **a** olarak belirleseydik, burada **menu=a** da diyebilirdik.

İkinci satırda ise **dosya** adlı başka bir Menu pencere aracı daha oluşturuyoruz. Hatırlarsanız ilk Menu aracını oluştururken parantez içine **pencere** yazarak aracı pencereye bağlamıştık. Bu kezse aracımızı bir önceki menünün üzerinde oluşturuyoruz. Aslında bu satır yardımıyla bir önceki aşamada oluşturduğunuz araç çubuğu üzerinde **iner menü** (*drop-down menu*) için bir yer açmış oluyoruz:

```
menu.add_cascade(label="Dosya",menu=dosya)
```

Şimdi yapmamız gereken, menünün aşağıya doğru açılmasını yani inmesini sağlamak. Bu iş için yukarıda gördüğünüz **add_cascade()** metodunu kullanıyoruz. Bu metodun **menu** adlı menü araç çubuğuna bağlı olduğuna dikkat edin. Parantez içinde gördüğümüz **label** ifadesi de tıpkı **text** gibi, menüye ad vermemizi sağlıyor. Menümüzün adını **Dosya** olarak belirledik. Parantez içindeki diğer ifade olan **menu** de **Dosya**'nın hangi araç çubuğu üzerinde oluşturulacağını gösteriyor:

```
dosya.add_command(label="Aç")
dosya.add_command(label="Kaydet")
dosya.add_command(label="Farklı Kaydet...")
dosya.add_command(label="Çıkış",command=pencere.quit)

mainloop()
```

Burada gördüğümüz ifadeler ise bir üstte oluşturduğumuz **Dosya** adlı menünün alt başlıklarını oluşturmamızı sağlıyor. Burada **add_command()** metotlarının dosya adlı araç çubuğuna bağlandığına dikkat edin. Bu satırlardan anladığınız gibi **Dosya** adlı menümüzün altında "*Aç*", "*Kaydet*", "*Farklı Kaydet...*" ve "*Çıkış*" gibi alt başlıklar olacak.

Şimdi kodlarımızın hepsini birlikte görelim:

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

from Tkinter import *

pencere = Tk()

menu = Menu(pencere)
pencere.config(menu=menu)
dosya = Menu(menu)
menu.add_cascade(label="Dosya",menu=dosya)
dosya.add_command(label="Aç")
```

```
dosya.add_command(label="Kaydet")
dosya.add_command(label="Farklı Kaydet...")
dosya.add_command(label="Çıkış",command=pencere.quit)

mainloop()
```

Bu kodları çalıştırdığınızda gayet güzel bir pencere elde etmiş olacaksınız... Yalnız dikkat ettiyseniz, **Dosya**'ya bastıktan sonra açılan alt menünün en üstünde "----------" gibi bir şey görüyoruz... Oraya tıkladığımızda ise bütün menü içeriğinin tek bir grup halinde toplanıp ayrı bir pencere oluşturduğunu görüyoruz. Eğer bu özellikten hoşlanmadıysanız, bu minik çizgileri kodlar arasına **tearoff=0** ifadesini ekleyerek yok edebilirsiniz (**tearoff=0** ifadesini **dosya** adlı değişkeni oluştururken ekliyoruz).

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

from Tkinter import *

pencere = Tk()

menu = Menu(pencere)
pencere.config(menu=menu)
dosya = Menu(menu, tearoff=0)
menu.add_cascade(label="Dosya",menu=dosya)
dosya.add_command(label="Aç")
dosya.add_command(label="Kaydet")
dosya.add_command(label="Farklı Kaydet...")
dosya.add_command(label="Çıkış",command=pencere.quit)

mainloop()
```

Konu belki biraz karışık gelmiş olabilir, ama aslında hiç de öyle değil. İşin mantığını anlamak için yukarıdaki kodlarda geçen şu satırlar bize epey yardımcı olabilir:

```
menu = Menu(pencere)
dosya = Menu(menu, tearoff=0)
menu.add_cascade(label="Dosya",menu=dosya)
```

Dikkat ettiyseniz, önce **Menu** pencere aracını oluşturuyoruz. Bu araç ilk oluşturulduğunda, parantez içi ifadeden anladığımız gibi **pencere** adlı ana pencere üzerine bağlanıyor.

İkinci satır vasıtasıyla ikinci kez bir **Menu** pencere aracı oluştururken ise; parantez içi ifadeden anlaşıldığı gibi oluşturduğumuz menüyü bir önceki **Menu** pencere aracına bağlıyoruz.

Üçüncü satırda inen menüyü oluştururken de, bunu bir önceki **Menu** pencere aracı olan **dosya**'ya bağlıyoruz...

Menülere başka alt menüler de eklerken bu mantık çok daha kolay anlaşılıyor. Şöyle ki:

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

from Tkinter import *

pencere = Tk()

menu = Menu(pencere)
pencere.config(menu=menu)
dosya = Menu(menu, tearoff=0)
menu.add_cascade(label="Dosya",menu=dosya)
dosya.add_command(label="Aç")
dosya.add_command(label="Kaydet")
dosya.add_command(label="Farklı Kaydet...")
dosya.add_command(label="Çıkış",command=pencere.quit)
yeni = Menu(dosya,tearoff=0)
dosya.add_cascade(label="Yeni",menu=yeni)
yeni.add_command(label="Metin Belgesi")
yeni.add_command(label="Resim Dosyası")
yeni.add_command(label="pdf dokümanı")

mainloop()
```

Gördüğünüz gibi, bu kez **Menu** pencere aracımızı ilk olarak **dosya** adlı araç üzerine bağlıyoruz. Çünkü yeni pencere aracımız, bir önceki pencere aracı olan **dosya**'nın üzerinde oluşturulacak. Bir sonraki satırda **add_command()** metodunu kullanırken de alt menüleri **yeni** adlı **Menu** pencere aracı üzerine bağlıyoruz. Çünkü bu alt menüler **yeni**'nin içinde yer alacak...

Aynı şekilde eğer Dosya başlığının yanına bir de mesela Düzen diye bir seçenek eklemek istersek, kodlarımız arasına şöyle bir bölüm ekliyoruz:

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

from Tkinter import *

pencere= Tk()

menu= Menu(pencere)
pencere.config(menu=menu)
dosya= Menu(menu, tearoff=0)
menu.add_cascade(label="Dosya",menu=dosya)
dosya.add_command(label="Aç")
dosya.add_command(label="Kaydet")
dosya.add_command(label="Farklı Kaydet...")
dosya.add_command(label="Çıkış",command=pencere.quit)
yeni= Menu(dosya,tearoff=0)
dosya.add_cascade(label="Yeni",menu=yeni)
yeni.add_command(label="Metin Belgesi")
yeni.add_command(label="Resim Dosyası")
yeni.add_command(label="pdf dokümanı")
dosya2= Menu(menu,tearoff=0)
menu.add_cascade(label="Düzen",menu=dosya2)
dosya2.add_command(label="Bul")

mainloop()
```

Text Pencere Aracı

Şimdiye kadar bir pencerenin sahip olması gereken pek çok özelliği gördük. Hatta pencerelerimize menüler dahi ekledik. Bütün bunların dışında öğrenmemiz gereken çok önemli bir pencere aracı daha var. O da **`Text()`** adlı pencere aracıdır. Bu araç sayesinde birden fazla satır içeren metinler oluşturabileceğiz. En basit haliyle **`Text()`** adlı pencere aracını şu şekilde oluşturabiliriz:

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

from Tkinter import *

pencere= Tk()
```

```
metin= Text()
metin.pack()

mainloop()
```

Oluşturduğumuz bu **`Text()`** aracı pek çok işlevi yerine getirebilecek durumdadır. Bu araç içine şu haliyle istediğimiz uzunlukta metin girebiliriz, klavye ve fareyi kullanarak metni yönetebiliriz, hatta oluşturduğumuz bu **`Text()`** aracını oldukça basit bir metin editörü olarak da kullanabiliriz. Eğer oluşturduğumuz bu **`Text()`** aracı içine öntanımlı olarak herhangi bir metin yerleştirmek istersek, şu kodu kullanmamız gerekir:

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

from Tkinter import *

pencere= Tk()

metin= Text(fg = "blue",font="Helvetica 13 bold")
metin.insert(END,"Sürüm 0.1.1")
metin.pack()

mainloop()
```

Gördüğünüz gibi **`Text`** aracını oluştururken, önceki yazılarda öğrendiğimiz şekilde fg seçeneği yardımıyla metni mavi yaptık. **`font`** seçeneği yardımıyla ise yazı tipini, "*Helvetica, 13, koyu ve altı çizili*" olarak belirledik.

Kodlarımız içinde kullandığımız **`metin.insert`** ifadesi de bize öntanımlı bir metin girme imkanı sağladı. Parantez içinde belirttiğimiz **`END`** ifadesi öntanımlı olarak yerleştireceğimiz metnin pencere aracının neresinde yer alacağını gösteriyor.

Yukarıda verdiğimiz kodu değiştirerek isterseniz daha çekici bir görünüm de elde edebilirsiniz:

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

from Tkinter import *

pencere= Tk()
a= "Sürüm 0.1.1"
metin= Text(bg="orange",fg = "blue",
```

```
   font="Helvetica 13 bold")
metin.insert(END,a.center(112,"*"))
metin.pack()

mainloop()
```

Eğer bir metin içinden herhangi bir bölümü almak isterseniz kullanmanız gereken şey **get** ifadesidir. Bu **get** ifadesinin nasıl kullanıldığını görelim şimdi:

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

from Tkinter import *

pencere= Tk()

metin= Text()
metin.pack()
metin.get(1.0,END)

mainloop()
```

Yukarıdaki örneği sadece **get** ifadesinin nasıl kullanıldığını göstermek için verdik. Şu haliyle bu kod bizim beklentilerimizi karşılayamaz. Çünkü **get** ifadesi yardımıyla metni aldık, ama aldığımız bu metni kullanmamızı sağlayacak araçları henüz penceremize yerleştirmediğimiz için **get** ifadesinin bize sağladığı işlevi kullanamıyoruz. Şimdilik burada şuna dikkat edelim: **metin.get()** gibi bir ifade kullanırken parantez içinde belirttiğimiz ilk sayı 1.0. Bu rakam metin kutusunun ilk satırının ilk sütununa işaret ediyor. Burada ilk satırın 1'den; ilk sütunun ise 0'dan başladığına dikkat edelim. Virgülden sonra gelen **END** ifadesi ise **Text** aracı içindeki metnin en sonuna işaret ediyor. Yani bu koda göre **get** ifadesi yardımıyla **Text** aracı içindeki bir metni, en başından en sonuna kadar alabiliyoruz. İsterseniz parantez içinde farklı sayılar belirterek, alınacak metnin ilk ve son konumlarını belirleyebiliriz. Şu koda bir bakalım:

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

from Tkinter import *

pencere= Tk()

metin= Text()
metin.pack()
```

```
metin.get(1.0,1.5)

mainloop()
```

Burada ise **`Text()`** aracının birinci satırı ve birinci sütunundan, birinci satırı ve beşinci sütununa kadar olan aralıktaki metin alınacaktır.

Şimdi henüz hiçbir iş yapmayan bu kodları biraz işlevli bir hale getirelim:

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

from Tkinter import *

pencere= Tk()

def al():
 a= metin.get(1.0,END)
 giris.insert(0, a)

metin= Text()
metin.pack()

btn= Button(text="al", command=al)
btn.pack()

giris= Entry()
giris.pack()

mainloop()
```

Burada öncelikle **`Text()`** aracı içindeki metnin tamamını alıp (**`metin.get(1.0,END)`**), **`giris`** adlı pencere aracına yerleştiren (**`giris.insert(0,a)`**) bir fonksiyon oluşturduk. Dikkat ederseniz kullanım kolaylığı açısından **`metin.get(1.0,END)`** ifadesini **`a`** adlı bir değişkene atadık.

Daha sonra metin adlı **`Text()`** aracımızı ve **`btn`** adlı **`Button()`** aracımızı oluşturduk. **`Button()`** aracımıza **komut** (*command*) olarak yukarıda tanımladığımız fonksiyonu göstererek **`Button`** ile fonksiyon arasında ilişki kurduk.

En sonunda da **`giris`** adlı **`Entry`** aracımızı tamamlayarak kodumuzu sona erdirdik.

Bu kodları çalıştırdığımızda karşımıza çıkan boş metin kutusuna herhangi bir şey yazıp alttaki düğmeye basınca, metin kutusunun bütün içeriği düğmenin hemen altındaki küçük metin kutusuna işlenecektir.

Şimdi daha karmaşık bir örnek yapalım...

Aşağıdaki örneği dikkatlice inceleyin:

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

from Tkinter import *

pencere= Tk()

def al():
 metin2.insert(END,metin.get(2.0,END))

a= "Sürüm 0.1.1"

metin= Text(height=15,bg="black",
   fg = "white",font="Helvetica 13 bold")
metin.insert(END,a.center(112,"*"))
metin.pack()

metin2= Text(height=15,width=115,
    bg="light blue",fg="red")
metin2.pack()

btn= Button(text="al",command=al)
btn.pack()

mainloop()
```

Yukarıdaki kodlarda bir metni ve pencere araçlarını nasıl biçimlendirdiğimize dikkat edin. Ayrıca Python'da karakter dizilerine ait bir metot olan **`center()`** yardımıyla bir kelimenin soluna ve sağına nasıl başka karakterler yerleştirdiğimizi inceleyin. Bu kodlar içinde kendinize göre bazı denemeler yaparak neyin ne işe yaradığını daha iyi anlayabilirsiniz.

Yukarıda bahsettiğimiz metnin koordinatlarını verme yöntemi her zaman tercih edilecek bir durum değildir. Ne de olsa kullanıcılarınızdan satır/sütun saymasını bekleyemezsiniz! Herhalde bu gibi durumlarda en iyi yöntem seçilen metnin alınması olacaktır. Bunu da basitçe şu kodlar yardımıyla yapıyoruz:

```
metin.get("sel.first","sel.last")
```

Burada **`sel.first`** ifadesi seçimin başlangıç noktasını; **`sel.last`** ifadesi ise seçimin bitiş noktasını gösteriyor.

Şimdi bu kod parçasını bir bağlam içinde kullanalım:

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

from Tkinter import *

pencere= Tk()

def al():
 a= metin.get("sel.first","sel.last")
 metin2.insert(END,a)

metin= Text(height=10,width=35,bg="white",
   fg ="blue",font="Helvetica 13 bold")
metin.pack()

metin2= Text(height=10,width=50,
    bg="black",fg="white")
metin2.pack()

btn= Button(text="al",command=al)
btn.pack()

mainloop()
```

Bu kodları çalıştırdığımızda, üstteki kutuya yazdığımız metnin istediğimiz bir kısmını seçtikten sonra alttaki düğmeye basarsak, seçili kısım ikinci kutuya işlenecektir.

