



[www.tdsoftware.tr.cx](http://www.tdsoftware.tr.cx)

# İçindekiler

<u>Temel Bilgiler</u>	6
<u>Giriş</u>	6
<u>Python'u Nereden Bulabilirim?</u>	6
<u>Python Nasıl Çalıştırılır?</u>	7
<u>print komutu</u>	7
<u>Python'da Sayılar ve Matematik İşlemleri</u>	9
<u>Değişkenler</u>	11
<u>Metin Düzenleyici Kullanılarak Python Programı Nasıl Yazılır?</u>	12
<u>Kullanıcıyla İletişim: Veri Alış-Verişi</u>	17
<u>raw_input() fonksiyonu</u>	17
<u>input() fonksiyonu</u>	19
<u>Python'da Koşula Bağlı Durumlar</u>	21
<u>if...</u>	21
<u>else:</u>	25
<u>elif...</u>	25
<u>Python'da Döngüler</u>	29
<u>while döngüsü</u>	29
<u>for döngüsü</u>	35
<u>range() fonksiyonu</u>	36
<u>len() fonksiyonu</u>	37
<u>break ifadesi</u>	37
<u>continue ifadesi</u>	38
<u>Python'da Listeler, Demetler, Sözlükler</u>	39
<u>Listeler</u>	39
<u>append parçacığı</u>	41
<u>insert parçacığı</u>	42
<u>extend parçacığı</u>	43
<u>liste + yeni_liste</u>	43
<u>remove parçacığı</u>	43
<u>pop parçacığı</u>	43
<u>index parçacığı</u>	44
<u>sort parçacığı</u>	44
<u>liste.sort()</u>	45
<u>reverse parçacığı</u>	45
<u>count parçacığı</u>	45
<u>Demetler</u>	48
<u>Sözlükler</u>	49
<u>telefon_defteri.clear()</u>	50
<u>if-elif-else Yerine Sözlük Kullanmak</u>	50
<u>Python'da Fonksiyonlar</u>	53
<u>Giriş</u>	53
<u>Fonksiyon Tanımlama</u>	53
<u>Komut Satırında Fonksiyon Tanımlama</u>	54
<u>Metin Düzenleyicide Fonksiyon Tanımlama</u>	54
<u>Fonksiyonlarda Parametreler</u>	55
<u>Fonksiyon İçindeki Değişkenlerin Okunma Sırası</u>	60

<a href="#">merhaba(arkadas, ark)</a> .....	62
<a href="#">"global" ifadesi</a> .....	62
<a href="#">"return" ifadesi</a> .....	64
<a href="#">"pass" ifadesi</a> .....	66
<a href="#">Modüller</a> .....	66
<a href="#">Giriş</a> .....	66
<a href="#">Modül Çekme (importing Modules)</a> .....	66
<a href="#">os Modülü</a> .....	67
<a href="#">name fonksiyonu</a> .....	68
<a href="#">listdir fonksiyonu</a> .....	69
<a href="#">getcwd fonksiyonu</a> .....	70
<a href="#">chdir() fonksiyonu</a> .....	71
<a href="#">mkdir() ve makedirs() fonksiyonları</a> .....	72
<a href="#">rmdir() ve removedirs() fonksiyonları</a> .....	74
<a href="#">os.sep fonksiyonu</a> .....	75
<a href="#">Dosya İşlemleri</a> .....	78
<a href="#">Yeni bir dosya yaratma</a> .....	79
<a href="#">Dosyaya Veri İşleme</a> .....	81
<a href="#">Bir Dosyadan Veri Okuma</a> .....	83
<a href="#">Dosyaya Değişken Yazdırma</a> .....	86
<a href="#">int(x)</a> .....	87
<a href="#">Dosya Silme</a> .....	87
<a href="#">Dosyanın Herhangi Bir Yerine Satır Ekleme</a> .....	87
<a href="#">Hatalarla Başetme</a> .....	89
<a href="#">Giriş</a> .....	89
<a href="#">Değişken İsmine İlişkin Hatalar (NameError)</a> .....	90
<a href="#">Sözdizimine İlişkin Hatalar (SyntaxError)</a> .....	91
<a href="#">Hata Kodu Vermeden Hata Yakalama</a> .....	92
<a href="#">Hatalarla Başetmede "pass" İfadesi</a> .....	94
<a href="#">Karakter Dizilerinin Metotları</a> .....	95
<a href="#">Giriş</a> .....	95
<a href="#">Kullanılabilir Metotları Listelemek</a> .....	96
<a href="#">capitalize metodu</a> .....	98
<a href="#">upper metodu</a> .....	99
<a href="#">lower metodu</a> .....	101
<a href="#">swapcase metodu</a> .....	101
<a href="#">title metodu</a> .....	102
<a href="#">center metodu</a> .....	102
<a href="#">ljust metodu</a> .....	103
<a href="#">rjust metodu</a> .....	103
<a href="#">zfill metodu</a> .....	103
<a href="#">replace metodu</a> .....	104
<a href="#">startswith metodu</a> .....	105
<a href="#">endswith metodu</a> .....	106
<a href="#">count metodu</a> .....	106
<a href="#">isalpha metodu</a> .....	107
<a href="#">isdigit metodu</a> .....	108
<a href="#">isalnum metodu</a> .....	108

<a href="#">islower metodu</a>	109
<a href="#">isupper metodu</a>	110
<a href="#">istitle metodu</a>	110
<a href="#">isspace metodu</a>	111
<a href="#">expandtabs metodu</a>	112
<a href="#">find metodu</a>	112
<a href="#">rfind metodu</a>	114
<a href="#">index metodu</a>	114
<a href="#">rindex metodu</a>	115
<a href="#">join metodu</a>	115
<a href="#">translate metodu</a>	116
<a href="#">partition metodu</a>	118
<a href="#">rpartition metodu</a>	119
<a href="#">strip metodu</a>	119
<a href="#">rstrip metodu</a>	120
<a href="#">lstrip metodu</a>	120
<a href="#">splitlines metodu</a>	120
<a href="#">split metodu</a>	121
<a href="#">rsplit metodu</a>	122
<a href="#">Nesne Tabanlı Programlama – OOP (NTP)</a>	123
<a href="#">Neden Nesne Tabanlı Programlama?</a>	124
<a href="#">Sınıflar</a>	125
<a href="#">Sınıf Tanımlamak</a>	125
<a href="#">Sınıfları Çalıştırmak</a>	126
<a href="#">Örnekleme (Instantiation)</a>	128
<a href="#">Çöp Toplama (Garbage Collection)</a>	129
<a href="#">Niteliklere Değınme (Attribute References)</a>	130
<a href="#">__init__ Nedir?</a>	134
<a href="#">self Nedir?</a>	136
<a href="#">Miras Alma (Inheritance)</a>	145
<a href="#">Eski ve Yeni Sınıflar</a>	155
<a href="#">Sonuç</a>	157
<a href="#">Grafık Arayüz Tasarımı // Temel Bilgiler</a>	157
<a href="#">Giriş</a>	157
<a href="#">Pencere Oluşturmak</a>	158
<a href="#">Pencere Başlığı</a>	161
<a href="#">Renkler</a>	163
<a href="#">fg seçeneğı</a>	163
<a href="#">bg seçeneğı</a>	164
<a href="#">Yazı Tipleri (Fonts)</a>	165
<a href="#">Metin Biçimlendirme</a>	166
<a href="#">İmleçler</a>	167
<a href="#">Pencere Boyutu</a>	167
<a href="#">Grafık Arayüz Tasarımı // Pencere Araçları (Widgets)</a>	169
<a href="#">Giriş</a>	169
<a href="#">"Label" Pencere Aracı</a>	169
<a href="#">"Button" Pencere Aracı</a>	170
<a href="#">"Entry" Pencere Aracı</a>	172

<a href="#"><u>"Checkbox" Pencere Aracı</u></a> .....	175
<a href="#"><u>"Toplevel" Pencere Aracı</u></a> .....	182
<a href="#"><u>"Listbox" Pencere Aracı</u></a> .....	184
<a href="#"><u>"Menu" Pencere Aracı</u></a> .....	193
<a href="#"><u>"Text" Pencere Aracı</u></a> .....	198

---

[Bu alışmanın Pardus Forumu'ndaki tartışma adresi](#)

e.posta: kistihza [et] yahoo [nokta] com

---

Bu alışmanın güncel sürümlerine řu iki adresten ulaşabilirsiniz:

<http://wiki.pardus-linux.org/index.php/Python>

<http://www.python.quotaless.com/>



# Temel Bilgiler

## Giriş

Python, Guido Van Rossum adlı Hollandalı bir programcı tarafından yazılmış bir programlama dili. Guido Van Rossum'un neye benzediğini merak edenler şu adresi ziyaret edebilir: [\[1\]](#)

Python kelimesi aslında çoğu kişinin zannettiği gibi "Piton Yılanı" anlamına gelmiyor. "Python" ismi Guido Van Rossum'un çok sevdiği, "Monty Python" adlı altı kişilik bir İngiliz komedi grubunun "Monty Python's Flying Circus" adlı gösterisinden geliyor.