Scrollbar Pencere Aracı

`Scrollbar()` adlı pencere aracı, Tkinter ile yazdığımız uygulamalara kaydırma çubuğu ekleme imkânı veriyor. Bu pencere aracının özelliği, öteki pencere araçlarının aksine tek başına kullanılmamasıdır. Kaydırma çubuğu kavramının mantığı gereği, bu pencere aracını başka pencere araçları ile birlikte kullanacağız. Mesela; **`Text`** pencere aracılığıyla bir metin kutusu oluşturmuş isek, şimdi öğreneceğimiz **`Scrollbar()`** adlı pencere aracı sayesinde bu metin kutusuna bir kaydırma çubuğu ekleyebileceğiz.

İsterseniz hemen ufak bir örnek yapalım:

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

from Tkinter import *

pencere = Tk()
```

Buraya kadar olan kısımda bir yenilik yok. Artık ezbere bildiğimiz komutlar bunlar. Devam ediyoruz:

```
kaydirma = Scrollbar(pencere)
kaydirma.pack(side=RIGHT,fill=Y)
```

Gördüğünüz gibi **`Scrollbar`** adlı pencere aracımızı oluşturup, bunu pack yöneticisiyle paketledik. **`pack()`** içindeki argümanlara dikkat edin. **`side=RIGHT`** ifadesi yardımıyla kaydırma çubuğumuzu sağa yaslarken, **`fill=Y`** ifadesi yardımıyla da **`Y`** düzlemi üzerinde (yani yukarıdan aşağıya doğru) boylu boyunca uzatıyoruz. Devam edelim:

```
metin = Text(yscrollcommand=kaydirma.set)
metin.pack()
```

Kaydırma çubuğunu bir metin kutusu içinde kullanacağımız için **`Text`** pencere aracı vasıtasıyla metin kutumuzu tanımlıyoruz. Burada **`yscrollcommand=kaydirma.set`** gibi bir ifade kullandığımıza dikkat edin. Bu ifade ile kaydırma işlemini etkinleştiriyoruz. Şimdi son darbeyi vurabiliriz:

```
kaydirma.config(command=metin.yview)

mainloop()
```

Burada **`kaydirma.config()`** satırını, metin aracını tanımladıktan sonra yazmamız önemli. Eğer metin aracını tanımlamadan **`config`** satırını yazarsak, komut satırında hataların uçuştuğunu görürüz...

Dilerseniz yukarıda parça parça verdiğimiz uygulamayı bir de topluca görelim:

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

from Tkinter import *

pencere = Tk()

kaydirma = Scrollbar(pencere)
kaydirma.pack(side=RIGHT,fill=Y)

metin = Text(yscrollcommand=kaydirma.set)
metin.pack()

kaydirma.config(command=metin.yview)

mainloop()
```

Böylece metin kutusuna yazdığımız yazılar aşağıya doğru uzadıkça sağ tarafta bir kaydırma çubuğu görüntülenecektir.

Peki, kaydırma çubuğunun, ekranın alt tarafında görünmesini istersek ne yapacağız? Yani yukarıdaki örnekte yaptığımız gibi, aşağıya doğru giden metinleri değil de, yana doğru giden metinleri kaydırmak istersek ne yapmalıyız? Çok kolay. Adım adım gidelim yine:

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

from Tkinter import *

pencere = Tk()
```

Devam ediyoruz…

```
kaydirma = Scrollbar(pencere, orient=HORIZONTAL)
kaydirma.pack(side=BOTTOM,fill=X)
```

Gördüğünüz gibi kaydırma çubuğunun pozisyonunu belirlemek için yeni bir seçenekten yararlanıyoruz. Bu yeni seçeneğin adı **`orient`**. Buna vermemiz gereken değer ise **`HORIZONTAL`**. Bu İngilizce kelime Türkçe'de **yatay** anlamına geliyor. Kaydırma çubuğumuz ekranda düşey değil, yatay vaziyette görüneceği için bu seçeneği kullandık. Ayrıca **`pack`** parametrelerine de dikkat edin. **`side`** seçeneğine

BOTTOM değerini verdik. Çünkü kaydırma çubuğumuzun ekranın alt tarafında görünmesini istiyoruz. Bir önceki örnekte, çubuğun ekranın sağında görünmesini istediğimiz için **side=RIGHT** şeklinde bir kullanım benimsemiştik (**RIGHT** kelimesi sağ anlamına gelirken, **BOTTOM** kelimesi dip-alt gibi anlamlara gelmektedir). Bu arada **fill** seçeneğinin değerinin de, bir önceki örneğin aksine, **X** olduğuna dikkat edin. Çünkü bu defa kaydırma çubuğumuzun **x** düzlemi üzerinde (yani soldan sağa doğru) boylu boyunca uzanması gerekiyor. Devam ediyoruz:

```
metin = Text(wrap=NONE, xscrollcommand=kaydirma.set)
metin.pack()
```

Şimdi de metin kutumuzu tanımladık. Buradaki parametrelere dikkat edin. Öncelikle metin kutumuza **wrap** adlı bir seçenek tanımlıyoruz. Bu **wrap** seçeneği metin kutusuna yazılan yazıların, ekranın sağ sınırına ulaşınca bir satır kaydırılıp kaydırılmayacağını belirliyor. Varsayılan olarak, yazılan metin ekranın sağ sınırına dayandığında otomatik olarak alt satıra geçecektir. Ama eğer biz bu **wrap** seçeneğine **NONE** değerini verirsek, metin sınırı geçip sağa doğru uzamaya devam edecektir. İşte aynı parantez içinde belirttiğimiz **xscrollcommand=kaydirma.set** seçeneği sayesinde sağa doğru uzayan bu metni kaydırabileceğiz. Burada, yukarıdaki örnekten farklı olarak **xscrollcommand** ifadesini kullandığımıza dikkat edin. Bir önceki örnekte **yscrollcommand** ifadesini kullanmıştık. Bu farkın nedeni; bir önceki örnekte **y** düzlemi üzerinde çalışırken, şimdiki örnekte **x** düzlemi üzerinde çalışıyor olmamız...

Artık son darbeyi vurabiliriz:

```
kaydirma.config(command=metin.xview)

mainloop()
```

Gördüğünüz gibi burada da bir önceki örnekten farklı olarak **xview** ifadesini kullandık. Bir önceki örnekte **yview** ifadesini kullanmıştık. Bu farkın nedenini söylemeye gerek yok...

Gelin isterseniz bu parçalı örneği bir araya getirelim:

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

from Tkinter import *

pencere = Tk()

kaydirma = Scrollbar(pencere, orient=HORIZONTAL)
kaydirma.pack(side=BOTTOM,fill=X)

metin = Text(wrap=NONE, xscrollcommand=kaydirma.set)
metin.pack()

kaydirma.config(command=metin.xview)

mainloop()
```

TKINTER UYGULAMALARINI GÜZELLEŞTİRMEK

5

Tkinter arayüz takımı konusunda en fazla şikayet edilen şey; Tkinter ile yazılan uygulamaların çirkin görünmesidir. Bu bölümde bu sıkıntıyı aşmanın bazı yollarını göstermeye çalışacağız. Ayrıca bu bölümde, güzellik ile ilgili olduğunu düşündüğüm başka konulara da değineceğiz. Mesela bu bölümde, Tkinter uygulamaları içinde kullandığımız pencere araçlarına nasıl simge yerleştirebileceğimizi de inceleyeceğiz. Bu bölüm bir yandan da örnek uygulamalarla zenginleştirilmeye çalışılacaktır. Yani konuları anlatırken ufak program örnekleri üzerinden gitmeye çalışacağız.

Tkinter Programlarının Renk Şemasını Değiştirmek

Tkinter programlarını güzelleştirme yolunda ilk inceleyeceğimiz konu; yazdığımız uygulamaların renk şemasını değiştirmek olacaktır. Biz herhangi bir Tkinter uygulaması yazdığımız zaman, bu uygulamanın varsayılan bir renk şeması olacaktır. Hemen bir örnekle başlayalım:

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

from Tkinter import *

pencere = Tk()

etiket = Label(text="Örnek Uygulama")
etiket.pack()

mainloop()
```

Buna benzer bütün uygulamalarımızda belli bir renk şeması varsayılan olarak pencerelerimize uygulanacaktır. Eğer biz müdahale etmezsek, oluşturacağımız bütün pencereler ve üzerindeki pencere araçları tek bir renkte görünecektir. Ama eğer istersek bu gidişe bir dur diyebiliriz. Bunu nasıl yapabileceğimizi daha önceki derslerimizde de kısmen görmüştük. Mesela şu seçenekler bize yardımcı oluyordu:

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

from Tkinter import *

pencere = Tk()

etiket = Label(text="Örnek Uygulama",
      fg="white",
      bg="blue")

etiket.pack()

mainloop()
```

Burada, kullandığımız etiket (**`Label()`**) adlı pencere aracını **`bg`** ve **`fg`** seçenekleri yardımıyla farklı renklerde çizdik. Siz renk adları yerine *http://www.html-color-codes.info/* adresine başvurarak, renk kodlarını da kullanabileceğinizi biliyorsunuz.

Aslında benim kastettiğim güzelleştirme operasyonu sadece pencere araçlarının rengini değiştirmekle ilgili değil. Aynı zamanda, penceremizin rengini değiştirmekten de bahsediyorum. İsterseniz hemen bir örnek vererek ne demek istediğimizi biraz daha net olarak anlatmaya çalışayım. Şu örneğe bir bakın:

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

from Tkinter import *

pencere = Tk()

pencere.tk_setPalette("#D0A9F5")

etiket = Label(text="Renk şeması değiştirilmiş bir örnek uygulama")
etiket.pack()

mainloop()
```

Gördüğünüz gibi sadece tek bir pencere aracının rengini değil, aynı zamanda bütün bir pencerenin rengini değiştirdik. Bunu yapmamızı sağlayan şey de kodlar içinde kullandığımız, **`pencere.tk_setPalette("#D0A9F5")`** satırı oldu. Yani buradaki **`tk_setPalette()`** adlı metodu kullanarak bütün pencerenin renk şemasını değiştirebildik. Bu **`tk_setPalette()`** komutu, **`Tk()`** sınıfının metotlarından yalnızca biridir. Bu sınıfın bize hangi metotları sunduğunu görmek için Python'un etkileşimli kabuğunda şu komutları verebilirsiniz:

```
from Tkinter import *
dir(Tk())
```

Bu komutun çıktısı olarak elde edilen liste alfabe sırasına göre dizilmiştir. Dolayısıyla bu listedeki **`tk_setPalette`** metodunu rahatlıkla bulabilirsiniz...

Gördüğünüz gibi **`Tk()`** sınıfının bir yığın metodu var. İşte **`tk_setPalette()`** metodu da bunlardan biri olup, Tkinter pencerelerinin renk şemasını değiştirmemizi sağlıyor.

Şimdi şu örneğe bakalım:

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

from Tkinter import *

pencere = Tk()

pencere.tk_setPalette("#D0A9F5")

etiket = Label(text="Hangi GNU/Linux Dağıtımını Kullanıyorsunuz?")
etiket.pack(pady=10,padx=10)

liste = Listbox()
liste.pack(side=LEFT)

GNULinux = ["Debian","Ubuntu","Kubuntu","RedHat"]

for i in GNULinux:
 liste.insert(END,i)

dugme = Button(text="KAPAT",command=pencere.quit)
dugme.pack()

mainloop()
```

Gördüğünüz gibi, **`tk_setPalette()`** metodu ile pencerenin renk şemasını değiştirdiğimizde, bundan pencere ile birlikte üzerindeki bütün pencere araçları da etkileniyor. Peki, biz pencere araçlarının pencereden farklı bir renkte olmasını istersek ne yapacağız? Çok basit:

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

from Tkinter import *

pencere = Tk()

pencere.tk_setPalette("#D0A9F5")

etiket = Label(text="Hangi GNU/Linux Dağıtımını Kullanıyorsunuz?",
      fg="#B4045F")

etiket.pack(pady=10, padx=10)

liste = Listbox()
liste.pack(side=LEFT)

GNULinux = ["Debian","Ubuntu","Kubuntu","RedHat"]

for i in GNULinux:
 liste.insert(END,i)

dugme = Button(text="KAPAT",
      bg="#610B5E",
      fg="white",
      command=pencere.quit)

dugme.pack()

mainloop()
```

Gördüğünüz gibi farklı renkte olmasını istediğimiz pencere araçlarının renklerini, daha önce öğrendiğimiz seçenekler yardımıyla açık açık belirtiyoruz. Tabii sizin benden daha zevkli renk kombinasyonları seçeceğinizi tahmin ediyorum.

Böylelikle bir Tkinter programının renk şemasını nasıl değiştireceğimizi öğrenmiş olduk. Bir sonraki bölümde Tkinter uygulamaları üzerinde resimleri/simgeleri nasıl kullanabileceğimizi göreceğiz. Neticede resimler ve simgeler de bir programı güzelleştiren öğelerdir.

Pencere Araçlarına Simge Eklemek

Şimdi Tkinter'de yazdığımız programlara nasıl simge ekleyebileceğimizi göreceğiz. Bu iş için **`ImageTk`** adlı modülü kullanacağız. Bu modülü kullanmak için **`python-imaging`** adlı paketi yüklemeniz gerekecektir. GNU/Linux kullanıcıları kendi dağıtımlarının paket yöneticisini kullanarak gerekli modülü kurabilirler. Windows kullanıcıları ise *http://www.pythonware.com/products/pil/* adresini ziyaret ederek, kullandıkları Python sürümüne uygun programı indirebilir.

Gelelim asıl konumuza. Öncelikle aşağıdaki gibi bir başlangıç yapalım:

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

from Tkinter import *
import ImageTk
```

Böylelikle gerekli bütün modülleri içe aktarmış olduk. Burada ImageTk adlı modülü de içe aktardığımıza dikkat edin. Devam ediyoruz.

```
pencere = Tk()

simge = ImageTk.PhotoImage(file="simge.png")
```

Yukarıdaki kodlar yardımıyla önce bir pencere oluşturduk her zamanki gibi. Ardından da, programın en başında içe aktardığımız **`ImageTk`** adlı modülün **`PhotoImage`** adlı metodundan ve file adlı seçeneğinden yararlanarak, bilgisayarımızdaki **simge.png** adlı dosyayı programımız içinde kullanılabilir hale getirdik. Dikkat edin, "kullandık" diyemiyoruz; "kullanılabilir hale getirdik" diyoruz. Yani kabaca ifade etmek gerekirse, bu şekilde bir resim nesnesi oluşturmuş oluyoruz. Devam edelim:

```
dugme = Button(text="Resim1",
      image=simge,
      compound="top")
dugme.pack()

mainloop()
```

Burada bildiğimiz şekilde **`Button`** pencere aracını kullanarak bir düğme oluşturduk. Ama burada bilmediğimiz birkaç şey var.

Hemen anlatalım: Öncelikle **image** adlı yeni bir seçenek görüyoruz kodlarımız arasında. Buraya, yukarıda kullanılabilir hale getirdiğimiz resim nesnesini yerleştiriyoruz. Bu seçeneğin ardından, **compound** adlı başka bir seçenek daha geliyor. Bu da **text** ve **image** seçenekleri ile tanımlanan öğelerin pencerede nasıl görüneceğini belirliyor. Burada kullandığımız **top** değeri, **image** seçeneğiyle gösterilen öğenin text seçeneği ile gösterilen öğenin üstünde yer alacağını ifade ediyor. Bu **compound** seçeneği dört farklı değer alabilir:

1. **top**: resim metnin üstünde
2. **bottom**: resim metnin altında
3. **right**: resim metnin sağında
4. **left**: resim metnin solunda

Biz örneğimizde top değerini tercih ettik. Tabii ki siz bu dört seçenek içinden istediğinizi kullanabilirsiniz. Yukarıdaki kodlarda görünen **pack()** ve **mainloop()** artık adımız soyadımız gibi bildiğimiz şeyler. O yüzden üzerinde durmuyoruz.

Şimdi isterseniz bu kodları topluca görelim:

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

from Tkinter import *
import ImageTk

pencere = Tk()

simge = ImageTk.PhotoImage(file=”simge.png”)

dugme = Button(text=”Resim1”,
      image=simge,
      compound=”top”)
dugme.pack()

mainloop()
```

Böylece Tkinter'de resimleri ve metinleri bir arada nasıl kullanacağımızı öğrenmiş olduk.

Şimdi gelin isterseniz yukarıdaki bilgilerimizi kullanarak örnek bir arayüz yazalım:

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

from Tkinter import *
import ImageTk

pencere = Tk()
pencere.tk_setPalette("#CECEF6")

cerceve1 = Frame()
cerceve1.grid(row=0, column=0, pady=5, padx=5)

cerceve2 = Frame()
cerceve2.grid(row=1, column=0, pady=10, padx=10)

simge1 = ImageTk.PhotoImage(file="simge1.png")
simge2 = ImageTk.PhotoImage(file="simge2.png")
simge3 = ImageTk.PhotoImage(file="simge3.png")
simge4 = ImageTk.PhotoImage(file="simge4.png")

etiket = Label(cerceve1,
      text="Kendinize Uygun bir Simge Seçiniz...",
      fg="#610B0B",
      font="Verdana 10 bold")

etiket.pack()

dugme1 = Button(cerceve2,
    text="Resim1",
    image=simge1,
    compound="top")

dugme1.grid(row=3, column=0, padx=6)

dugme2 = Button(cerceve2,
    text="Resim2",
    image=simge2,
    compound="top")

dugme2.grid(row=3, column=3, padx=6)

dugme3 = Button(cerceve2,
    text="Resim3",
    image=simge3,
    compound="top")
```

```
dugme3.grid(row=3, column=6, padx=6)

dugme4 = Button(cerceve2,
    text="Resim4",
    image=simge4,
    compound="top")

dugme4.grid(row=3, column=9, padx=6)
```

Bu programı çalıştırdığımızda şöyle bir görüntü elde ediyoruz:



Tabii bu programın düzgün çalışması için resimlerinizle program dosyasını aynı klasöre koymayı unutmuyorsunuz. Eğer resimlerle program dosyası ayrı klasörlerde duracaksa simge öğelerini belirlerken resimlerin bulunduğu klasör adlarını tam olarak yazmaya dikkat etmelisiniz.

Burada düğmeleri konumlandırmak için **`pack()`** geometri yöneticisi yerine **`grid()`** adlı geometri yöneticisini kullandığımıza dikkat edin. Öğeleri bu şekilde sıralamak için **`grid()`** yöneticisi daha uygundur. Ayrıca yukarıdaki kodlarda iki farklı **`Frame`** oluşturduğumuza da dikkat edin. **`Label`** adlı pencere aracımızı bir **`Frame`** üzerine, geri kalan pencere araçlarımızı (yani düğmelerimizi) ise öteki **`Frame`** üzerine yerleştirdik. İki farklı **`Frame`** kullanmamız sayesinde aynı program içinde hem **`grid()`** hem de **`pack()`** adlı geometri yöneticilerini bir arada kullanma imkanı elde ettik. İki farklı Frame kullanmamız aynı zamanda bize, pencere araçlarımızı daha özgür bir şekilde konumlandırma imkanı da sağladı. Ayrıca **`grid()`** yöneticisinin padx seçeneğini kullanarak pencere araçlarının birbirlerine olan uzaklıklarını da ayarlayabildik. Yukarıdaki kodlarla ilgili olarak anlamadığınız noktalar olması durumunda bana nasıl ulaşabileceğinizi biliyorsunuz.

Böylelikle bu konuyu da bitirmiş olduk. Bir sonraki bölümde Tkinter ile yazdığımız pencere araçlarına nasıl **ipucu metni** (*tooltip*) ekleyeceğimizi göreceğiz.

Pencere Araçlarına İpucu Metni (Tooltip) Eklemek

Fareyi pencere araçlarının üzerine getirdiğimizde ekranda beliren ve o aracın ne işe yaradığını kısaca açıklayan ufak metin parçalarına **ipucu metni** diyoruz. İşte bu bölümde Tkinter'deki pencere araçlarına nasıl ipucu metni ekleyeceğimizi öğreneceğiz. Öncelikle şunu söyleyelim: Tkinter böyle bir özelliği bize kendiliğinden sunmuyor. Ama hem Python'un hem de Tkinter'in en güçlü yanlarından biri, dilde olmayan bir özelliği üçüncü parti yazılımlar aracılığıyla dile ekleyebilmemizdir. Pencere araçlarına ipucu metni ekleyebilmek için bu üçüncü parti yazılımlardan birini kullanacağız. Şu noktada yapmamız gereken ilk iş; *http://svn.effbot.org/public/stuff/sandbox/wcklib/* adresinden **`wckToolTips`** adlı modülü bilgisayarımıza indirmek olacaktır. Bu modül pencere araçlarına ipucu metni eklememizi sağlayacak. Bu modülü, içinde ipucu metni kullanacağımız programlarımızla aynı klasör içinde tutmamız gerekiyor. Örneğin; diyelim ki falanca.py adında bir program yazdık. Bu programın **FALANCA** adlı klasör içinde yer aldığını varsayarsak, **`wckToolTips`** modülünü **falanca.py** ile aynı klasör içine, yani **FALANCA** adlı klasöre atmamız gerekiyor. Bu sayede **`wckToolTips`** modülünü programımız içine aktarırken (*import*) herhangi bir sorun yaşamayacağız. Bununla ilgili ufak bir örnek yapalım:

```
from Tkinter import *
import wckToolTips
```

Böylece Tkinter modülüyle birlikte **`wckToolTips`** modülünü de içe aktarmış olduk. Python, biz bu modülü içe aktarmaya çalıştığımızda, öncelikle programımızın içinde bulunduğu dizini kontrol ederek, **wckToolTips.py** adlı bir dosya olup olmadığına bakacaktır. Eğer modül bu dizinde bulunamaz ise, Python **`sys.path`** çıktısında görünen dizinlerin içini de denetleyecektir. Eğer modül orada da yoksa, Python bize bir hata mesajı gösterecektir. Bu durumda, **`wckToolTips`** gibi harici bir modülü programın bulunduğu dizin içine atmak en kolay yol olacaktır. Neyse, biz örneğimize devam edelim.

```
pencere = Tk()

etiket = Label(text="Herhangi bir etiket")
etiket.pack()

dugme = Button(text="Tamam", command=pencere.destroy)
dugme.pack()
```

Burada normal bir şekilde, penceremizi tanımladık. Bunun yanında bir adet etiket ve bir adet düğme oluşturduk. Yeni bir şey yok... Devam edelim:

```
wckToolTips.register(dugme, "Programı kapat")

mainloop()
```

İşte **wckToolTips** modülü burada devreye giriyor. Bu modülün **register()** adlı metodunu kullanarak, dugme adlı pencere aracının üzerine fare ile geldiğimizde, ekranda görünmesini istediğimiz ipucu metnini oluşturuyoruz. Burada ipucu metnimiz şu: "*Programı kapat*"

Programımızı bütünüyle bir görelim bakalım:

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

from Tkinter import *
import wckToolTips

pencere = Tk()

etiket = Label(text="Herhangi bir etiket")
etiket.pack()

dugme = Button(text="Tamam", command=pencere.destroy)
dugme.pack()

wckToolTips.register(dugme, "Programı kapat")

mainloop()
```

Gördüğünüz gibi düğmenin üzerine fare ile geldiğimizde ipucu metnimiz ekranda görünüyor... İşte Tkinter'de ipucu metinlerini kullanmak bu kadar kolaydır. Kendi yazdığınız programlarda bu modülü kullanarak, programınızı kullanıcılar açısından daha cazip ve anlaşılır bir hale getirebilirsiniz. Gelin isterseniz bununla ilgili bir örnek daha yapalım:

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

from Tkinter import *
import wckToolTips
import time

pencere = Tk()

etiket = Label(text="Saat 17:00'da randevunuz var!")
etiket.pack(expand=YES, fill=BOTH)
```

```
def saat(pencere_araci, fonksiyon):
 return "Şu anda saat: %s"%time.strftime("%H:%M:%S")

dugme = Button(text="Tamam", command=pencere.destroy)
dugme.pack(expand=YES)

wckToolTips.register(pencere, saat)

mainloop()
```

Burada ipucu metnimizi doğrudan pencerenin kendisine atadık. Böylece fare ile pencere üzerine geldiğimizde o an saatin kaç olduğu ekranda görünecektir... Ayrıca burada **`register()`** metoduna argüman olarak bir fonksiyon atadığımıza dikkat edin. Yukarıda yazdığımız kodları şöyle yazarsak ne olur peki?

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

from Tkinter import *
import wckToolTips
import time

pencere = Tk()

etiket = Label(text="Saat 17:00'da randevunuz var!")
etiket.pack(expand=YES, fill=BOTH)

dugme = Button(text="Tamam", command=pencere.destroy)
dugme.pack(expand=YES)

wckToolTips.register(pencere, "Şu anda saat: %s" \
%time.strftime("%H:%M:%S"))

mainloop()
```

Bu şekilde programımız yine çalışır ve hatta saatin kaç olduğunu ekranda ipucu metni olarak da gösterir. Ama bir farkla: Bu yazım şeklinde ekranda görünen saat durağan olacaktır. Yani program kapatılıp tekrar açılana kadar hep aynı saati gösterecektir. Ama **`register()`** metoduna bir önceki örnekte olduğu gibi bir fonksiyon atarsak saat devinecek, yani pencere üzerine ikinci kez geldiğimizde ipucu metni saati güncellenmiş olarak verecektir.

Bu son kodlarda, ayrıca **`pack()`** geometri yöneticisinin **`expand`** ve **`fill`** seçeneklerini kullanarak pencere boyutlandırmaları sırasında, pencere araçlarının uzayıp kısalmasını sağladık. Bu kodları bir de bu seçenekler olmadan çalıştırırsanız, ne demek istediğim daha bariz bir şekilde ortaya çıkacaktır.

Son olarak, yukarıdaki kodlarda `saat()` adlı fonksiyonun iki argüman aldığına dikkat edin. Bu fonksiyonu argümansız olarak yazarsak, programımız çalışsa da ipucu metni görüntülenmeyecektir. Böyle bir durumda ipucu metninin görüntülenmemesine yol açan hatayı komut satırı çıktılarından takip edebilirsiniz.

NASIL YAPILIR?

6

Tkinter'de Fare ve Klavye Hareketleri (Events and Bindings)

Tkinter'de yazdığımız bir uygulamada, kullanıcının fare hareketlerini ve klavyede bastığı tuşları belli fonksiyonlara bağlamak ve bu hareketleri yakalamak bu bölümün konusunu oluşturuyor. Mesela; kullanıcı klavyedeki **ENTER** tuşuna bastığında programımızın özel bir işlemi yerine getirmesini sağlamak, kullanıcının hangi tuşa bastığını bulmak bu konunun kapsamına giriyor. Her zamanki gibi lafı hiç uzatmadan, bir örnekle derdimizi anlatmaya çalışalım. Diyelim ki şöyle bir şey var elimizde:

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

from Tkinter import *

pencere = Tk()

etiket = Label(text="Aşağıdaki kutucuğa adınızı yazıp \
Tamam düğmesine basın!\n")
etiket.pack()

giris = Entry()
giris.pack()

def kaydet():
 dosya = open("isimler_python.txt","a")
```

```
 veri = giris.get()
 veri = unicode.encode(unicode(veri),"utf8")
 dosya.write(veri+"\n")
 dosya.close()
 etiket["text"] = etiket["text"] + u"%s dosyaya başarıyla eklendi\
n"%giris.get()
 giris.delete(0,END)

dugme = Button(text="Tamam",command=kaydet)
dugme.pack()

mainloop()
```

Burada kullanıcı **Tamam** düğmesine bastığında **`kaydet()`** fonksiyonu içinde tanımlanan işlemler gerçekleştirilecektir. Bu arada yukarıdaki kodlarda Türkçe karakterleri alabilmek için nasıl bir yol izlediğimize dikkat edin. Dediğimiz gibi bu programın çalışması için kullanıcının Tamam düğmesine basması gerekiyor. Peki, biz aynı işlemin **ENTER** düğmesine basılarak da yapılabilmesini istersek ne olacak? Yani kullanıcının mutlaka Tamam düğmesine basmasını zorunlu kılmadan, klavyedeki **ENTER** tuşuna bastığında da kutucuğa yazılan isimlerin dosyaya eklenmesini nasıl sağlayacağız? İşte bunu yapabilmek için **`bind()`** fonksiyonundan yararlanmamız gerekiyor. Bu fonksiyon; bize klavye ve fare hareketlerini belli fonksiyonlara bağlama ve bu hareketleri yakalama imkanı verecek. Nasıl mı? Hemen görelim:

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

from Tkinter import *

pencere = Tk()

etiket = Label(text="Aşağıdaki kutucuğa adınızı yazıp \
Tamam düğmesine basın!\n")
etiket.pack()

giris = Entry()
giris.pack()

def kaydet(event=None):
 dosya = open("isimler_python.txt","a")
 veri = giris.get()
 veri = unicode.encode(unicode(veri),"utf8")
 dosya.write(veri+"\n")
```

```
 dosya.close()
 etiket["text"] = etiket["text"] + u"%s dosyaya başarıyla eklendi\
n"%giris.get()
 giris.delete(0,END)

dugme = Button(text="Tamam",command=kaydet)
dugme.pack()

pencere.bind("<Return>",kaydet)

mainloop()
Burada, önceki kodlarımıza yalnızca şu eklemeleri yaptık:
def kaydet(event=None):
 ....