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...

## Python'u Nereden Bulabilirim?

Python hemen hemen bütün Linux dağıtımlarında kurulu olarak geliyor. Pardus'ta Python'un yüklü olduğunu biliyoruz, o yüzden Pardus kullanıyorsanız ayrıca yüklemenize gerek yok. Eğer Python'u yüklemeniz gerekirse <http://www.python.org/download/> adresinden yükleyebilirsiniz. Ancak Python GNU/Linux dağıtımlarında çok önemli bazı parçalarla etkileşim halinde olduğu için kaynaktan kurulum pek tavsiye edilmez... Hele hele Pardus gibi, sistemin belkemiğini Python'un oluşturduğu bir dağıtımda kaynaktan kurulum epeyce baş ağrıtabilir... Sözün özü, GNU/Linux sistemlerinde en pratik yol dağıtımın kendi Python paketlerini kullanmaktır...

Yukarıdaki indirme adresinde GNU/Linux kaynak kodlarıyla birlikte programın Windows sürümünü de bulabilirsiniz. Bu adresten Python'u indirmek isteyen çoğu Windows kullanıcısı için uygun sürüm "Python x.x.x Windows installer" olacaktır... İndirilen bu dosya .msi uzantılıdır. Eğer sisteminizde .msi yükleyici uygulama yoksa (muhtemelen vardır) [buradan](#) gerekli uygulamayı bilgisayarınıza indirip kurabilirsiniz.

## Python Nasıl Çalıştırılır?

Eğer KDE masaüstü kullanıyorsak Python programını çalıştırmak için ALT+F2 tuşlarına basıp çıkan ekranda

`konsole`

yazarak bir konsol ekranı açıyoruz.

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

`gnome-terminal`

Bu şekilde konsol ekranına ulaştığımızda;

`python`

yazıp "enter"e basarak Python Programlama Dili'ni başlatıyoruz. Karşımıza şuna benzer bir ekran gelmeli:

`Python 2.4.4 (#1, Oct 23 2007, 11:25:50) [GCC`

`3.4.6] on linux2 Type "help", "copyright", "credits" or "license" for more information.`

`>>>`

Bu ekranda kullandığımız Python sürümünün 2.4.4 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 konsoldan çalıştırabiliriz. Bu işlemin nasıl yapılacağını daha sonra konuşacağız.

Windows kullanıcıları ise Python komut satırına *Başlat > Programlar > Python > Python (Command Line)* yolunu takip ederek ulaşabilirler...

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

### **print komutu**

Bu komut ekrana bir şeyler yazdırmamızı sağlar. Mesela bu komutu tek başına kullanmayı deneyelim:



`print`

yazıp hemen "enter" tuşuna basıyoruz.

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!"`

yazıp "enter" tuşuna bastıktan sonra ekranda "Ben Python, Monty Python!" çıktısını görürüz.

Gördüğünüz gibi "print" komutunun ardından gelen "Ben Python, Monty Python!" ifadesini çift tırnak içinde belirtiyoruz. Eğer burada çift tırnak işaretini koymazsak veya koymayı unutursak Python bize bir hata çıktısı gösterecektir. Biz istersek çift tırnak yerine tek tırnak (') da kullanabiliriz. Ancak tek tırnak bazı yerlerde bize sorun çıkarabilir. Diyelim ki "Linux'un faydaları" ifadesini ekrana yazdırmak istiyoruz. Eğer bunu çift tırnakla gösterirsek sorun yok:

`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:

`print 'Linux'un faydaları'`

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 illa tek tırnak kullanmak istiyorsak, kodu şu hale getirmemiz gerekir:

`print 'Linux\'un faydaları'`

Buradaki "\" işareti olası bir hatadan kaçmamızı sağlar. Bu yüzden bu tür ifadelere Python dilinde "Kaçış Dizileri" (Escape Sequences) adı verilir. Python'da "print" komutunun nasıl kullanıldığını gördüğümüze göre artık Python'un başka bir özelliğini anlatmaya başlayabiliriz:

## Python'da Sayılar ve Matematik İşlemleri

Python'da henüz dört başı mamur bir program yazamamak da en azından şimdilik onu basit bir hesap makinesi niyetine kullanabiliriz! Örneğin:

$2 + 5$

$5 - 2$

$2 * 5$

$6 / 2$

İsterseniz bunların başına "print" komutu ekleyerek de kullanabilirsiniz bu işlevi. Bir örnek verelim:

```
print 234 + 546
```

Gördüğünüz gibi tamsayıları (integer) yazarken tırnak işaretlerini kullanmıyoruz. Eğer tırnak işareti kullanırsak Python yazdıklarımızı "tamsayı" (integer) olarak değil "karakter dizisi" (string), yani bir nevi "harf" olarak algılayacaktır. Bu durumu birkaç örnekle görelim:

```
print 25 + 50
```

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

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

Bu komut 25 ile 50'yi toplamak yerine, ekrana "25 + 50" şeklinde bir çıktı verecektir. Peki şöyle bir komut verirse ne olur:

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

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

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

```
2550
```

Uzun lafın kısıası, "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:

```
print 5 / 2
```

```
2
```

Ama biz biliyoruz ki 5'i 2'ye bölerseniz 2 değil 2,5 çıkar... O zaman nedir bu şimdi? Yoksa Python matematikten anlamıyor mu?! Anlıyor anlamasına ama bizim de Python'a biraz yardımcı olmamız gerekiyor. Aynı komutu bir de şöyle deneyelim:

```
print 5.0 / 2
```

```
2.5
```

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

```
print 5 / 2.0
```

```
print 5.0 / 2.0
```

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

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

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

```
from __future__ import division
```

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

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

Buraya kadar Python'da üç 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" (variables) 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 "print n" komutunu verirse ekrana yazdırılacak veri 5 olacaktır. Yani:

```
n = 5
```

```
print n
```

```
5
```

Bu "n" değişkenini alıp bununla matematik işlemleri de yapabiliriz:

```
n * 2
```

```
n / 5
```

Hatta bu "n" değişkeni, içinde bir matematik işlemi de barındırabilir:

```
n = 34 * 45
```

```
print n
```

```
1530
```

Şu örneklerle bir göz atalım:

```
a = 5
```

```
b = 3
```

```
print a * b
```

```
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" % (3, 5, 3*5)
```

Burada, kullanacağımız her bir "tamsayı" için "%s" ekliyoruz. İfadenin en sonunda da % işaretinin ardından parantez içinde bu değişkenleri teker teker tanımlıyoruz. Buna göre birinci değişkenimiz "3", ikincisi "5", üçüncüsü ise bunların çarpımı...

Bir de şu örneği inceleyelim:

```
print "%s %s'yi seviyor!" % ("Ali", "Ayşe")
```

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

## 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 girecektir. 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.

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

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

Windows kullanıcıları ise *Başlat > Programlar > Python > IDLE (Python GUI)* yolunu takip ederek IDLE adlı geliştirme aracı ile çalışabilirler.

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

kwrite

yazmaktır...

Boş kwrite belgesi karşımıza geldikten sonra ilk yapmamız gereken, ilk satıra

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

yazmak olacaktır. Bu komut sayesinde kwrite yazacağımız kodları Python'la çalıştırması gerektiğini anlayacak. Bu konuyu biraz sonra daha ayrıntılı olarak göreceğiz. Kwrite belgesinin ilk satırına yukarıda verilen ifadeyi yerleştirdikten sonra artık kodlarımızı yazmaya başlayabiliriz. Aslında metin içine kod yazmak, Python komut satırına kod yazmaktan çok farklı değil. Şimdi aşağıda verilen satırları kwrite belgesi içine ekleyelim:

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

```
a = "elma"
```

```
b = "armut"
```

```
c = "muz"
```

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

Bunları yazıp bitirdikten sonra sıra geldi dosyamızı kaydetmeye. Şimdi dosyamızı "deneme.py" adıyla herhangi bir yere kaydediyoruz. Gelin biz masaüstüne kaydedelim dosyamızı! Şu anda masaüstünde "deneme.py" adında, muhtemelen yeşil renkli, üzerinde bir yılan resmi bulunan bir dosya görüyor olmamız lazım... Gerçi uzaktan bakınca kaplumbağaya benziyor ya, neyse... Şimdi hemen bir konsol ekranı açıyoruz. (Ama python komut satırını çalıştırmıyoruz) Şu komutu vererek, masaüstüne, yani dosyayı kaydettiğimiz yere geliyoruz:

```
cd Desktop
```

Yazdığımız programı çalıştırmak için ise şu komutu verip enter'e basıyoruz:

```
python deneme.py
```

Eğer her şey yolunda gitmişse şu çıktıyı almamız lazım:

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



GNOME kullanıcıları da yukarıda anlatılan işlemi takip ederek dosyayı kaydedip çalıştırabilir.

Windows kullanıcıları ise IDLE adlı geliştirme aracını yukarıda anlattığımız şekilde açtıktan sonra File > New Window yolunu takip ederek yeni bir dosya oluşturmali, ardından yukarıda verdiğimiz kodları yazmalı, en son olarak da File > Save as... yolunu takip ederek dosyayı deneme.py adıyla herhangi bir yere kaydetmelidir... Bu arada Windows kullanıcılarının, #!/usr/bin/env python satırını yazmalarına gerek yok... Bu satır sadece GNU/Linux kullanıcılarını ilgilendiriyor. Windows kullanıcıları IDLE ile dosyayı kaydettikten sonra Run > Run Module yolunu takip ederek veya doğrudan F5 tuşuna basarak yazdıkları programı çalıştırabilir.

"python deneme.py" komutuyla programlarımızı çalıştırabiliyoruz. Peki ama acaba Python programlarını başa "python" komutu eklemeyen çalıştırmanın bir yolu var mı? İşte burada biraz önce bahsettiğimiz "#!/usr/bin/env python" satırının önemi ortaya çıkıyor... Başa "python" komutu getirmeden programımızı çalıştırmak için öncelikle programımıza "çalıştırma yetkisi" vermemiz gerekiyor. Bunu şu komut yardımıyla yapıyoruz:

**cd Desktop**

komutuyla dosyayı kaydettiğimiz yer olan masaüstüne geliyoruz.

**chmod a+x deneme.py**

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

İstersek bu işlemi şu şekilde de yapabiliriz:

- + Masaüstündeki deneme.py dosyasına sağ tıklayın
- + "özellikler" menüsüne girin
- + "izinler" sekmesi altındaki "çalıştırılabilir" seçeneğinin solundaki kutucuğu işaretleyin.

Artık komut satırında şu komutu vererek programımızı çalıştırabiliriz:

**cd Desktop**

**./deneme.py**

Peki tüm bu işlemlerin "#!/usr/bin/env python" satırıyla ne alakası var? El Cevap: Eğer bu satırı metne yerleştirmesek "./deneme.py" komutu çalışmayacaktır...

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 anlama geliyor? Bilindiği gibi, bir programın veya dosyanın "yolu", kabaca o programın veya dosyanın içinde yer aldığı dizindir.... Örneğin GNU/Linux sistemlerindeki fstab dosyasının yolu /etc/fstab'dır. Başka bir örnek vermek gerekirse, xorg.conf dosyasının yolu /etc/X11/xorg.conf'tur... Bu "yol" kelimesinin bir de daha özel bir anlamı bulunur. Bilgisayar dilinde, çalıştırılabilir dosyaların (örneğin Windows'taki .exe dosyaları ve GNU/Linux'taki .bin dosyaları çalıştırılabilir dosyalardır.) içinde yer aldığı dizinlere de özel olarak YOL adı verilir ve bu anlamda kullanıldığında "yol" kelimesi genellikle büyük harfle yazılır... İşte çalıştırılabilir dosyalar eğer YOL üstünde iseler doğrudan isimleriyle çağrılabilirler. Şimdi bu konuyu daha iyi anlayabilmek için birkaç deneme yapalım. Hemen bir konsol ekranı açıp şu komutu veriyoruz:

**echo \$PATH**

Bu komutun çıktısı şöyle bir şey olacaktır:

```
/usr/local/bin:/usr/bin:/bin:/opt/bin:/usr/i686-pc-linux-gnu/gcc-bin/3.4.6:  
/opt/sun-jre/bin:/usr/qt/3/bin:/usr/kde/3.5/bin
```

Bu çıktı bize YOL değişkeni dediğimiz şeyi gösteriyor. İşte eğer çalıştırılabilir dosyalar bu dizinlerden herhangi biri içinde ise o dosyaları isimleriyle çağırabiliyoruz. Örneğin;

**which amarok**

komutun çıktısına bir bakalım:

```
/usr/kde/3.5/bin/amarok
```

Gördüğünüz gibi amarok programının YOL'u /usr/kde/3.5/bin/amarok. Hemen yukarıda echo \$PATH komutunun çıktısını kontrol ediyoruz ve görüyoruz ki /usr/kde/3.5/bin/ dizini YOL değişkenleri arasında var... Dolayısıyla, amarok programı YOL üstündedir, diyoruz. Amarok programı YOL üstünde olduğu için, konsolda sadece "amarok" yazarak programı başlatabiliyoruz.

Şimdi eğer biz de yazdığımız programı doğrudan ismiyle çağırabilmek istiyorsak programımızı echo \$PATH çıktısında verilen dizinlerden birinin içine kopyalamamız gerekiyor. Mesela programımızı /usr/local/bin içine kopyalayalım. Tabii ki bu dizin içine bir dosya kopyalayabilmek

için root yetkilerine sahip olmalısınız. Şu komut işi halledecektir:

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

Şimdi konsol ekranında

```
deneme
```

yazınca programımızın çalıştığını görmemiz lazım.

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

```
PATH=$PATH:/home/kullanıcı_adınız/Desktop
```

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

```
echo $PATH
```

komutuyla masaüstünüzün YOL üstünde görünüp görünmediğini kontrol edebilirsiniz... Bu sayede artık masaüstünde bulunan çalıştırılabilir dosyalar da kendi adlarıyla çağrılacaklar. Ancak masaüstünü YOL'a eklediğinizde, masaüstünüz hep YOL üstünde kalmayacak, mevcut konsol oturumu kapatılınca her şey yine eski haline dönecektir.

Şimdiye kadar öğrendiklerimizi kısaca özetlemek gerekirse:

- Python programı çoğu GNU/Linux dağıtımında zaten yüklü olarak geldiği için ayrıca yüklemeye gerek yok.
- Python kodlarını yazmak için iki seçeneğimiz var. Birincisi kodlarımızı doğrudan Python komut satırına yazabiliyoruz. Python komut satırını açmak için ALT+F2 tuşlarına basıp çıkan ekrana "konsole" yazmamız, ardından da konsolda "python" komutunu vermemiz gerekiyor. Bu ekranda komutlarımızı ">>>" işaretinden hemen sonra yazacağız.
- İkinci seçeneğimiz ise bir metin düzenleyici kullanmaktır. Bazı ufak kodları denemek için komut satırı yeterli olsa da hem kodlarımızı kaydetmek hem de büyük programlarda rahat hareket edebilmek için mutlaka bir metin düzenleyici kullanmamız gerekiyor. Şu aşamada kullanım kolaylığı nedeniyle "kwrite" metin düzenleyici önerilebilir.
- ALT+F2 tuşlarına basıp "kwrite" yazarak boş bir kwrite belgesi açabiliriz. Python kodlarını yazmaya başlamadan önce, boş belgenin ilk satırına

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

yazmamız gerekiyor. Bu satır sayesinde sistemimiz, yazdığımız kodların hangi program tarafından çalıştırılacağını anlıyor.