....
pencere.bind("<Return>",kaydet)
...
```

`kaydet()` fonksiyonuna eklediğimiz **`event`** parametresi zorunlu bir adlandırma değildir. Hatta oraya kendi isminizi dahi yazdığınızda programın düzgün çalıştığını göreceksiniz. Ancak oraya bir parametre yazacağınız zaman herhangi bir kelime yerine **`event`** ifadesini kullanmanızı tavsiye ederim. Çünkü **`event`** tıpkı sınıflardaki "self" gibi kemikleşmiştir. Burada dikkat etmemiz gereken iki şey var: Birincisi, o fonksiyonu parametresiz bırakmamak; ikincisi, parametreye varsayılan değer olarak **`None`** vermek. Eğer bu **`None`** değerini vermezseniz, kullanıcı **ENTER** tuşuna bastığında program çalışır, ama Tamam düğmesine bastığında çalışmaz. Çünkü; Tkinter event parametresinin değerini **`command=kaydet`** şeklinde gösterdiğimiz koda otomatik olarak atamayacaktır. Bu yüzden, oraya varsayılan değer olarak **`None`** yazmazsak, **`kaydet()`** fonksiyonunun bir adet argüman alması gerektiğine dair bir hata alırız.

Kodlarımıza yaptığımız ikinci ekleme, **`bind()`** fonksiyonudur. Burada dikkat etmemiz gereken birkaç şey var. Öncelikle bu **`bind()`** fonksiyonunu pencere ile birlikte kullandığımıza dikkat edin. Bu satırla aslında Tkinter'e şu komutu vermiş oluyoruz:

"Ey Tkinter! 'pencere' seçili iken, yani etkinken, kullanıcı RETURN düğmesine (veya başka bir söyleyişle ENTER düğmesine) bastığında kaydet() adlı fonksiyonu çalıştır!"

Artık kullanıcımız, kutucuğa bir isim yazıp **ENTER** düğmesine bastığında, yazdığı isim dosyaya eklenecektir.

Gördüğünüz gibi, eğer klavyedeki **ENTER** düğmesini bağlamak istiyorsak, kullanmamız gereken ifade **<Return>**. Peki, başka hangi ifadeler var? Mesela **ESCAPE** için bir örnek verelim. Yukarıdaki kodlara ufacık bir ekleme yapıyoruz:

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

from Tkinter import *
import sys

pencere = Tk()

etiket = Label(text="Aşağıdaki kutucuğa adınızı yazıp \
Tamam düğmesine basın!\n")
etiket.pack()

giris = Entry()
giris.pack()

def kaydet(event=None):
 dosya = open("isimler_python.txt","a")
 veri = giris.get()
 veri = unicode.encode(unicode(veri),"utf8")
 dosya.write(veri+"\n")
 dosya.close()
 etiket["text"] = etiket["text"] + u"%s dosyaya \
başarıyla eklendi\n"%giris.get()
 giris.delete(0,END)

dugme = Button(text="Tamam", command=kaydet)
dugme.pack()

pencere.bind("<Return>", kaydet)
pencere.bind("<Escape>", sys.exit)

mainloop()
```

Gördüğünüz gibi, klavyedeki **ESCAPE** düğmesini kullanabilmek için kodlarımız içine **<Escape>** ifadesini eklememiz yeterli oluyor. Peki, klavyedeki öbür düğmelerin karşılıkları nelerdir?

Listeleyelim:

- **F1:** <F1> (Bütün F'li tuşlar aynı şekilde kullanılabilir)
- **Num Lock:** <Num_Lock>
- **Scroll Lock:** <Scroll_Lock>
- **Backspace:** <BackSpace>
- **Delete:** <Delete>
- **Sol Ok:** <Left>
- **Sağ Ok:** <Right>
- **Aşağı Ok:** <Down>
- **Yukarı Ok:** <Up>
- **Alt Düğmesi:** <Alt_L>
- **Ctrl Düğmesi:** "<Control_L>
- **Shift Düğmesi:** <Shift_L>
- **Ctrl + S:** <Control-s> (Öteki harfler de aynı şekilde kullanılabilir)
- **Shift + S:** <Shift-s> (Öteki harfler de aynı şekilde kullanılabilir)
- **Alt + S:** <Alt-s> (Öteki harfler de aynı şekilde kullanılabilir)
- **Boşluk Düğmesi:** <space>
- **Print Düğmesi:** <Print>

Klavyedeki harfleri pencere araçlarına bağlamak için ise, doğrudan o harfin adını kullanabiliriz. Mesela; bir işlemin, **`f`** harfine basıldığında gerçekleşmesini istersek, yazmamız gereken ifade **`f`** olacaktır. Bunun **`<f>`** değil, sadece **`f`** şeklinde olduğuna dikkat edin. Tabii burada Türkçe'ye özgü harfleri kullanmamaya dikkat ediyoruz.

Klavye hareketlerini bağlamayı öğrendik. Peki, ya fare hareketlerini nasıl bağlayacağız? O da çok basittir. Burada da yine **`bind()`** fonksiyonundan yararlanacağız. Gelin isterseniz şöyle bir uygulama yazalım:

```
# -*- coding: utf-8 -*-
from Tkinter import *

pencere = Tk()
pencere.geometry("400x200")
```

```
etiket = Label(text="Farenin pencere üzerindeki konumu:")
etiket.pack(anchor=NW)

girisx_etiket = Label(text="X:")
girisx_etiket.pack(side=LEFT, anchor=N)

girisx = Entry(width=5)
girisx.pack(side=LEFT,padx=15, anchor=N)

girisy_etiket = Label(text="Y:")
girisy_etiket.pack(side=LEFT, anchor=N)

girisy = Entry(width=5)
girisy.pack(side=LEFT, anchor=N)

def hareket(event=None):
 girisx.delete(0, END)
 girisy.delete(0, END)
 girisx.insert(0, event.x)
 girisy.insert(0, event.y)

pencere.bind("<Button-1>", hareket)

mainloop()
```

Yukarıdaki uygulamayı çalıştırıp pencere üzerinde herhangi bir yere farenin sol tuşu ile tıkladığımızda, tıkladığımız noktanın koordinatları (x ve y düzlemine göre) kutucuklar içinde görünecektir.

Bu kodları biraz açıklayalım:

Kodlar arasında gördüğünüz **`anchor`** ifadesi, **`pack()`** adlı pencere yöneticisinin seçeneklerinden biridir. Bu seçenek, ilgili pencere aracının pencere üzerinde bir noktaya çapa atmasını sağlar (İngilizce **anchor** kelimesi Türkçe'de **çapa atmak**, **demir atmak** gibi anlamlara gelir). Bu seçenek dört farklı değer alabilir:

- **`N`**: Kuzey anlamına gelen North kelimesinin kısaltmasıdır. Pencere aracının kuzey yönüne sabitlenmesini sağlar.
- **`S`**: Güney anlamına gelen South kelimesinin kısaltmasıdır. Pencere aracının güney yönüne sabitlenmesini sağlar.
- **`W`**: Batı anlamına gelen West kelimesinin kısaltmasıdır. Pencere aracının batı yönüne sabitlenmesini sağlar.
- **`E`**: Doğu anlamına gelen East kelimesinin kısaltmasıdır. Pencere aracının doğu yönüne sabitlenmesini sağlar.

Bu yönleri bir arada da kullanabiliriz. Örneğin; pencere aracının kuzeybatı yönünde sabitlenmesini istersek, kullanmamız gereken ifade **`NW`** olacaktır. Mesela; yukarıdaki kodlarda bunu kullandık. **`etiket.pack(anchor=NW)`** ifadesi yardımıyla, etiket adlı pencere aracımızın kuzeybatı yönüne çapa atmasını sağladık (Yani pencerenin üst-sol ucuna). Öteki pencere araçlarında sadece **`anchor=N`** ifadesini kullandık. Çünkü bu araçlardaki **`side=LEFT`** ifadesi, aracımızın sol yana yerleşmesini zaten sağlıyor. **`pack()`** yöneticisi içinde kullandığımız padx seçeneğini zaten biliyorsunuz. Bu seçenek yardımıyla pencere araçlarının dirsek mesafesini ayarlıyoruz...

Yukarıdaki kodlar içinde en önemli nokta tabii ki **`hareket()`** adlı fonksiyon. Burada parantez içinde **`event`** parametresini belirtmeyi unutmuyoruz. Kabaca ifade etmek gerekirse, bu parametre bizim örneğimizde fare hareketlerini temsil ediyor. Buna göre, fonksiyon içindeki **`event.x`** ve **`event.y`** ifadelerini, **farenin x düzlemi üzerindeki hareketi** ve **farenin y düzlemi üzerindeki hareketi** şeklinde düşünebiliriz. Bu arada, **`giris.delete()`** fonksiyonları yardımıyla, Entry aracının içini sürekli olarak boşalttığımıza dikkat edin. Eğer **`giris.delete()`** fonksiyonlarını yazmazsak, fare tıklaması ile bulduğumuz eski ve yeni koordinatlar birbirine karışacaktır.

`mainloop()` satırından hemen önce programımızdaki asıl işi yapan **`bind()`** fonksiyonunu görüyoruz. Farenin sol düğmesini **`Button-1`** ifadesinin temsil ettiğini görüyorsunuz. Peki, buna benzer başka neler var?

- **`Button-2`**: Farenin orta düğmesine (veya tekerine) basılması
- **`Button-3`**: Farenin sağ düğmesine basılması
- **`Motion`**: Farenin, hiç bir düğme basılı değilken hareket ettirilmesi
- **`B1-Motion`**: Farenin, sol düğme basılı halde hareket ettirilmesi
- **`B2-Motion`**: Farenin, orta düğme (veya teker) basılı halde hareket ettirilmesi
- **`B3-Motion`**: Farenin, sağ düğme basılı halde hareket ettirilmesi
- **`ButtonRelease-1`**: Farenin sol düğmesinin serbest bırakılması (yani düğmeye basıldığındaki değil, düğmenin bırakıldığındaki hali...)
- **`ButtonRelease-2`**: Farenin orta düğmesinin (veya tekerinin) serbest bırakılması
- **`ButtonRelease-3`**: Farenin sağ düğmesinin serbest bırakılması
- **`Double-Button-1`**: Farenin sol düğmesine çift tıklanması

- **Double-Button-2:** Farenin orta düğmesine (veya tekerine) çift tıklanması
- **Double-Button-3:** Farenin sağ düğmesine çift tıklanması
- **Enter:** Farenin pencere aracının üzerine gelmesi (Bunun **ENTER** tuşuna basmak anlamına gelmediğine dikkat edin.)
- **Leave:** Farenin pencere aracını terk etmesi

Sırada klavye hareketlerini yakalamak var. Yani şimdiki görevimiz, bir kullanıcının klavyede hangi tuşlara bastığını bulmak. Bunun için keysym niteliğinden yararlanacağız:

```
# -*- coding: utf-8 -*-
from Tkinter import *

pencere = Tk()
pencere.geometry("200x200")

etiket = Label(text="Basılan Tuş:\n",wraplength=150)
etiket.pack()

def goster(event=None):
 etiket["text"] += event.keysym

pencere.bind("<Key>",goster)

mainloop()
```

Yine kodlarımızı biraz açıklayalım.

Öncelikle, gördüğünüz gibi, etiket adlı pencere aracımız içinde **wraplength** adlı bir seçenek kullandık. Bu seçenek etikete yazdırılabilecek metnin uzunluğunu sınırlandırıyor. Biz bu değeri 150 piksel olarak belirledik. Buna göre, etikete yazılan metin 150 pikseli aştığında kendiliğinden bir alt satıra geçecektir.

Şimdi fonksiyonumuza bir göz atalım: Burada **event.keysym** adlı bir ifade görüyoruz. Bu ifade, klavye üzerindeki sembolleri, yani tuşları temsil ediyor. Fonksiyonumuz içinde yazdığımız koda göre kullanıcının girdiği klavye sembolleri ana penceremiz üzerindeki etikete yazdırılacak. Bir sonraki satırda yine **bind()** fonksiyonunu kullanarak, **goster()** adlı fonksiyon ile **<Key>** adlı özel ifadeyi birbirine bağlıyoruz. Bu **<Key>** ifadesi, kullanıcının klavye üzerindeki tuşlara basma hareketini temsil ediyor. Kodlarımızı bir bütün halinde düşünürsek; yukarıdaki uygulama, kullanıcının klavyede bastığı bütün tuşları ana pencere üzerindeki

etikete atayacaktır. Bu uygulama yardımıyla, esasında **salt okunur** (*read-only*) bir pencere aracı olan **`Label`**'i **yazılabilir** (*writable*) hale getirmiş oluyoruz (en azından görünüş olarak).

Öğrendiğimiz bu **`keysym`** niteliği yardımıyla bazı faydalı işler de yapabiliriz. Mesela; bir **`Entry`** aracına kullanıcı tarafından girilen bir metnin silinmesini engelleyebiliriz. Nasıl mı?

```
# -*- coding: utf-8 -*-
from Tkinter import *

pencere = Tk()
pencere.geometry("200x200")

giris = Entry()
giris.pack()

def silemezsin(event=None):
 if event.keysym == "BackSpace" or event.keysym == "Delete":
  return "break"

giris.bind("<Key>", silemezsin)

mainloop()
```

Buradaki kodlara göre, eğer kullanıcı **`BACKSPACE`** veya **`DELETE`** tuşlarına basarak yazdığı yazıyı silmek isterse, beklentilerinin aksine bu tuşlar çalışmayacaktır. Burada **`return "break"`** adlı özel bir ifadeden yararlanıyoruz. Bu ifade, normal şartlarda gerçekleşmesi engellenemeyecek olan işlemlerin etkisizleştirilmesini sağlayan özel bir kullanımdır. Örneğin; yukarıdaki **`silemezsin()`** fonksiyonunu bir de şöyle yazmayı deneyin:

```
def silemezsin(event=None):
 if event.keysym:
  return "break"
```

Bu şekilde, kullanıcı **`Entry`** aracı içine hiçbir şekilde yazı yazamayacaktır. Dolayısıyla yukarıdaki fonksiyonun adını "*silemezsin*" yerine "*yazamazsın*" koymak daha uygun olacaktır!

Son olarak, bu konuyla ilgili olduğu için, **focus_set()** fonksiyonundan da bahsetmekte fayda var. Bu fonksiyon, pencere araçlarını etkin hale getirmemizi sağlıyor. Bu bölümün en başında verdiğimiz örneği hatırlıyorsunuz. Kullanıcı bir kutucuğa adını giriyor ve girdiği bu ad bir dosyaya yazdırılıyordu. O örneği isterseniz yeniden çalıştırın. Göreceğiniz gibi kutucuğa bir şey yazabilmek için öncelikle o kutucuğa bir kez tıklamak gerekiyor. Yani aslında programımız ilk açıldığında o kutucuk etkin değil. Bizim onu etkin hale getirmek için üzerine bir kez tıklamamız gerekiyor. Ama istersek, o kutucuğun açılışta etkin hale gelmesini sağlayabiliriz. Böylece kullanıcılarımız doğrudan kutuya yazmaya başlayabilirler:

```
# -*- coding: utf-8 -*-
from Tkinter import *

pencere = Tk()

etiket = Label(text="Aşağıdaki kutucuğa adınızı yazıp \
Tamam düğmesine basın!\n")
etiket.pack()

giris = Entry()
giris.pack()
giris.focus_set()

def kaydet():
 dosya = open("isimler_python.txt","a")
 veri = giris.get()
 veri = unicode.encode(unicode(veri),"utf8")
 dosya.write(veri+"\n")
 dosya.close()
 etiket["text"] = etiket["text"] + u"%s dosyaya \
başarıyla eklendi\n"%giris.get()
 giris.delete(0,END)

dugme = Button(text="Tamam", command=kaydet)
dugme.pack()

mainloop()
```

Kodlar içinde yaptığımız eklemeyi koyu renkle gösterdim. **giris.focus_set()** fonksiyonu sayesinde, programımız açılır açılmaz kutucuk etkin hale geliyor. Böylece hemen adımızı yazmaya başlayabiliyoruz.

Bu konuyla bağlantılı olan şu örneğe bakalım bir de:

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

from Tkinter import *
from sys import exit

pencere = Tk()

etiket = Label(text="Programdan çıkmak istediğinize \
emin misiniz?")
etiket.pack()

cerceve = Frame()
cerceve.pack()

def sagol(event=None):
 yeni = Toplevel()
 etiket = Label(yeni, text="Teşekkürler... :)")
 yeni.bind("<Return>", exit)
 etiket.pack()

evet = Button(cerceve, text="Evet", command=pencere.destroy)
evet.grid(row=1,column=0)

hayir = Button(cerceve, text="Hayır", command=sagol)
hayir.grid(row=1,column=1)

evet.bind("<Return>", exit)
hayir.bind("<Return>", sagol)

mainloop()
```

Gördüğünüz gibi klavyedeki **ENTER** tuşunu bazı fonksiyonlara bağladığımız halde, programı çalıştırdığımızda **ENTER** tuşuna basmak hiçbir işe yaramıyor. Bunun nedeni, programımız ilk açıldığında pencerenin kendisi hariç hiç bir pencere aracının etkin olmamasıdır. Bağladığımız tuşların çalışabilmesi için öncelikle ilgili pencere araçlarının etkinleştirilmesi gerekiyor:

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

from Tkinter import *
from sys import exit

pencere = Tk()

etiket = Label(text="Programdan çıkmak istediğinize emin misiniz?")
```

```
etiket.pack()

cerceve = Frame()
cerceve.pack()

def sagol(event=None):
 yeni = Toplevel()
 etiket = Label(yeni, text="Teşekkürler... :)")
 yeni.bind("<Return>", exit)
 etiket.pack()

kapat = Button(cerceve, text="Evet", command=pencere.destroy)
kapat.grid(row=1, column=0)

kapatma = Button(cerceve, text="Hayır", command=sagol)
kapatma.grid(row=1, column=1)

kapat.focus_set()

kapat.bind("<Return>", exit)
kapatma.bind("<Return>", sagol)

mainloop()
```

Buradaki tek fark, kodlar arasına **`kapat.focus_set()`** ifadesinin eklenmiş olması. Bu ifade sayesinde, programımız ilk açıldığında odak **Evet** düğmesi üzerinde olacaktır. Dolayısıyla **ENTER** tuşuna bastığımızda, bu düğmenin bağlı olduğu fonksiyon çalışacak ve programımız kapanacaktır. Program açıkken, klavyedeki **TAB** tuşuna basarak odağı değiştirebiliriz. Mesela; programımızı ilk çalıştırdığımızda odak Evet düğmesi üzerinde olacağı için, **TAB** tuşuna bir kez bastığımızda odak Hayır düğmesine geçecektir. Bu haldeyken **ENTER** tuşuna basarsak, **Hayır** düğmesinin bağlı olduğu fonksiyon çalışacaktır.

İsterseniz konuyu kapatmadan önce basit bir oyun denemesi yapalım:

Önce *http://www.akcilaclama.com/images/sinek.ilaclama.jpg* adresine gidip oradaki sinek resmini bilgisayarınıza indirin. Adını da **sinek.jpg** koyun. Daha sonra şu kodları yazın:

```
# -*- coding: utf-8 -*-
from Tkinter import *

import random, ImageTk

pencere = Tk()
```

```
pencere.geometry("600x600")
pencere.tk_setPalette("white")

def yakala(event=None):
  k = random.randint(0,550)
  v = random.randint(0,550)
  print k, v
  dugme.place(x=k, y=v)

simge = ImageTk.PhotoImage(file="sinek.jpg")
dugme = Button(image=simge, command=yakala, relief="flat")
dugme.place(x=1, y=0)

dugme.bind("<Enter>", yakala)

mainloop()
```

Gördüğünüz gibi **ENTER** ifadesi, kullanıcının **ENTER** tuşuna bastığı anlamına gelmiyor. O iş için **`Return`** ifadesini kullanıyoruz. Fazlasıyla birbirine karıştırılan bir konu olduğu için özellikle vurgulama gereği duyuyorum.

Burada sineği yakalama ihtimaliniz var. Eğer **`random`**'un ürettiği sayılar birbirine yakın olursa sinek elinizden kaçamayabilir.

Böylelikle bu konuyu da bitirmiş olduk. Her ne kadar başlangıçta karışıkmış gibi görünse de aslında hem çok kolay hem de çok keyifli bir konudur fare ve klavye hareketleri konusu. Bu yazıyı birkaç kez gözden geçirerek bazı noktaların tam olarak zihninize yerleşmesini sağlayabilirsiniz.

Listbox Öğelerine Görev Atamak

Daha önce **`Listbox`** (*Liste kutusu*) adlı pencere aracının ne olduğunu ve nasıl kullanıldığını öğrenmiştik. Tkinter'de liste kutularını şöyle oluşturuyorduk:

```
from Tkinter import *

pencere = Tk()

listekutusu = Listbox()
listekutusu.pack()

mainloop()
```

İsterseniz bu liste kutusunun daha belirgin olması için buna birkaç öğe ekleyelim:

```
from Tkinter import *

pencere = Tk()

listekutusu = Listbox()
listekutusu.pack()

dagitimlar = [“Debian”, “Ubuntu”, “Mandriva”, “Arch”, “Gentoo”]

for i in dagitimlar:
 listekutusu.insert(0, i)

mainloop()
```

Elbette **`dagitimlar`** adlı listedeki öğelerin **`Listbox`**'taki konumlanışını ters de çevirebiliriz:

```
from Tkinter import *

pencere = Tk()

listekutusu = Listbox()
listekutusu.pack()

dagitimlar = [“Debian”, “Ubuntu”, “Mandriva”, “Arch”, “Gentoo”]

for i in dagitimlar:
 listekutusu.insert(END, i)

mainloop()
```

Burada **`listekutusu.insert()`** satırında bir önceki kodlarda `0` sıra numarasını verirken, yukarıdaki kodlarda **`END`** ifadesini kullandığımıza dikkat edin.

Gelin şimdi isterseniz bu liste öğelerinin her birine bir görev atayalım. Yani mesela kullanıcı "*Debian*" adlı öğenin üzerine çift tıkladığında kendisine Debian GNU/Linux hakkında bilgi verelim:

```
#-*-coding: utf-8-*-
from Tkinter import *

pencere = Tk()
pencere.geometry(“400x200”)

listekutusu = Listbox()
```

```
listekutusu["relief"] = "raised"
listekutusu.pack(side=LEFT, anchor=NW, fill=BOTH)

etiket = Label()
etiket["text"] = ""

dagitimlar = ["Debian", "Ubuntu", "Mandriva", "Arch", "Gentoo"]

for i in dagitimlar:
 listekutusu.insert(END, i)

def gosterici(event):
 etiket["text"] = "%s bir GNU/Linux dağıtımıdır!" \
%listekutusu.get(ACTIVE)

 etiket.pack()

listekutusu.bind("<Double-Button-1>", gosterici)

mainloop()
```

Burada bizi ilgilendiren kısım şu satır:

```
listekutusu.bind("<Double-Button-1>", gosterici)
```

Liste kutusuna görev atama işlemini bu satır yardımıyla yaptık. Kullanıcı liste kutusu içindeki herhangi bir öğeye çift tıkladığı zaman, tıklanan öğeyle ilgili bilgi veriyoruz. **`Listbox()`** adlı pencere aracının kendisine ait bir command seçeneği olmadığı için, istediğimiz işlevi, daha önceki derslerde gördüğümüz **`bind`** metodu yardımıyla hallediyoruz. Burada **`Double-Button-1`** ifadesini kullandığımıza dikkat edin. Bu ifade farenin sol tuşuna çift tıklama hareketini temsil ediyor. Liste kutularındaki öğelere işlev atamak istediğimiz zaman en doğru sonucu **`Double-Button-1`** ile elde ediyoruz. Öteki seçenekler her zaman istediğimiz işlevi yerine getirmeyebilir.

Pencereleri Başlıksız Hale Getirmek

Tkinter'de standart bir pencerenin nasıl oluşturulacağını biliyoruz:

```
pencere = Tk()
```

Bu şekilde bir pencere oluşturduğumuzda, elde ettiğimiz pencere, bir pencerenin sahip olması gereken bütün özellikleri taşır. Dolayısıyla pencereyi kapatmak,

pencereyi küçültmek veya pencereyi aşağı indirmek istediğimizde özel olarak herhangi bir kod yazmamıza gerek yoktur. Pencere başlığı üzerindeki çarpı, kare ve eksi düğmelerine tıklayarak gerekli işlevleri yerine getirebiliriz. Normal şartlar altında bu tür özelliklerin olmadığı bir pencere pek bir işimize yaramaz. Ama bazen pencerenin sürükleme çubuğu dahil hiç bir özelliğinin olmamasını isteyebiliriz. İşte bu bölümde bu isteğimizi nasıl yerine getirebileceğimizi göreceğiz. Şimdi normal bir şekilde programımızı yazalım:

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

from Tkinter import *

pencere = Tk()

mainloop()
```

Bu şekilde içi boş bir pencere oluşturduk. Şimdi bu kodlara şöyle bir şey ekleyelim:

```
pencere.overrideredirect(1)
```

Burada **overrideredirect** kelimesinin yazılışına azami özen göstermek gerekir. Uzun bir kelime olduğu için yanlış yazılma olasılığı oldukça yüksektir.

Kodlarımızın son hali şöyle oldu:

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

from Tkinter import *

pencere = Tk()
pencere.overrideredirect(1)

mainloop()
```

Yalnız bu kodları mutlaka komut satırından çalıştırın. Kesinlikle yazdığınız betiğin üzerine çift tıklayarak çalıştırmayı denemeyin.

Şimdi komut satırını kullanarak betiğimizi çalıştırıyoruz. Gördüğünüz gibi, bomboş bir kare elde ettik. Bu pencerenin, kendisini kapatmamızı sağlayacak bir düğmesi bile yok. Şimdi sanırım neden bu betiği komut satırından çalıştırdığımızı anlamışsınızdır. Bu başlıksız pencereyi **CTRL+C** (GNU/Linux) veya **CTRL+Z**

(Windows) tuşlarına basarak kapatabilirsiniz. Ya da doğrudan komut ekranını kapatarak da bu pencereyi ortadan kaldırabilirsiniz.