- Python'da en temel komutlardan biri de "print" komutudur. Bu komut bizim ekrana bir şeyler yazdırmamızı sağlıyor. Örneğin şu kod, bir bilgisayar diliyle yazılabilecek en basit programdır:

```
print "Merhaba Python!"
```

Yukarıdaki kodu, değişkenleri kullanarak da yazabiliriz:

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

```
ilk_program = "Merhaba Python!"
```

```
print ilk_program
```

## Kullanıcıyla İletişim: Veri Alış-Verişi

Python'da kullanıcıdan birtakım veriler alabilmek, yani kullanıcıyla iletişime geçebilmek için iki tane fonksiyondan faydalanılır. Bunlardan öncelikle ilkinde bakalım:

### raw\_input() fonksiyonu

Bu fonksiyon yardımıyla kullanıcıların veri girişi yapmasını sağlayabiliriz. Hemen bununla ilgili bir örnek verelim. Öncelikle boş bir kwrite belgesi açalım. Her zaman yaptığımız gibi, ilk satırımızı ekleyelim belgeye:

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

Şimdi raw\_input fonksiyonuyla kullanıcıdan bazı bilgiler alacağız. Mesela kullanıcıya bir şifre sorup kendisine teşekkür edelim...:

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

```
raw_input("Lütfen parolanızı girin:")
```

```
print "Teşekkürler!"
```

Python yazdığımız kodlar içindeki Türkçe karakterler nedeniyle bize bir uyarı mesajı gösterecektir. Bu uyarı mesajını görmek istemiyorsak, programımızın içine şöyle bir kod eklememiz gerekiyor:

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

Böylelikle kullandığımız karakter tipini Python'a tanıtmış oluyoruz. Programımızın en son hali şöyle olacak:

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

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

```
raw_input("Lütfen parolanızı girin:")
```

```
print "Teşekkürler!"
```

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
```

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

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:

```
a = raw_input("Lütfen parolanızı girin:")
```

Şimdi kullanıcıya teşekkür ediyoruz:

```
print "Teşekkürler!"
```

Kullanıcı tarafından girilen parolayı ekrana yazdırmak için şu satırı ekliyoruz:

```
print a
```

Biraz önce `raw_input` fonksiyonunu neden bir değişkene atadığımızı anladınız sanırım. Bu sayede doğrudan "a" değişkenini çağırarak kullanıcının yazdığı şifreyi ekrana döküyoruz.

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 -*-
a = raw_input("Lütfen parolanızı girin:")
print "Teşekkürler!"
print a
print "Ne yazık ki doğru parola bu değil"
```

İsterseniz son satırda şu değişikliği yapabiliriz:

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

Böylelikle, "a" değişkenini, yani kullanıcının yazdığı parolayı cümlemizin içine (ya da Python'ca ifade etmek gerekirse: "karakter dizisi içine") eklemiş olduk...

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

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

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

## **input() fonksiyonu**

Tıpkı `raw_input` fonksiyonunda olduğu gibi, bu komutla 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. 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 sayı 2540 olacaktır.

Buradan çıkan sonuç şudur:

Yukarıda anlatılanlar, `raw_input()` fonksiyonunun, girilen verileri "karakter dizisi" (string) olarak; `input()` fonksiyonunun ise "tamsayı" (integer) olarak algıladığını gösteriyor. Yani eğer biz programımız aracılığıyla kullanıcılardan bazı sayılar isteyeceksek ve eğer biz bu sayıları işleme sokacaksak (çıkarma, toplama, bölme gibi...) `input` fonksiyonunu kullanmamız gerekiyor. Ama eğer biz kullanıcılardan sayı değil de "kelime" veya başka bir ifadeyle "karakter dizisi" girmesini istiyorsak `raw_input` fonksiyonunu kullanacağız. Örneğin bir hesap makinesi programı yapacaksak kullanacağımız fonksiyon `input` fonksiyonu olmalı. Eğer burada `raw_input` fonksiyonunu kullanırsak hesap makinemiz istediğimiz gibi çalışmayacak, girilen sayıları birbirleriyle toplayamayacaktır. Tıpkı bunun gibi, eğer programımız aracılığıyla kullanıcının ismini soyismini öğreneceksek, bu işlem için de `raw_input` komutunu kullanmamız gerekiyor. Mesela aşağıda `raw_input` fonksiyonuyla yazdığımız kodları siz bir de `input` fonksiyonuyla yazmayı deneyin:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
a = raw_input("Lütfen isminizi yazın:")
b = raw_input("Lütfen soyisminizi yazın:")
print a + " " + b
```

Son kodu yazarken kullandığımız " " işaretinin amacı isim ve soyismi ekrana yazdırırken arada bir boşluk bırakmaktır... Aksi halde kullanıcı isim ve soyisini girdikten sonra bunlar ekranda birbirine bitişik olarak görünecektir.

Eğer bu kodları input fonksiyonuyla yazmayı denediyseniz, Python'un ilk veri girişinden sonra hata verdiğini görmüşsünüzdür. Python'un input fonksiyonuyla bu hatayı vermemesi için tek yol, kullanıcının ismini ve soyisini tırnak içinde yazması olacaktır... Ama tabii ki normal şartlarda kimseden ismini ve soyisini tırnak içinde yazmasını bekleyemezsiniz...

## Python'da Koşula Bağlı Durumlar

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 şifre sorulur. Siz de kendinize ait kullanıcı adını ve şifreyi size verilen kutucuklara yazarsınız. **Eğer** yazdığınız kullanıcı adı ve şifre doğruysa hesabınıza erişebilirsiniz. **Yok eğer** kullanıcı adınız ve şifreniz doğru değilse, hesabınıza erişemezsiniz. Yani e. posta hesabınıza erişmeniz, kullanıcı adı ve şifreyi doğru girme koşuluna bağlıdır. Ya da şu örneği düşünelim: Diyelim ki Pardus'ta konsol ekranından güncelleme işlemi yapacaksınız. sudo pisi up komutunu verdiğiniz zaman, güncellemelerin listesi size bildirilecek, bu güncellemeleri yapmayı isteyip istemediğiniz size sorulacaktır. **Eğer** "evet" cevabı verirsiniz güncelleme işlemi başlayacaktır. **Yok eğer** "hayır" cevabı verirsiniz güncelleme işlemi başlamayacaktır. Yani güncelleme işleminin başlaması kullanıcının "evet" cevabı vermesi koşuluna bağlıdır. Biz de şimdi Python'da bu tip koşullu durumların nasıl oluşturulacağını öğreneceğiz. Bu iş için kullanacağımız üç tane ifade var: **if...**, **else:** ve **elif...**

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

**if...**

**If...** sözü İngilizce'de "eğer" anlamına geliyor. Dolayısıyla, adından da anlaşılacağı gibi, bu ifade 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 aynı ise..."

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

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, "Sen a diyorsun, b diyorsun ama, a'nın b'nin ne demek olduğunu ben bilmiyom abi!" diye bir hata mesajı verecektir...

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 muhatap 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'e 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 izlemeniz gereken genel kural şöyledir: Klavyedeki "tab" tuşuna bir kez veya "space" tuşuna dört kez basın..

Ancak bu kuralı uygularken "tab" veya "space" tuşlarına basma seçeneklerinden yalnızca birini uygulayın... Yani bir yerde "tab" tuşuna başka yerde "space" tuşuna basıp da Python'un kafasını karıştırmayın...

Şimdi de Python'da kodlar içine nasıl açıklama eklenir, biraz da bundan bahsedelim:

Diyelim ki, içerisinde bir ton 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 başladığımız bir programa aylar sonra geri dönmek istediğinizde, "Arkadaş, ben buraya böyle bir kod yazmışım zamanında ama hangi akla hizmet böyle bir şey yapmışım acaba!" 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" #Ben bir küçük cezveyim, köşe bucak gezmeyim!
```

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

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

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

Bunlar zaten ilk etapta yazmamız gereken kodlardı. Devam ediyoruz:

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-
a = 23
b = 23
```

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

Programımızı yazmaya devam ediyoruz:

```
#!/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** ifadesinin 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 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 hoşgeldiniz!"
```

Gördüğünüz gibi, burada öncelikle "parola" adlı bir değişken yarattık. (Bu arada değişkenlere ad verirken Türkçe karakter kullanmamalısınız.) Bu parola adlı değişkenin değeri, kullanıcının girmesi gereken şifre oluyor... Ardından "cevap" adlı başka bir değişken daha yaratıp raw\_input() fonksiyonunu bu değişkene atadık. Daha sonra da **if** ifadesi yardımıyla, "Eğer cevap değişkeninin değeri parola değişkeninin değeriyle aynı ise ekrana 'Parola onaylandı! Programa hoşgeldiniz!'" yazdır dedik...

Bu programı çalıştırdığımızda, eğer kullanıcının girdiği şifre "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 hoşgeldiniz!"
```

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 hoşgeldiniz!' yazdır"

## else:

"else:" ifadesi her zaman **if** ifadesi ile birlikte kullanılır. "else:" ifadesi kısaca, "if ifadesiyle 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 "Yarabbim! Ne güzel bir isim bu!"
else:
    print isim, "adını hiç sevmem!"
```

Burada yaptığımız şey şu: Öncelikle kullanıcıya, "Senin ismin ne?" diye soruyoruz (bu soruyu, "isim" adı verdiğimiz bir değişkene atadık.) Daha sonra şu cümleyi Python'caya çevirdik:

"Eğer isim değişkeninin değeri "Ferhat" ise, ekrana "Yarabbim! 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 hiç sevmem!" cümlesini yazdır."

Bu öğrendiğimiz "else:" fonksiyonu sayesinde artık kullanıcı yanlış parola girdiğinde uyarı mesajı gönderebileceğiz:

```
#!/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...** ifadesinden 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 Python'caya çevirdik:

"Kullanıcıya, bir meyve ismi yazmasını söyle. Eğer kullanıcının yazdığı isim "elma" ise, ekrana "elma bir meyvedir" çıktısı verilsin. Yok eğer kullanıcının yazdığı isim "elma" değil, ama "armut" ise ekrana "armut bir meyvedir" çıktısı verilsin. 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" ifadesini 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... ifadesini 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... ifadesi kullanarak yazarsak Python belirtilen koşulu karşılayan ilk sonucu ekrana yazdıracaktır.

Buraya kadar Python'da pek çok şey öğrenmiş olduk. If..., elif... else: ifadelerini de öğrendiğimize göre artık çok basit bir hesap makinesi yapabiliriz!

```
#!/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("Lütfen yapmak istediğiniz işlemin numarasını girin: ")

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

if soru == "2":
    sayi3 = input("Lütfen çıkarma işlemi için ilk sayıyı girin: ")
    print sayi3
    sayi4 = input("Lütfen çıkarma işlemi için ikinci sayıyı girin: ")
    print sayi3, "-", sayi4, ":", sayi3 - sayi4

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

if soru == "4":
    sayi7 = input("Lütfen bölme işlemi için ilk sayıyı girin: ")
    print sayi7
    sayi8 = input("Lütfen bölme işlemi için ikinci sayıyı girin: ")
    print sayi7, "/", sayi8, ":", sayi7 / sayi8
```

# Python'da Döngüler

Bir önceki bölümün sonunda hatırlarsanız 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 şifre yanlış girildiğinde bu kullanıcı adı ve şifreyi tekrar tekrar soramayız; programı yeniden başlatmamız gerekir... İşte bu bölümde Python'da yazdığımız kodları sürekli hale getirmeyi, tekrar tekrar döndürmeyi öğreneceğiz.

Kodlarımızı "sürekli döndürmemizi" sağlamada bize yardımcı olacak parçacıklara Python'da "döngü" (İngilizce: Loop) adı veriliyor... Bu bölümde iki tane döngüden bahsedeceğiz: "while" ve "for" döngüleri. Ayrıca bu bölümde döngüler dışında "break" ve "continue" ifadeleri ile range() ve len() fonksiyonlarına da değineceğiz.

## while döngüsü

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. Şu küçük örnek bir inceleyelim bakalım:

```
#!/usr/bin/ env python
#-*- coding: utf-8 -*-
a = 0
a = a + 1
print a
```

Bu minicik kodun yaptığı iş, birinci satırda "a" değişkeninin değerine bakıp ikinci satırda bu değere 1 eklemek, üçüncü satırda da bu yeni değeri ekrana yazdırmaktır.. Dolayısıyla bu kod parçasının vereceği çıktı da, "1" olacaktır. Bu çıktıyı verdikten sonra ise program sona erdirilecektir. 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:

ilk satırda, "0" değerine sahip bir "a" değişkeni tanımladık.

ikinci 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)

Başta 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 işlemlere (operators) değinelim:

Şimdiye kadar aslında bu işlemlerden 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

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("Lütfen yapmak istediğiniz işlemin numarasını girin: ")
```



```
if soru == "1":  
    sayi1 = input("Lütfen toplama işlemi için ilk sayıyı girin: ")  
    print sayi1  
    sayi2 = input("Lütfen toplama işlemi için ikinci sayıyı girin: ")  
    print sayi1, "+", sayi2, ":", sayi1 + sayi2
```

```
if soru == "2":  
    sayi3 = input("Lütfen çıkarma işlemi için ilk sayıyı girin: ")  
    print sayi3  
    sayi4 = input("Lütfen çıkarma işlemi için ikinci sayıyı girin: ")  
    print sayi3, "-", sayi4, ":", sayi3 - sayi4
```

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

```
if soru == "4":  
    sayi7 = input("Lütfen bölme işlemi için ilk sayıyı girin: ")  
    print sayi7  
    sayi8 = input("Lütfen bölme işlemi 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 ufaklık 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("Lütfen yapmak istediğiniz işlemin numarasını girin: ")
```

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

```
    sayi1 = input("Lütfen toplama işlemi için ilk sayıyı girin: ")  
    print sayi1  
    sayi2 = input("Lütfen toplama işlemi için ikinci sayıyı girin: ")  
    print sayi1, "+", sayi2, "=", sayi1 + sayi2
```

```
if soru == "2":
```

```
    sayi3 = input("Lütfen çıkarma işlemi için ilk sayıyı girin: ")  
    print sayi3  
    sayi4 = input("Lütfen çıkarma işlemi için ikinci sayıyı girin: ")  
    print sayi3, "-", sayi4, "=", sayi3 - sayi4
```

```
if soru == "3":
```

```
    sayi5 = input("Lütfen çarpma işlemi için ilk sayıyı girin: ")  
    print sayi5  
    sayi6 = input("Lütfen çarpma işlemi için ikinci sayıyı girin: ")  
    print sayi5, "x", sayi6, "=", sayi5 * sayi6
```

```
if soru == "4":
```

```
    sayi7 = input("Lütfen bölme işlemi için ilk sayıyı girin: ")  
    print sayi7
```

```
sayi8 = input("Lütfen bölme işlemi 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" ifadesi "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.

Şu örneğe bir bakalım:

```
#!/usr/bin/env python  
#-*- coding: utf-8 -*-  
soru = raw_input("Arkadaşım sen deli misin?")  
while soru != "hayır":  
    print "delisin sen deli! Kulakları küpeli!"
```

Dikkat ederseniz burada da işleçlerimizden birini kullandık. Kullandığımız işleç "eşit değildir" anlamına gelen "!=" işleci...

Bu programı çalıştırdığımızda sorulan soruya "hayır" cevabı vermezsek, program biz müdahale edene kadar ekrana "delisin sen deli! Kulakları küpeli" çıktısını vermeye devam edecektir... Çünkü biz Python'a şu komutu vermiş olduk bu kodla:

"Soru değişkeninin cevabı "hayır" olmadığı müddetçe ekrana "delisin sen deli! Kulakları küpeli" çıktısını vermeye devam et."

Eğer bu programı durdurmak istiyorsak CTRL+C'ye 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("Arkadaşım sen deli misin?")
if soru != "hayır":
    print "delisin sen deli! Kulakları küpeli!"
```

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

### **for döngüsü**

Bir önceki konuda while döngülerini 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 yardımıyla da yapabiliriz:

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-
for zombi in range(1, 100):
    print zombi
```

Ben burada değişken adı olarak zombi kelimesini kullandım, siz isterseniz Osman da diyebilirsiniz...

Yukarıdaki Pythonca kod Türkçe'de aşağı yukarı şu anlama gelir:

"1, 100 aralığındaki sayılara zombi adını verdikten sonra ekrana zombi'nin değerini yazdır."

for döngüsüyle ilgili şu örneğe de bir bakalım:

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-
for kelimeler in "linux":
    print kelimeler
```

Gördüğünüz gibi, for döngüsüyle yalnızca sayıları değil, karakter dizilerinin öğelerini de dökabiliyoruz ekrana.

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() fonksiyonu**

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. 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() fonksiyonu

Bu fonksiyon, karakter dizilerinin uzunluğunu gösterir. Mesela:

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-
a = "Afyonkarahisar"
print len(a)
```

Bu kod, "Afyonkarahisar" karakter dizisi içindeki harflerin sayısını ekrana dökcektir.