Peki, böyle başlıksız bir pencere oluşturmak ne işimize yarar? Mesela bu başlıksız pencereleri, bir düğmeye tıklandığında aşağıda doğru açılan bir menü olarak kullanabiliriz. Şu örneğe bakalım:

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

from Tkinter import *

pencere = Tk()

def liste():
 yenipencere = Toplevel()
 x = dgm.winfo_rootx()
 y = dgm.winfo_rooty() + dgm.winfo_height()
 yenipencere.geometry('+%d+%d' % (x,y))
 menu = Listbox(yenipencere)
 menu.pack()

 yenipencere.overrideredirect(1)

 dagitimlar = ["Debian", "Ubuntu", "Mandriva", "Arch", "Gentoo"]

 for i in dagitimlar:
  menu.insert(0, i)

dgm = Button(text="deneme", command=liste)
dgm.pack(fill=X)

mainloop()
```

Bu kodlar içinde geçen **x** ve **y** değişkenlerinin ne olduğunu bir sonraki bölümde inceleyeceğiz. Şimdilik bu değişkenlere fazla takılmayalım. Bu **x** ve **y** değişkenleri **yenipencere**'nin ana pencereye ve bunun üzerindeki **deneme** adlı düğmeye göre konumunu belirlememizi sağlıyor. Burada **deneme** adlı düğmeye tıkladığımızda bir seçim kutusu açıldığını görüyoruz. Elbette burada sadece kavramsal olarak durumu göstermek için bir örnek verdik. Şu haliyle yukarıdaki kodlar oldukça eksik. Amacım yalnızca başlıksız pencerelerle neler yapılabileceğine dair ufak bir örnek göstermekti... Yukarıdaki kodları işe yarar bir hale getirmek için üzerinde biraz çalışmanız gerekir.

Hatırlarsanız, önceki bölümlerden birinde "Pencere Araçlarına İpucu Metni Eklemek" başlıklı bir konu işlemiştik. Orada gösterdiğimiz **wcktooltips.py** adlı modülde de bu **overrideredirect** özelliğinden yararlanılıyor. Şu adrese gidip **wcktooltips** modülünün kaynağına baktığımızda kodlar arasında **self.popup.overrideredirect(1)** satırını görüyoruz. Yani Fredrik Lundh ipucu penceresini oluşturmak için başlıksız pencerelerden yararlanmış. Gördüğünüz gibi başlıksız pencereler pek çok yerde işimize yarayabiliyor.

Pencere / Ekran Koordinatları ve Boyutları

Bu bölümde bir pencerenin veya üzerinde çalıştığımız ekranın koordinatlarını ve boyutlarını nasıl öğrenebileceğimizi inceleyeceğiz. Peki, bu bilgi ne işimize yarayacak? Mesela; yazdığımız program ilk açıldığında kullanıcının ekranını ortalasın istiyorsak; programımızın çalıştırıldığı bilgisayar monitörünün boyutlarını/çözünürlüğünü bilmemiz gerekir ki, buna göre ekranın orta noktasını hesaplayabilelim. Ayrıca yazdığımız program içindeki pencere araçlarını da özellikle belli noktalarda konumlandırmak istiyorsak pencere ve üzerindeki araçların koordinatlarını bilmemiz gerekir. İsterseniz öncelikle hangi metotlardan yararlanabileceğimize bakalım:

```
>>> from Tkinter import *

>>> pencere = Tk()

>>> for i in dir(pencere):
...  if i.startswith("winfo"):
...   print i
...
winfo_atom
winfo_atomname
winfo_cells
winfo_children
winfo_class
winfo_colormapfull
winfo_containing
winfo_depth
winfo_exists
winfo_fpixels
winfo_geometry
```

```
winfo_height
winfo_id
winfo_interps
winfo_ismapped
winfo_manager
winfo_name
winfo_parent
winfo_pathname
winfo_pixels
winfo_pointerx
winfo_pointerxy
winfo_pointery
winfo_reqheight
winfo_reqwidth
winfo_rgb
winfo_rootx
winfo_rooty
winfo_screen
winfo_screencells
winfo_screendepth
winfo_screenheight
winfo_screenmmheight
winfo_screenmmwidth
winfo_screenvisual
winfo_screenwidth
winfo_server
winfo_toplevel
winfo_viewable
winfo_visual
winfo_visualid
winfo_visualsavailable
winfo_vrootheight
winfo_vrootwidth
winfo_vrootx
winfo_vrooty
winfo_width
winfo_x
winfo_y
```

Gördüğünüz gibi, elimizde epey metot var. Yukarıdaki kod yardımıyla **`Tk()`** sınıfı içindeki **`winfo`** ile başlayan bütün metotları listelediğimize dikkat edin. Biz bu bölümde bu metotların hepsini değil, bunlar arasında en sık kullanılanları inceleyeceğiz. İnceleyeceğimiz metotlar şunlar olacak:

```
winfo_height
winfo_width
winfo_rootx
winfo_rooty
winfo_screenheight
winfo_screenwidth
winfo_x
winfo_y
```

Hemen ilk metodumuzla işe başlayalım.

winfo_height()

Bu metot, oluşturduğumuz pencerenin (veya pencere aracının) yüksekliği hakkında bilgi verir bize. Şöyle bir örnekle durumu görelim:

```
>>> pencere = Tk()
>>> yukseklik = pencere.winfo_height()
>>> print yukseklik
1
```

Galiba bu çıktı pek fazla bir şey anlatmıyor bize. Bu kodlardan böyle bir çıktı almamızın nedeni bir pencere ilk oluşturulduğunda yükseklik değerinin 1 olarak kabul edilmesidir. Eğer pencerenin gerçek boyutunu görmek istersek yukarıdaki kodları şu şekilde yazmamız gerekir:

```
pencere = Tk()
pencere.update()
yukseklik = pencere.winfo_height()
print yukseklik
mainloop()
```

Burada kullandığımız **pencere.update()** komutu penceremizin mevcut durumunu güncellememizi sağlıyor. Dolayısıyla bu komut bize doğru bir şekilde 200 gibi bir çıktı veriyor. Buradan aldığımız sonuca göre oluşturduğumuz pencerenin yüksekliği 200 piksel. Dediğimiz gibi bu metodu yalnızca pencere üzerine uygulamak zorunda değiliz. Aynı metodu pencere araçlarıyla birlikte de kullanabiliriz:

```
from Tkinter import *

pencere = Tk()
pencere.geometry("200x200")

btn = Button(text="deneme")
btn.pack()

btn.update()

yukseklik = btn.winfo_height()

print yukseklik

mainloop()
```

Burada da düğme bilgilerini güncellemek için **btn.update()** gibi bir komuttan yararlandığımıza dikkat edin. Eğer bu örnekte **update()** metodunu kullanmazsak biraz önce olduğu gibi alacağımız çıktı 1 olacaktır.

Elbette **winfo_height()** metodunu bir değişkene atamadan da kullanabiliriz. Ama açıkçası o şekilde pek pratik olmayacaktır...

Şimdi sıra geldi **winfo_width()** metodunu incelemeye:

winfo_width()

İlk incelediğimiz metot olan **winfo_height**, bir pencere veya pencere aracının yüksekliğini veriyordu. **winfo_width()** metodu ise pencere veya pencere aracının genişliğini verir. Hemen görelim:

```
from Tkinter import *

pencere = Tk()

btn = Button(text="deneme")
btn.pack()
```

```
pencere.update()

genislik = pencere.winfo_width()

print genislik

mainloop()
```

Buradan aldığınız çıktı, pencerenin genişliğini gösterir. Eğer pencerenin değil de düğmenin genişliğini almak isterseniz ne yapmanız gerektiğini tahmin edebilirsiniz:

```
from Tkinter import *

pencere = Tk()

btn = Button(text="deneme")
btn.pack()

btn.update()

genislik = btn.winfo_width()

print genislik

mainloop()
```

Muhtemelen aldığınız çıktı, pencereden aldığınız çıktı ile aynı olacaktır. Çünkü Tkinter pencere üzerindeki araçlara göre pencerenin boyutunu ayarlıyor. Burada da Tkinter pencere üzerinde tek bir pencere aracı olduğu için, pencereyi **deneme** adlı düğmenin boyutu kadar ufalttı. Dolayısıyla pencerenin kendisi ve düğme aynı genişliğe sahip oldu. Pencerenin üzerinde birden fazla pencere aracı olduğu durumlarda **winfo_width()** metodunun işlevi daha net görülecektir.

winfo_rootx()

Bu metot pencere veya pencere araçlarının x düzlemi üzerindeki koordinatını verir. Mesela:

```
from Tkinter import *

pencere = Tk()
pencere.geometry("200x200")

btn = Button(text="deneme")
btn.pack()
```

```
pencere.update()

xkoord = pencere.winfo_rootx()

print xkoord

mainloop()
```

Ben bu kodları kendi bilgisayarımda çalıştırdığımda 965 çıktısını aldım. Benim monitörümün çözünürlüğü 1440x900. Demek ki bu pencerenin sol kenarı, ekranın soldan sağa 965'inci noktasına denk geliyormuş. Bir de şuna bakalım:

```
from Tkinter import *

pencere = Tk()
pencere.geometry("200x200")

btn = Button(text="deneme")
btn.pack()

btn.update()

xkoord = btn.winfo_rootx()

print xkoord

mainloop()
```

Bu kodları çalıştırdığımda ise konsolda 1027 çıktısını gördüm. Demek ki **deneme** adlı düğmenin sol kenarı 1440x900'lük monitörün 1027'inci noktasına denk geliyormuş...

winfo_rooty()

Bir önceki metodumuz olan **winfo_rootx()** x koordinatlarının bilgisini veriyordu. **winfo_rooty()** ise y koordinatlarının bilgisini verir:

```
from Tkinter import *

pencere = Tk()
pencere.geometry("200x200")

btn = Button(text="deneme")
btn.pack()

pencere.update()

ykoord = pencere.winfo_rooty()
```

```
print ykoord

mainloop()
```

Buradan aldığımız sonuç, pencerenin üst sınırının y düzlemi üzerinde hangi noktaya karşılık geldiğini gösteriyor. Penceremiz ekranın en tepesinde bile olsa bu kodlar 0 veya 1 gibi bir sonuç vermez. Çünkü pencere başlığının hemen altındaki alanın karşılık geldiği nokta hesaplanmaktadır. Bir de şu örneğe bakalım:

```
from Tkinter import *

pencere = Tk()
pencere.geometry("200x200")

btn = Button(text="deneme")
btn.pack(pady=30)

btn.update()

ykoord = btn.winfo_rooty()

print ykoord

mainloop()
```

Burada **pady** seçeneğini kullanarak düğmeyi biraz aşağıya kaydırdık ki bir önceki kodla arasındaki farkı görebilelim...

winfo_screenheight()

Bu metot ekran yüksekliğinin kaç olduğunu söyler. Örneğin 1024x768'lik bir çözünürlükte bu metodun verdiği değer 768 olacaktır...

```
from Tkinter import *

pencere = Tk()

ekran_y = pencere.winfo_screenheight()

print ekran_y

mainloop()

winfo_screenwidth()
```

Bu metot da winfo_screenheight() metoduna benzer. Ama onun aksine, bir pencerenin yüksekliğini değil, genişliğini verir. Dolayısıyla 1024x768'lik bir ekran çözünürlüğünde bu değer 1024 olacaktır:

```
from Tkinter import *

pencere = Tk()

ekran_g = pencere.winfo_screenwidth()

print ekran_g

mainloop()
```

En önemli **winfo** metotlarını gördüğümüze göre bunları kullanarak bazı yararlı işler yapmaya başlayabiliriz...

Programı Tam Ekran olarak Çalıştırmak

Geçen bölümde winfo metotlarını gördüğümüze göre bu bölümde bu metotları kullanarak bazı faydalı işler yapmaya çalışacağız. Bu metotlar yazdığımız programları istediğimiz şekilde konumlandırmada ve boyutlandırmada bize epey yardımcı olacaklardır. Örneğin; geçen bölümde öğrendiklerimizi kullanarak, yazdığımız bir programı tam ekran çalıştırabiliriz:

```
from Tkinter import*

pencere = Tk()
gen = pencere.winfo_screenwidth()
yuks = pencere.winfo_screenheight()
pencere.geometry("%dx%d"%(gen, yuks))

dgm = Button(text="~~~~~~~~~~~~TAM EKRAN~~~~~~~~~~")
dgm.pack(expand=YES, fill=BOTH)

pencere.mainloop()
```

Burada yaptığımız şey şu: Önce **pencere.winfo_screenwidth()** ifadesi yardımıyla ekran genişliğini alıyoruz. Daha sonra **pencere.winfo_screenheight()** ifadesini kullanarak ekran yüksekliğini öğreniyoruz. Bu bilgileri, kullanım kolaylığı açısından **gen** ve **yuks** adlı iki değişkene atadık. Ardından da **pencere.geometry()** içinde bu değerleri kullanarak programımızın ilk açılışta ekranın tamamını kaplamasını sağladık.

Python'un arkaplanda neler çevirdiğini daha net görmek için, kullandığımız bu **gen** ve **yuks** değişkenlerini ekrana yazdırmak isteyebilirsiniz...

Ekrani Ortalamak

winfo metotlarını kullanarak, yazdığımız bir programın ilk açıldığında ekranın tam ortasına denk gelmesini de sağlayabiliriz:

```
from Tkinter import*

pencere = Tk()

pgen = 200
pyuks = 200

ekrangen = pencere.winfo_screenwidth()
ekranyuks = pencere.winfo_screenheight()

x = (ekrangen - pgen) / 2
y = (ekranyuks - pyuks) / 2

pencere.geometry("%dx%d+%d+%d"%(pgen, pyuks, x, y))

pencere.mainloop()
```

Burada önce **pgen** ve **pyuks** değişkenleri içinde programımızın genişliğini ve yüksekliğini belirttik. Bunları elbette doğrudan **pencere.geometry** ifadesi içine de yerleştirebilirdik. Ama bu iki değerle bazı hesaplamalar yapacağımız için, en iyisi bunları bir değişken içine atmak.

İlk değişkenlerimizi tanımladıktan sonra ekran genişliğini ve ekran yüksekliğini öğreniyoruz. Bunun için **winfo_screenwidth()** ve **winfo_screenheight()** metotlarını kullandık. Yine kullanım kolaylığı açısından bu iki değeri sırasıyla **ekrangen** ve **ekranyuks** adlı değişkenlere atıyoruz.

Şimdi programımızın ekranı tam ortalayabilmesi için ufak bir hesap yapmamız gerekecek. x değerini bulabilmek için ekran genişliğinden programımızın pencere genişliğini çıkarıp, elde ettiğimiz değeri ikiye bölüyoruz. Mesela; eğer kullandığımız ekran çözünürlüğü 1024x768 ise, x değeri şöyle olacaktır:

```
x = (1024 - 200) / 2
```

y değerini bulmak için de benzer bir şekilde ekran yüksekliğinden program yüksekliğini çıkarıp, bu değeri yine ikiye bölüyoruz. Dolayısıyla y değeri 1024x768'lik bir ekranda şöyle olur:

```
y = (768 - 200) / 2
```

Bu hesaplamadan elde ettiğimiz verileri **`pencere.geometry()`** içinde uygun yerlere yerleştirdiğimizde, programımız ekranın tam ortasında açılacaktır.

Pencereleri Her Zaman En Üstte Tutmak

Bazen ana pencereye ek olarak ikinci bir pencere daha oluşturmamız gerekir. Ancak bazen programın işleyişi sırasında, aslında hep üstte durması gereken bu ikinci pencerenin, ana pencerenin arkasına düştüğünü görürüz. Böyle bir durumda yapmamız gereken şey, ikinci pencerenin daima üstte kalmasını sağlayacak bir kod yazmaktır. Neyse ki Tkinter bize böyle bir durumda kullanılmak üzere faydalı bir metot sunar. Bu metodun adı **`transient()`**. İsterseniz hemen bununla ilgili bir örnek yapalım. Diyelim ki şöyle bir uygulamamız var:

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

from Tkinter import *

from tkFileDialog import askopenfilename

pencere = Tk()

pgen = 200
pyuks = 200

ekrangen = pencere.winfo_screenwidth()
ekranyuks = pencere.winfo_screenheight()

x = (ekrangen - pgen) / 2
y = (ekranyuks - pyuks) / 2

pencere.geometry("%dx%d+%d+%d"%(pgen, pyuks, x, y))

def yeniPencere():
 yeni = Toplevel()

 xkonum = pencere.winfo_rootx()
```

```
 ykonum = pencere.winfo_rooty()
 yeni.geometry("+%d+%d"%(xkonum, ykonum))

 ybtn = Button(yeni, text="Dosya aç",
      command=yeniDosya)
 ybtn.pack()

def yeniDosya():
 dosya = askopenfilename()

btn = Button(pencere, text="yeni pencere aç",
    command=yeniPencere)
btn.pack()

mainloop()
```

Burada gördüğümüz **`tkFileDialog`** modülünün şimdilik üzerinde durmayalım. Birkaç bölüm sonra bu ve benzeri modülleri ayrıntılı olarak inceleyeceğiz. Biz şimdilik bu modülün dosya açma işlemlerinde kullanıldığını bilelim yeter...

Programımızın, ekranın tam ortasında açılacak şekilde ayarlandığına dikkat edin. Aynı şekilde ikinci pencere de ana pencerenin üzerinde açılacak şekilde ayarlandı. Bu işlemleri **`winfo`** metotları yardımıyla yaptığımızı görüyorsunuz.

Bu programı çalıştırdığımızda pencere araçlarının biraz tuhaf davrandığını görebilirsiniz. Mesela; ikinci pencere üzerindeki **Dosya aç** düğmesine bastığımızda açılan dosya seçme penceresi ikinci pencerenin altında kalıyor olabilir. Aynı şekilde, dosya seçme ekranında **Cancel** tuşuna bastığımızda ikinci pencere bir anda ortadan kaybolacak ve ana pencerenin arkasına gizlenecektir. Bu program belki her sistemde aynı tepkiyi vermeyebilir. Hatta belki bazı sistemlerde bu şekilde bile düzgün çalışıyor olabilir. Ama bizim istediğimiz şey, programımızın hemen her sistemde mümkün olduğunca aynı şekilde çalışmasını sağlamak. O halde hemen gerekli kodları yazalım. Yazacağımız kod çok basittir. Tek yapmamız gereken, ikinci pencerenin ana pencereye göre üstte kalmasını garanti etmek. Bunu şu satırla yapacağız:

```
yeni.transient(pencere)
```

Yani kodlarımızın son hali şöyle olacak:

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

from Tkinter import *
from tkFileDialog import askopenfilename

pencere = Tk()

pgen = 200
pyuks = 200

ekrangen = pencere.winfo_screenwidth()
ekranyuks = pencere.winfo_screenheight()

x = (ekrangen - pgen) / 2
y = (ekranyuks - pyuks) / 2

pencere.geometry("%dx%d+%d+%d"%(pgen, pyuks, x, y))

def yeniPencere():
 yeni = Toplevel()
 yeni.transient(pencere)

 xkonum = pencere.winfo_rootx()
 ykonum = pencere.winfo_rooty()
 yeni.geometry("+%d+%d"%(xkonum, ykonum))

 ybtn = Button(yeni, text="Dosya aç",
      command=yeniDosya)
 ybtn.pack()

def yeniDosya():
 dosya = askopenfilename()

btn = Button(pencere, text="yeni pencere aç",
    command=yeniPencere)
btn.pack()

mainloop()
```

Yeni eklediğimiz satır, **`yeni`** adlı ikinci pencerenin ana pencereye göre hep üstte kalmasını temin ediyor. Programımızı bu şekilde çalıştırdığımızda her şeyin olması gerektiği gibi olduğunu göreceğiz. Kodlarımızı bu şekilde yazdığımızda, ikinci pencereyi açtıktan sonra ana pencereye tıklasak bile ikinci pencere ana pencerenin arkasına düşmeyecektir.

STANDART BİLGİ PENCERELERİ (STANDARD DIALOGS)

7

Bu bölümün konusu; Tkinter'de Standart Bilgi Pencereleri'nin kullanımı. Nedir bu "Standart Bilgi Penceresi" denen şey? Diyelim ki bir program yazıyorsunuz. Bu programda herhangi bir işlemle ilgili olarak kullanıcıya bir mesaj göstermek istediğimizde karşımızda iki yol var. Birinci yolda, Tkinter'in **`Toplevel()`** aracını kullanarak, göstermek istediğimiz mesajı ve gerekli düğmeleri içeren bir pencere oluşturabiliriz. Hemen bununla ilgili bir örnek verelim:

```
# -*- coding: utf-8 -*-
from Tkinter import *

pencere = Tk()

def kapat():
 if not entry.get():
  yenipen = Toplevel()

  uyari = Label(yenipen)
  uyari["text"] = "Lütfen geçerli bir e.posta \
adresi yazın!"
  uyari.pack()

  uybtn = Button(yenipen)
  uybtn["text"] = "Tamam"
  uybtn["command"] = yenipen.destroy
  uybtn.pack()

 else:
  pencere.destroy()
```

```
etiket = Label()
etiket["text"] = "e.posta adresiniz:"
etiket.pack()

entry = Entry()
entry.pack()

dgm = Button()
dgm["text"] = "Gönder"
dgm["command"] = kapat
dgm.pack()

pencere.mainloop()
```

Burada eğer kullanıcı **`Entry()`** aracına herhangi bir şey yazmadan "*Gönder*" düğmesine basarsa bir uyarı penceresi açılacak ve kullanıcıyı **`Entry()`** aracına geçerli bir e-posta yazması konusunda uyaracaktır.

Dediğimiz gibi böyle bir işlev yerine getirmeniz gerekirse, yukarıdaki yöntemi kullanabilirsiniz. Ama aslında buna hiç gerek yok. Tkinter bize bu tür işleri yapabilmemiz için bize bazı araçlar sunar. İşte biz de bu bölümde Tkinter'in bize sunduğu bu araçları inceleyeceğiz.

Hata Mesajı Gösteren Pencere

Bu bölümde inceleyeceğimiz ilk penceremiz hata mesajı gösteren pencere olacaktır. Yalnız bahsettiğimiz bu pencereleri kullanabilmek için öncelikle tkMessageBox adlı modülü içe aktarmamız gerekiyor.

```
from Tkinter import *
from tkMessageBox import *
```

Tkinter'de kullanıcıya bilgi mesajı gösteren bütün pencereler **`tkMessageBox`** adlı modülün içinde yer alıyor. Dolayısıyla bu pencereleri kullanmak istediğimizde, Tkinter modülüyle birlikte **`tkMessageBox`** modülünü de içe aktarmayı unutmuyoruz.

Bir önceki bölümde hatırlarsanız, kullanıcıdan e-posta adresi girmesini istediğimiz ufak bir uygulama yazmıştık. Yine o örnek üzerinden gidelim.

Öncelikle kodlarımızın ne olduğuna bakalım:

```
# -*- coding: utf-8 -*-
from Tkinter import *

pencere = Tk()

def kapat():
 if not entry.get():
  yenipen = Toplevel()

  uyari = Label(yenipen)
  uyari["text"] = "Lütfen geçerli bir e.posta \
adresi yazın!"
  uyari.pack()

  uybtn = Button(yenipen)
  uybtn["text"] = "Tamam"
  uybtn["command"] = yenipen.destroy
  uybtn.pack()

 else:
  pencere.destroy()

etiket = Label()
etiket["text"] = "e.posta adresiniz:"
etiket.pack()

entry = Entry()
entry.pack()

dgm = Button()
dgm["text"] = "Gönder"
dgm["command"] = kapat
dgm.pack()

pencere.mainloop()
```

Gördüğünüz gibi, aslında basit bir iş için çok fazla kod yazmışız.

Gelin bu kodları, **tkMessageBox** modülünün de yardımıyla sadeleştirelim ve hatta güzelleştirelim:

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

from Tkinter import *
from tkMessageBox import *

pencere = Tk()

def kapat():
 if not entry.get():
  showerror("Hata!", "Lütfen geçerli bir e.posta \
adresi girin!")
 else:
  pencere.destroy()

etiket = Label()
etiket["text"] = "e.posta adresiniz:"
etiket.pack()

entry = Entry()
entry.pack()

dgm = Button()
dgm["text"] = "Gönder"
dgm["command"] = kapat
dgm.pack()

pencere.mainloop()
```

Ne kadar hoş, değil mi? Bu şekilde hem daha az kod yazmış oluyoruz, hem de elde ettiğimiz çıktı daha oturmuş bir görünüme sahip. Dikkat ederseniz, hata mesajımız sevimli bir hata simgesi de içeriyor. Bu hata penceresinin bize sağladıklarını kendimiz yapmaya çalışırsak epey kod yazmamız gerekir. Dolayısıyla yazacağımız programlarda böyle bir imkanımızın olduğunu da göz önünde bulundurmamız bizim için faydalı olacaktır.

Bir hata penceresinin sahip olması gereken bütün özellikleri taşıyan bu hata penceresini, şu tek satırlık kod yardımıyla oluşturduk:

```
showerror("Hata!", "Lütfen geçerli bir e.posta adresi girin!")
```

Burada neyin ne olduğu bellidir. Ama isterseniz bu satırı biraz inceleyelim:

Gördüğünüz gibi aslında kullandığımız şey **`showerror()`** adlı bir fonksiyon. Bu fonksiyonu iki parametreli olarak kullandık. **`showerror()`** fonksiyonu içinde kullandığımız ilk parametre "*Hata*" adlı bir karakter dizisi. Bu ifade, oluşacak hata penceresinin başlığı olacak. İkinci parametremiz olan "*Lütfen geçerli bir e.posta adresi girin!*" adlı karakter dizisi ise, hata penceremizin gövdesi, yani kullanıcıya göstereceğimiz mesajın ta kendisidir.

Burada kullandığımız **`showerror()`** fonksiyonu başka bir parametre daha alabilir. Bu parametrenin adı detail'dir. Bu parametreyi şöyle kullanıyoruz:

```
showerror("Hata!", "Lütfen geçerli bir e.posta adresi girin!",
    detail = "e.posta adresini doğru yazmaya özen gösterin!")
```

`detail` parametresi, varsa hatayla ilgili ayrıntıları da kullanıcıya göstermemizi sağlıyor. Bu arada yukarıdaki örnek kod satırı eğer gözünüze uzun görünüyorsa bu satırı şu şekilde bölebilirsiniz:

```
showerror("Hata!",
    "Lütfen geçerli bir e.posta adresi girin!",
    detail = "e.posta adresini doğru yazmaya özen gösterin!")
```

Özellikle Python kodlarına duyarlı bir metin düzenleyici kullanıyorsanız, (mesela IDLE) virgülden sonra **ENTER** tuşuna bastığınızda metin düzenleyiciniz satırları uygun şekilde hizalayacaktır.

Elbette isterseniz daha estetik bir görüntü elde etmek için değişkenlerden de faydalanabilirsiniz:

```
baslik = "Hata!"
mesaj = "Lütfen geçerli bir e.posta adresi girin!"
detay = "e.posta adres uydurma olmasın!"

showerror(baslik, mesaj, detail = detay)
```

`detail` parametresi dışında, **`showerror()`** fonksiyonu için (ve tabii öteki bilgi mesajları için) kullanabileceğimiz başka parametreler de vardır. Mesela; bu parametrelerden biri **`type`** parametresidir. Bu parametre ile bilgi ekranında gösterilecek düğmelerin tipini belirleyebiliriz.

Bu parametre ile birlikte kullanabileceğimiz altı farklı seçenek vardır:

- **ABORTRETRYIGNORE:** Pencere üzerinde **vazgeç**, **tekrar dene**, **yoksay** düğmelerini gösterir.
- **OK:** Pencere üzerinde **tamam** düğmesi gösterir.
- **OKCANCEL:** Pencere üzerinde **tamam** ve **iptal** düğmelerini gösterir.
- **RETRYCANCEL:** Pencere üzerinde **yeniden dene** ve **iptal** düğmelerini gösterir.
- **YESNO:** Pencere üzerinde **evet** ve **hayır** düğmelerini gösterir.
- **YESNOCANCEL:** Pencere üzerinde **evet**, **hayır** ve **iptal** düğmelerini gösterir.

Bu seçenekler şöyle kullanılır:

```
showerror(baslik, mesaj, detail=detay,
    type=ABORTRETRYIGNORE)
```

Burada **`ABORTRETRYIGNORE`** yerine başka herhangi bir seçeneği de koyabilirsiniz elbette...