Bu fonksiyonu nerelerde kullanabiliriz? Mesela yazdığınız bir programa kullanıcıların giriş yapabilmesi için şifre belirlemelerini istiyorsunuz. Seçilecek şifrelerin uzunluğunu sınırlamak istiyorsanız bu fonksiyondan yararlanabilirsiniz. Hemen örnek bir kod yazalım:

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-
a=raw_input("Lütfen bir şifre belirleyin: ")
if len(a) >= 6:
    print "Seçtiğiniz şifre en fazla 5 karakterden oluşabilir!"
else:
    print "Şifreniz etkinleştirilmiştir."
```

## break ifadesi

break ifadesi program içinde bir noktada programı sona erdirmek gerektiği zaman kullanılır. Aşağıdaki örnek break ifadesinin ne işe yaradığını açıkça gösteriyor:

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-
kullanici_adi = "kullanici"
parola = "sifre"
while True:
    soru1 = raw_input("Kullanıcı adı: ")
    soru2 = raw_input("Şifre: ")
    if soru1 == kullanici_adi and soru2 == parola:
        print "Kullanıcı adı ve şifreniz onaylandı. Programa hoşgeldiniz!"
```

```
break
```

```
else:
```

```
    print "Kullanıcı adınız veya şifrenizden en az birisi onaylanmadı. Lütfen tekrar deneyiniz!"
```

Bu programda break ifadesi yardımıyla, kullanıcı adı ve şifre doğru girildiğinde şifre sorma işleminin durdurulması sağlanıyor. Yukarıdaki kodlar arasında, dikkat ederseniz, daha önce bahsettiğimiz işleçlerden birini daha kullandık. Kullandığımız bu işleç, "ve" anlamına gelen "and" işleci. Bu işlecin geçtiği satıra tekrar bakalım:

```
if soru1 == kullanıcı_adi and soru2 == parola:
```

```
    print "Kullanıcı adı ve şifreniz onaylandı. Programa hoşgeldiniz!"
```

Burada şu Türkçe ifadeyi Python'caya çevirmiş olduk:

"Eğer soru1 değişkeninin değeri kullanıcı\_ad değişkeniyle aynı ve soru2 değişkeninin değeri parola değişkeniyle aynı ise ekrana 'Kullanıcı adı ve şifreniz onaylandı. Programa hoşgeldiniz,' 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 parola'dan biri yanlışsa "if" ifadesinin gerektirdiği koşul yerine gelmemiş olacaktır... Okulda mantık dersi almış olanlar bu "and" işlecinin 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" dersiniz, burada hem a'nın hem de b'nin doğru olması gerekir...

## **continue ifadesi**

Bu ifade ise döngü içinde bir bloğun es geçilip ondan sonraki bloğun çalıştırılmasını 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 girebilirsiniz."
```

Burada eğer kullanıcı klavyede "iptal" yazarsa programdan çıkılacaktır. Bunu,

```
if s == "iptal":  
    break
```

satırıyla sağlamayı başardık.

Eğer kullanıcı tarafından girilen sayı üç haneli veya daha az haneli bir sayı ise, "continue" ifadesinin etkisiyle

```
print "En fazla üç haneli bir sayı girebilirsiniz."
```

satırı es geçilecek ve en başa dönecektir.

Eğer kullanıcının girdiği sayıdaki hane üçten fazlaysa ekrana:

```
print "En fazla üç haneli bir sayı girebilirsiniz."
```

cümlesi yazdırılacaktır.

## Python'da Listeler, Demetler, Sözlükler

Bu bölümde Python'da nasıl liste (list), demet (tuple) ve sözlük (dictionary) hazırlanacağını öğreneceğiz. Burada göstereceğimiz küçük kod parçalarını metin dosyası yerine doğrudan Python komut satırında deneyebilirsiniz. Böylesi hem daha hızlı, hem daha kolay, hem de daha etkili olacaktır.

İlk konumuz listeler:

### Listeler

Python'da herhangi bir liste oluşturmak için önce listemize bir ad vermemiz, ardından da köşeli parantez kullanarak 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.

Yukarıda da belirttiğimiz gibi, hızlilik açısından burada öğreneceğimiz kodları Python komut satırına yazıp deneyeceğiz. Python komut satırını, hatırlarsanız, şöyle açıyorduk: ALT+F2 tuşlarına



basıp çıkan pencerede "konsol" yazıyoruz ve konsol ekranında "python" komutu vererek Python komut satırını başlatıyoruz. 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 (strings) her zamanki gibi tırnak içinde belirtmeyi unutmuyoruz... Tabii ki sayıları (integers) 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 dizisi".

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, "tamsayı".

Şimdi bu 123 ifadesini tırnak içinde yazalım:

```
type("123")
```

Sonuç:

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

Gördüğünüz gibi yazdığınız şey sayı da olsa, siz bunu tırnak içinde belirtirseniz, Python gözünüzün yaşına bakmıyor...

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 len() fonksiyonu yardımıyla öğrenebiliriz:

```
len(liste)
```

```
5
```

Şimdi listeleri yönetmeyi; yani listeye öğe ekleme, listeden öğe çıkarma gibi işlemleri yapmayı öğreneceğiz. Bu işi Python'da bazı parçacıklar yardımıyla yapıyoruz. İsterseniz gelin şimdi bu parçacıkların neler olduğuna ve nasıl kullanıldıklarına bakalım.

### **append parçacığı**

İlk parçacığımız "append". Bu kelime Türkçe'de "eklemek, iliştmek" anlamına geliyor...

Oluşturduğumuz listeye yeni bir öğe eklemek için "append" parçacığından faydalanıyoruz:

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

Dikkat edin, liste tanımlarken köşeli parantez kullanıyorduk... Listeleri yönetirken ise (yani parçacıkları kullanarak ekleme, çıkarma, vb. yaparken) normal parantezleri kullanıyoruz. Ayrıca gördüğünüz gibi, bu "append" parçacığını, liste isminin yanına koyduğumuz bir noktadan sonra yazıyoruz.

"append" parçacığı yardımıyla, öğeyi oluşturduğumuz bir listenin en sonuna ekleyebiliyoruz. Peki bu parçacık yardımıyla birden fazla öğe ekleyebilir miyiz? Ne yazık ki, append parçacığı 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 parçacıktan faydalanıyoruz. Ama bu parçacığı 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 içindeki öğeleri sıralarken, listenin ilk öğesini 0'dan başlatır."

Yani:

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

biçiminde gördüğümüz listenin ilk ögesine "0'ıncı öge" denir. Bu listedeki birinci öge ise "Jale"dir. Bunu şu şekilde teyit edelim:

```
liste[0]
```

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

```
liste[2]
```

komutu ise bize 2. ögenin "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 öge bulunuyor.

Python'un öge sıralama mantığını öğrendiğimize göre, şimdi listenin en sonuna değil de kendi belirleyeceğimiz başka bir noktaya öge eklememizi sağlayacak parçacığı görebiliriz:

### **insert parçacığı**

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

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

Burada parantez içindeki ilk sayı, "Ahmet" ögesinin liste içinde yerleştirileceği sırayı gösteriyor. 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 öge eklemek istiyorsak şu komutu kullanacağız:

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

Bu parçacık da tıpkı append parçacığında olduğu gibi listeye yalnızca bir adet öge eklememize izin verir...

## extend parçacığı

Bu kelime "genişletmek, uzatmak" anlamına geliyor. extend parçacığı, oluşturduğumuz listeleri "genişletmemizi" veya "uzatmamızı" sağlar. Bu parçacığın 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)
```

Bu komutun çıktısı şöyle olacaktır:

```
['Veli', 'Hale', 'Ahmet', 'Jale', 'Lale', 12, 23, 'Mehmet', 'Simovic', 'Prekazi', 'Jardel', 'Nouma']
```

Gördüğünüz gibi, extend parçacığı iki listenin öğelerini tek bir liste içinde birleştirmeye yarıyor. Ya da başka bir ifadeyle, bir listeyi genişletiyor, uzatıyor...

extend parçacığıyla yaptığımız işlemin aynısını "+" işlecini kullanarak şu şekilde de yapabiliriz:

```
liste + yeni_liste
```

## remove parçacığı

Liste oluşturmayı, append ve insert parçacıkları yardımıyla listeye öğeler eklemeyi öğrendik... Peki ya listemizden bazı öğeleri nasıl çıkaracağız? Python'da bu işi yapmamızı sağlayan iki tane parçacık var. Biz önce bunlardan ilki olan remove parçacığına bakacağız. Bu kelime "çıkarmak, kaldırmak, silmek" anlamına geliyor.

Diyelim ki listemizden "Nouma" öğesini çıkarmak/kaldırmak istiyoruz. O zaman şu komutu vermemiz gerekir:

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

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

## pop parçacığı

İngilizce'de "pop" kelimesi, "fırlamak, pırtlamak, aniden açılmak" gibi anlamlar taşıyor. Biz bu kelimeyi internette bir adrese tıkladığımızda pırt diye önümüze çıkan "pop up"lardan yani "açılır

pencereler"den hatırlıyoruz... Python'da listeler ile birlikte kullandığımız "pop" parçacığı ise listeden bir öge silerken, bu sildiğimiz öğenin pırt diye ekrana yazdırılmasını sağlıyor...

Şu komutu deneyelim:

`liste.pop()`

Gördüğünüz gibi, Python bu pop parçacığı yardımıyla listenin son ögesini çıkaracak, üstelik çıkardığı öğeyi ekrana yazdıracaktır. Eğer bu komutu şöyle verirsek ne olur?

`liste.pop(0)`

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

Şimdiye kadar bir listenin **en sonuna** nasıl öge ekleyeceğimizi (append parçacığı), listenin **herhangi bir yerine** nasıl öge ekleyeceğimizi (insert parçacığı), listeden **isim vererek** nasıl öge çıkaracağımızı (remove parçacığı), listeden **sayı vererek** nasıl öge çıkaracağımızı (pop parçacığı) öğrendik.

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

### **index parçacığı**

Diyelim ki listedeki "Jardel" ögesinin listenin kaçınıcı sırasında olduğunu merak ediyorsunuz. İşte bu index parçacığı sizin aradığınız şey! Bu parçacığı şöyle kullanıyoruz:

`liste.index("Jardel")`

Bu komut, "Jardel" ögesinin liste içinde kaçınıcı sırada olduğunu gösterecektir bize...

### **sort parçacığı**

Bazen listemizdeki öğeleri alfabe sırasına dizmek isteriz. (isteriz, değil mi?) İşte yüreğimizde böyle

bir istek hasıl olduğunda kullanacağımız parçacığın adı "sort":

`liste.sort()`

### **reverse parçacığı**

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

`liste.reverse()`

Bu komutu üst üste iki kez verirsiniz listeniz ilk haline dönecektir. Yani bu komut aslında sort parçacığının yaptığı gibi alfabe sırasını kaale almaz... Listenizdeki öğelerin sırasını ters çevirmekle yetinir.

### **count parçacığı**

Listelerle birlikte kullanabileceğimiz başka bir parçacık 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'ıncı" 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 düşüğü 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 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 dahil 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 öge yerleştirmek, listeden öge silmek, vb. de mümkündür. Yani yukarıda "parçacıklar" 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 öge eklemek için (append parçacığına benzer...):

```
liste[5:5] = ["domates", "salata"]  
['elma', 'armut', 'kiraz', 'karpuz', 'kavun', 'domates', 'salata']
```

Hatırlarsanız, append parçacığıyla listeye yalnızca bir adet öge ekleyebiliyorduk.. Yukarıdaki yöntem yardımıyla birden fazla öge de ekleyebiliyoruz listeye.

Bu listenin 3. sırasına bir veya birden fazla öge yerleştirmek için (insert parçacığına benzer...):

```
liste[3:3] = ["kebab", "lahmacun"]  
['elma', 'armut', 'kiraz', 'kebab', 'lahmacun', 'karpuz', 'kavun', 'domates', 'salata']
```

Bu listenin 2. sırasındaki ögeyi silmek için (remove parçacığına benzer...):

```
liste[2:3] = []  
['elma', 'armut', 'kebab', 'lahmacun', 'karpuz', 'kavun', 'domates', 'salata']
```

Bu listenin 2. sırasındaki ögeyi silip yerine bir veya birden fazla öge eklemek için:

```
liste[2:3] = ["nane", "limon"]  
['elma', 'armut', 'nane', 'limon', 'kebab', 'lahmacun', 'karpuz', 'kavun', 'domates', 'salata']
```

Bu listenin 2. sırasındaki ögeyi silip yerine bir veya birden fazla ögeye sahip bir liste yerleştirmek için:

```
liste[2] = ["ruj", "maskara", "rimel"]  
['elma', 'armut', ['ruj', 'maskara', 'rimel'], 'nane', 'limon', 'kebab', 'lahmacun', 'karpuz',  
'kavun', 'domates', 'salata']
```

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 (tuples) listelere benzer. Ama listeler ile aralarında çok temel bir fark vardır. Listeler üzerinde oynamalar yapabiliriz. Yani öge ekleyebilir, öge 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:

```
hede = ("inek",)
```

Gördüğünüz gibi, tek öğeli bir demet oluşturabilmek için ögenin yanına bir virgül koyuyoruz!

Hemen teyit edelim:

```
type(hede)
```

```
<type 'tuple'>
```

O virgülü koymazsak ne olur?

```
hede2 = ("inek")
```

hede2'nin tipini kontrol edelim:

```
type(hede2)
```

<type 'str'>

Demek ki, virgüli koymazsak demet değil, alade bir karakter dizisi oluşturmuş oluyoruz...

Yukarıda anlattığımız şekilde bir demet oluşturma işine "demetleme" (tuple packing) adı veriliyor. Bunun tersini de yapabiliriz. Buna da "demet açma" deniyor (sequence unpacking):

Önce demetleyelim:

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

Şimdi demeti açalım:

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

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

Peki bu demetler ne işe yarar? 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...

## Sözlükler

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

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

Burada örneğin, "elma" bir "anahtar", "meyve" ise bu anahtarın "değeri"dir. Aynı şekilde "sebze" değerinin anahtarı "domates"tir.

Dolayısıyla Python'da sözlük; "anahtar" ve "değer" arasındaki ilişkiyi gösteren bir veri tipidir! Mesela bir adres veya telefon defteri yaratmak istediğimizde bu sözlüklerden faydalanabiliriz. Yani "sözlük" denince aklımıza sadece bildiğimiz sözlükler gelmemeli... Şu örneğe bir bakalım:

```
telefon_defteri = {"Ahmet": "0533 123 45 67",  
"Kezban": "0532 321 54 76", "Feristah": "0533 333 33 33"}
```

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["Kezban"]
```

Bu komutlar "Ahmet" ve "Kezban" adlı "anahtar"ların karşısında hangi "veri" varsa onu ekrana yazdıracaktır... Dikkat edin, sözlükten öge çağırırken küme parantezlerini değil, köşeli parantezleri kullanıyoruz. Bu arada aklınızda bulunsun, sözlük içindeki öğeleri "anahtar"a göre çağırıyoruz, "veri"ye 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 öge eklemek istiyoruz:

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

Peki sözlüğümüzdeki bir ögenin değerini değiştirmek istersek ne yapacağız?

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

Buradan anladığımız şu: Bir sözlüğe yeni bir öge eklerken de, varolan bir ögeyi değiştirirken de aynı komutu kullanıyoruz... Demek ki bir ögeyi değiştirirken aslında ögeyi değiştirmiyor, silip yerine yenisini koyuyoruz...

Eğer bir ögeyi listeden silmek istersek şu komutu kullanıyoruz:

```
del telefon_defteri["Kezban"]
```

Eğer biz sözlükteki bütün öğeleri silmek istersek şu komut kullanılıyor:

```
telefon_defteri.clear()
```

## **if-elif-else Yerine Sözlük Kullanmak**

Şimdi isterseniz, Python sözlüklerinin pratikliğini bir örnek yardımıyla görmeye çalışalım:

Diyelim ki bir havadurumu programı yazmak istiyoruz. Tasarımıza göre kullanıcı bir şehir adı girecek. Program da girilen şehre özgü havadurumu bilgilerini ekrana yazdıracak. Bunu yapabilmek için, daha önceki bilgilerimizi de kullanarak şöyle bir şey yazabiliriz:

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

```
#-*-coding:utf8-*-
```

```
soru = raw_input("Bulunduğunuz şehrin 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 havadurumu bilgilerini sorgulayacak olsak mesele değil, ancak onlarca şehri kapsayacak bir program üretmekse amacımız, yukarıdaki yöntem yerine daha pratik bir yöntem uygulamak akıl sağlığımız için de gayet yerinde bir tercih olacaktır.