Başka bir parametre ise **`default`** parametresidir. Bu parametre yardımıyla, hata penceresi ilk açıldığında hangi düğmenin seçili (etkin) olacağını belirleyebiliriz. Yani doğrudan **ENTER** tuşuna basıldığında, hangi düğmenin çalışacağını bu default adlı parametre ile belirliyoruz.

default parametresi yedi farklı seçeneğe sahiptir:

- **`ABORT`**
- **`RETRY`**
- **`IGNORE`**
- **`OK`**
- **`CANCEL`**
- **`YES`**
- **`NO`**

Bu parametreyi de şöyle kullanıyoruz:

```
showerror(baslik, mesaj, detail=detay, type=ABORTRETRYIGNORE,
default=RETRY)
```

En son parametremiz ise **icon** adlı olan. Bu parametre yardımıyla, bilgi penceresi üzerinde gördüğümüz simgenin türünü belirleyebiliriz. Bu parametre ile şu seçenekleri kullanabiliriz:

- **ERROR** (Hata simgesi)
- **INFO** (Bilgi simgesi)
- **QUESTION** (Soru simgesi)
- **WARNING** (Uyarı simgesi)

Bu parametreyi ise şöyle kullanacağız:

```
showerror(baslik, mesaj, detail=detay,
    type=ABORTRETRYIGNORE, default=RETRY,
    icon=WARNING)
```

Bu arada, mümkün olan yerlerde kod satırlarımızı yukarıda gibi uzun tutmamak iyi bir yaklaşım olacaktır. Dolayısıyla yukarıdaki satırı biraz daha bölmek okunaklılığı artırır:

```
showerror(baslik,
    mesaj,
    detail = detay,
    type = ABORTRETRYIGNORE,
    default=RETRY,
    icon=WARNING)
```

Peki, kullanıcı bu bilgi ekranındaki herhangi bir düğmeye bastığında hangi işlemin yapılacağını nasıl belirleyeceğiz? Yani mesela diyelim ki kullanıcı **OK** tuşuna bastı, işte böyle bir durumda hangi işlemin yapılması gerektiğini nasıl bulacağız? Bu işi yapmakta oldukça kolaydır.

Hemen şöyle bir örnek verelim:

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

from Tkinter import *
from tkMessageBox import *

pencere = Tk()

def soru():
 cevap = showerror("Uyarı",
       "Bu işlem diskteki bütün verileri silecektir.",
       type=OKCANCEL)

 if cevap == "ok":
  etiket["text"] = "disk biçimlendirildi!"

 if cevap == "cancel":
  etiket["text"] = "işlem durduruldu!"

etiket = Label(text="işlem durumu: ")
etiket.pack()

dgm = Button(text="diski biçimlendir!", command=soru)
dgm.pack()

mainloop()
```

Gördüğünüz gibi, **`showerror()`** fonksiyonunu bir değişkene atadık. Burada type parametresini kullanarak, içinde **OK** ve **CANCEL** gibi iki farklı düğme barındıran bir bilgi ekranı oluşturduğumuza dikkat edin. Eğer kullanıcı **OK** tuşuna basarsa cevap değişkeninin değeri ok, yok eğer kullanıcı **CANCEL** tuşuna basarsa cevap değişkeninin değeri **`cancel`** olacaktır. İşte bu değerleri birer **`if`** deyimine bağlayarak her bir düğmeye ayrı bir işlev atayabiliriz.

Burada kendi kendinize, type parametresine farklı değerler vererek denemeler yapmanızı tavsiye ederim. Mesela; **`OKCANCEL`**, **`RETRYCANCEL`**, **`ABORTRETRYIGNORE`** gibi değerlerin ne tür çıktılar verdiğini kontrol ederek çıktıya göre farklı fonksiyonlar yazabilirsiniz.

Mesela; şöyle bir test uygulaması ile her bir düğmenin nasıl bir çıktı verdiğini kontrol edebilirsiniz:

```
from Tkinter import *
from tkMessageBox import *

pencere = Tk()

def soru():
 cevap = showerror("Uyarı",
       "Bu işlem diskteki bütün verileri silecektir.",
       type=ABORTRETRYIGNORE)
 print cevap

dgm = Button(text="diski biçimlendir!", command=soru)
dgm.pack()

mainloop()
```

Elbette burada çıktıları konsola veriyoruz. Ama eğer siz isterseniz arayüz üzerindeki **`Label()`** aracının **`text`** seçeneğine de bu çıktıları yazdırabilirsiniz:

```
from Tkinter import *
from tkMessageBox import *

pencere = Tk()

def soru():
 cevap = showerror("Uyarı",
       "Bu işlem diskteki bütün verileri silecektir.",
       type=ABORTRETRYIGNORE)

 etiket["text"] = cevap

etiket = Label()
etiket.pack()

dgm = Button(text="diski biçimlendir!", command=soru)
dgm.pack()

mainloop()
```

Gördüğünüz gibi, **`showerror()`** fonksiyonunu kullanmak zor değil. Gelin isterseniz şimdi öteki bilgi pencerelerine de şöyle bir göz atalım.

Bilgi Mesajı Gösteren Pencere

Bir önceki bölümde hata mesajı gösteren pencereyi işlemiştik. Bu bölümde ise "bilgi mesajı gösteren pencere"yi öğreneceğiz.

Bilgi mesajı gösterne pencere de tıpkı hata mesajı gösteren pencere gibidir. Bu kez **`showerror()`** fonksiyonu yerine **`showinfo()`** fonksiyonunu kullanacağız. En basit şekliyle **`showinfo()`** fonksiyonu şöyle kullanılır:

```
showinfo("Bilgi", "İşlem başarıyla tamamlandı!")
Gelin isterseniz bu fonksiyonu kapsamlı bir örnek içinde
inceleyelim:
# -*- coding: utf-8 -*-

from Tkinter import *
from tkMessageBox import *

pencere = Tk()

def kapat():
 if not entry.get():
  showerror("Hata!", "Ürün sayısı belirtmediniz!")
  return "break"
 try:
  int(entry.get())
  showinfo("Bilgi", "%s ürün sepete eklendi!"%entry.get())

 except ValueError:
  showerror("Hata!", "Lütfen bir sayı girin!")

etiket = Label()
etiket["text"] = "ürün adedi:"
etiket.pack()

entry = Entry()
entry.pack()

dgm = Button()
dgm["text"] = "Gönder"
dgm["command"] = kapat
dgm.pack()

pencere.mainloop()
```

Burada birkaç işlemi aynı anda yaptık. Programımızın amacı; kullanıcının kutucuğa bir sayı yazmasını sağlamak. Ayrıca kullanıcının, kutucuğa hiç bir şey yazmadan "*Gönder*" tuşuna basabileceğini de hesaba katmamız gerekir... Böyle bir duruma karşı, kodlarımız içinde şu satırları yazdık:

```
if not entry.get():
 showerror("Hata!", "Ürün sayısı belirtmediniz!")
 return "break"
```

Eğer kullanıcımız, kutucuğu boş bırakırsa kendisine bir hata mesajı gösteriyoruz. Burada **`return "break"`** ifadesini kullanmamızın nedeni, hata mesajı gösterildikten sonra programın çalışmaya devam etmesini engellemek. Eğer bu ifadeyi yazmazsak, programımız kullanıcıya hata mesajı gösterdikten sonra alt satırdaki kodları da işletmeye devam edecektir. İsterseniz, o satırı kaldırıp kendi kendinize birtakım denemeler yapabilirsiniz...

Bildiğimiz gibi, **`entry.get()`** metodundan elde edilen verinin tipi karakter dizisidir. Kullanıcının bir sayı mı yoksa bir harf mi girdiğini tespit edebilmek için öncelikle, kutucuğa kullanıcı tarafından yazılan değerin sayıya dönüştürülebilen bir veri olup olmadığına bakıyoruz. Bunun için şu kodları yazdık:

```
try:
 int(entry.get())
 showinfo("Bilgi", "%s ürün sepete eklendi!"%entry.get())

except ValueError:
 showerror("Hata!", "Lütfen bir sayı girin!")
```

Burada eğer kullanıcının yazdığı şey sayıya dönüştürülebilen bir veri ise kendisine **`showinfo()`** fonksiyonunu kullanarak kaç ürünün sepete eklendiği bilgisini veriyoruz. Ama eğer mesela kullanıcı sayı girmek yerine "*fsdfd*" gibi bir şey yazarsa, bu defa **`showerror()`** fonksiyonunu kullanarak ona, sayı girmediğine dair bir hata mesajı gösteriyoruz. Burada **`try except`** bloklarını nasıl kullandığımıza dikkat edin...

Gördüğünüz gibi, **`showinfo()`** fonksiyonunun kullanımı, **`showerror()`** fonksiyonunun kullanımına çok benziyor. Hatta hemen hemen aynı bile diyebiliriz...

Geçen bölümde **showerror()** fonksiyonunu işlerken, bu fonksiyonun birtakım parametreler de alabildiğini söylemiştik. Orada söylediklerimiz **showinfo()** fonksiyonu için de geçerlidir. Mesela; burada da detail parametresini kullanarak, kullanıcıya konuyla ilgili ek bilgi verebiliriz:

```
showinfo("Bilgi", "%s ürün sepete eklendi!"%entry.get(),
   detail="Ama bu kadar ürün az!\nÜrünlerimizden \
biraz daha alın!")
```

showerror() ile **showinfo()** arasındaki benzerlikler bununla da sınırlı değildir. Mesela; geçen bölümde öğrendiğimiz type parametresini burada da kullanabiliriz:

```
showinfo("Bilgi", "%s ürün sepete eklendi!"%entry.get(),
   detail="Ama bu kadar ürün az!\nÜrünlerimizden \
biraz daha alın!",
   type=OKCANCEL)
```

Gördüğünüz gibi **showerror()** konusunda ne öğrendiysek burada da uygulayabiliyoruz. İsterseniz geçen bölümde öğrendiğimiz bilgileri **showinfo()** fonksiyonuna da uygulayarak kendi kendinize el alıştırmaları yapabilirsiniz.

Sıra geldi başka bir bilgi mesajı penceresini incelemeye...

Uyarı Mesajı Gösteren Pencere

Bu bölümde kullanıcılarımıza bir uyarı mesajı gösteren pencerenin nasıl oluşturulacağına bakacağız... Burada da yeni bir fonksiyonla karşılaşacağız. Ama endişelenmeye gerek yok. Bu fonksiyon da tıpkı **showerror()** ve **showinfo()** fonksiyonlarına benzer. Adı ve tabii ki üzerindeki ünlem simgesi dışında her yönüyle onlarla aynıdır. Bu fonksiyonumuzun adı **showwarning()**. İsterseniz bununla ilgili de basit bir örnek yapıp bu konuyu kapatalım:

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

from Tkinter import *
from tkMessageBox import *

pencere = Tk()

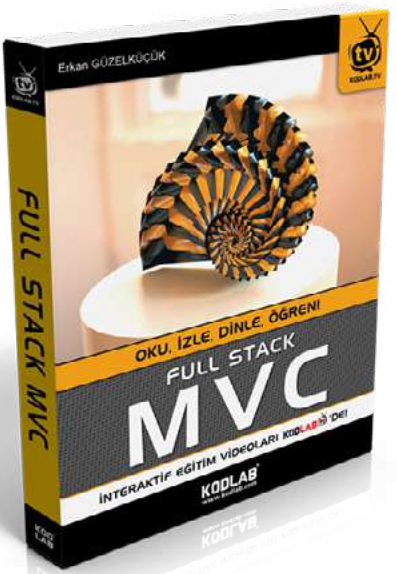
def kapat():
   showwarning("Bağlantı yok!", "İşleminizi \
gerçekleştiremiyorum!")
```

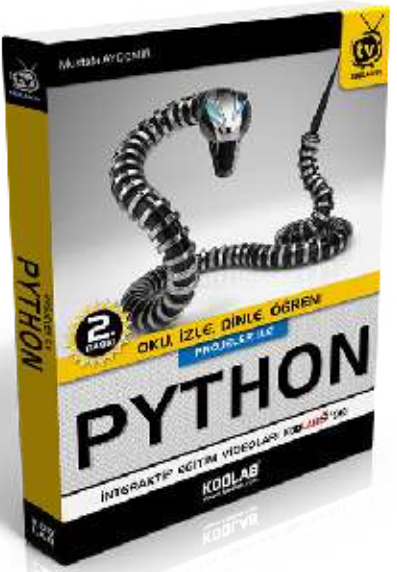
```
etiket = Label()
etiket["text"] = "ürün adedi:"
etiket.pack()

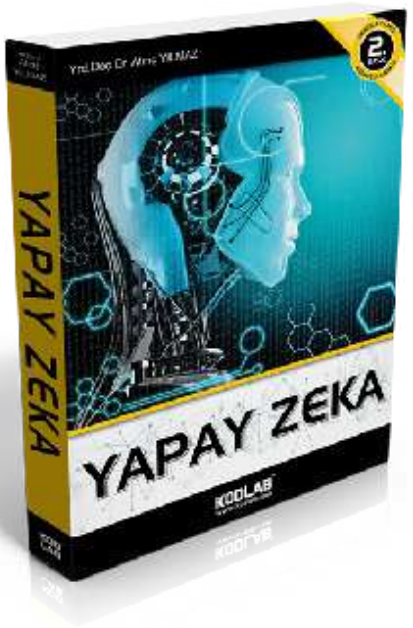
entry = Entry()
entry.pack()

dgm = Button()
dgm["text"] = "Gönder"
dgm["command"] = kapat
dgm.pack()

pencere.mainloop()
```


Erkan GÜZELKÜÇÜK
FULL STACK MVC
OKU, İZLE, DİNLE, ÖĞREN!
FULL STACK
MVC
İNTERAKTİF EĞİTİM VİDEOLARI KODLAB'DE!
KODLAB


PYTHON
2. BASKI
OKU, İZLE, DİNLE, ÖĞREN!
PROJELERLE
PYTHON
İNTERAKTİF EĞİTİM VİDEOLARI KODLAB'DE!
KODLAB


YAPAY ZEKA
KODLAB


Tolga BÜYÜKTANIR - Büşra ÖZER
NESNELERİN İNTERNETİ
OKU, İZLE, DİNLE, ÖĞREN!
YENİ BAŞLAYANLAR İÇİN
IoT (Internet of Things)
NESNELERİN İNTERNETİ
İNTERAKTİF EĞİTİM VİDEOLARI KODLAB'DE!
KODLAB

robocombo.com
hayal et ve yap

"Binlerce hobi elektronik ve robotik ürünleri bulabileceğiniz tek adres!"

arduino | raspberry pi | devre elemanları | motorlar | geliştirme kartları | eğitim kitapları | drone | daha fazlası...



Türkiye'nin her yerine kargo 5TL

süper hızlı gönderi

size özel fiyatlar

/robocombocom