İşte bu noktada programcının imdadına Python'daki sözlük veritipi yetişecektir. Yukarıdaki kodların yerine getirdiği işlevi, şu kodlarla da gerçekleştirebiliriz:

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

```
#-*-coding:utf8-*-
```

```
soru = raw_input("Bulunduğunuz şehrin 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 havadurumu bilgilerini bir sözlük içinde depoluyoruz.

Daha sonra, sözlük metotlarından biri olan "get" metodunu seçiyoruz. Bu metot bize sözlük içinde bir değer var olup olmadığını denetleme imkanının yanısıra, adı geçen değer sözlük içinde varolmaması 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 havadurumu bilgisi bulunmamaktadır.")
```

satırı yardımıyla "soru" adlı değişkenin değerinin sözlük içinde var olup var 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 havadurumu 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 imkanı 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 havadurumu bilgileri tanımlamaya

kıyasla sözlük içinde yeni "anahtar-değer" çiftleri oluşturmak daha pratiktir.

Böylelikle Python'da Listeler, Demetler ve 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 Fonksiyonlar

### Giriş

Bazen kodları her defasında tekrar tekrar yazmak yerine, bir kez yazıp tekrar tekrar kullanabilmek isteriz. Eğer böyle bir şey istiyorsak Python'da fonksiyonları kullanmamız gerekiyor. Python'un içinde gömülü vaziyette duran onlarca fonksiyon bulunur. Mesela `len()` ve `range()` birer fonksiyondur.

### Fonksiyon Tanımlama

Python, kendisinde var olan fonksiyonların yanısıra bizim de yeni fonksiyon tanımlamamıza olanak tanır. Fonksiyonları tanımlarken "def" adlı bir parçacıktan yararlanacağız. Bu "def" parçacığının hemen ardından ise fonksiyonumuzun adını belirlememiz gerekiyor... Bu adı belirlerken Türkçe karakter kullanmamamız gerektiğini hatırlatmaya gerek yok herhalde.

Hemen bir örnek verelim:

```
def ilk_fonksiyonumuz():
```

Gördüğünüz gibi önce "def" parçacığı yardımıyla Python'a bir fonksiyon tanımlayacağımızı bildirdik. Daha sonra bu fonksiyona "ilk\_fonksiyonumuz" adını verdik. Tabii siz isterseniz fonksiyonunuza başka adlar da verebilirsiniz. Bu arada ifadenin en sonuna iki nokta üst üste eklemeyi de unutmuyoruz.

Şu anda fonksiyonumuzun fonksiyon olduğu ve adının ne olduğu belli. Ama ne yaptığı, ne işe yaradığı belli değil... O halde devam edelim:

```
def ilk_fonksiyonumuz():
```

```
    print "Ben Python! Monty Python!"
```

Aslında şu an itibariyle ilk fonksiyonumuzu tanımlamış bulunuyoruz. İsterseniz bu kodu Python komut satırında yazabiliriz. Peki nasıl?

## Komut Satırında Fonksiyon Tanımlama

Her zamanki gibi ALT+F2 tuşlarına basıp, çıkan ekrana "konsole" yazıyoruz ve siyah ekranda "python" yazarak Python komut satırını başlatıyoruz.

Komut satırında:

```
def ilk_fonksiyonumuz():
```

yazıp enter'e bastığımızda "..." işaretini görürüz. Bu, Python'un bizden yeni bir satır yazmamızı beklediği anlamına geliyor. Şimdi de

```
print "Ben Python! Monty Python!"
```

satırını yazacağız. Ama bu satırı girintili yazmamız gerekiyor. "Space" tuşuna dört kez basarak girintileme işlemini yapıyoruz. Şimdi;

```
print "Ben Python! Monty Python!"
```

satırını yazabiliriz.

Bu satırı da yazdıktan sonra iki kez enter tuşuna basarak ">>>" işaretinin belirmesini bekliyoruz.

Bu işareti gördükten sonra:

```
ilk_fonksiyonumuz()
```

yazıp enter'e bastığımızda fonksiyonumuzun içeriği ekrana yazılacaktır...

## Metin Düzenleyicide Fonksiyon Tanımlama

Peki ya bu fonksiyonu bir metin dosyasına kaydetmek istersek ne yapacağız? (ki doğrusu da budur aslında)

Hemen boş bir kwrite dosyası açıp içine şunları yazıyoruz:

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

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

```
def ilk_fonksiyonumuz():
```

```
    print "Ben Python! Monty Python!"
```

Şimdi son bir satır daha eklememiz gerekiyor. Komut satırında yazarken kodumuz bu satır olmadan da çalışıyordu, ama metin dosyasına şöyle bir satır eklememiz gerekiyor:

```
ilk_fonksiyonumuz()
```

Metnimiz en son şöyle görünecek:

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-
def ilk_fonksiyonumuz():
    print "Ben Python! Monty Python!"
ilk_fonksiyonumuz()
```

Bu son satır, yazdığımız fonksiyonu çağırmanızı sağlıyor. Bu metin dosyasını ilk\_fonksiyon.py adıyla bilgisayarımıza kaydediyoruz. Artık konsol'da

```
python ilk_fonksiyonumuz.py
```

komutuyla fonksiyonumuzu çalıştırabiliriz.

## Fonksiyonlarda Parametreler

Gördüğünüz gibi tanımladığımız fonksiyonda içi boş parantezler kullandık: "def ilk\_fonksiyonumuz()"

Bu parantezlerin içine, yerine göre bazı ifadeler, yani parametreler ekleme şansımız da var... Ancak bu parantezlerin içi boş da olsa, dolu da olsa fonksiyon tanımlarken bu parantezleri koymayı unutmamamız gerekiyor. Peki bu parantezlerin içine neler yazabiliriz? Mesela çarpma yapan bir fonksiyon tanımlayacaksak eğer, birbirleriyle çarpılacak değerleri yazabiliriz. Tabii bu değerler mutlaka sayı cinsinden olacak diye bir kaide yok... Biz değerlerimizi "değişken" olarak da tanımlayabiliriz:

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-
def carpma(a, b):
    print a * b
carpma(124, 345)
```

Dikkat ederseniz, son satırda, "124" ve "345" sayılarını kullanarak sırasıyla "a" ve "b" değişkenlerini tanımlamış olduk. Bu fonksiyon çalıştırıldığında ekrana 124 x 345 işleminin sonucu yazdırılacaktır.

Yukarıdaki örnekte değişkenlerin değerini (124 ve 345) son satırda parantez içine yazarak bunları doğrudan tanımlamış olduk. Ama biz istersek bu değişkenleri ayrı olarak da tanımlayabiliriz:



```
#!/usr/bin/env python
#-*- coding:utf-8 -*-
def carpma(a, b):
    print a * b
a = 124
b = 345
carpma(a, b)
```

Burada a ve b değişkenlerini print ifadesinin alt hizasına yazmadığımıza dikkat edin.. Eğer bu değerleri print ifadesinin alt hizasına yazarsak Python bize pislik yapacak, değişkenleri tanımlamadığımızı iddia edecektir..

Ama istersek yukarıdaki kodu şu şekilde de yazabiliriz:

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-
def carpma(a, b):
    a = 124
    b = 345
    print a * b
carpma(a, b)
```

Gördüğünüz gibi, a ve b değişkenlerini fonksiyonu tanımladıktan hemen sonra belirlersek print ifadesinin üst hizasına yazabiliyoruz.. Tabii şimdi bunları satırın en başına alırsanız Python bu defa da size başka bir bahaneyle pislik yapacak, "Abi bizim buralarda iki nokta üst üste koyup enter'e bastıktan sonraki ifade girintili yazılır.. Şimdi eski köye yeni adet getirme," diye bir hata mesajı verecektir.

Şimdi başka bir örnek daha görelim:

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-
def liste_kontrol(liste):
    if liste == []:
        print "listeniz şu anda boş"
    else:
```

```
print "listenizde şu anda", liste, "adlı öğeler görünüyor"
liste = ["ahmet", "mehmet"]
liste_kontrol(liste)
```

Bu örnekte de liste öğelerini ayrı olarak tanımladık. Tabii biraz önceki örnekte olduğu gibi istersek bu öğeleri fonksiyonun içine de yedirebiliriz:

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-
def liste_kontrol(liste):
    if liste == []:
        print "listeniz şu anda boş"
    else:
        print "listenizde şu anda", liste, "adlı öğeler görünüyor"
liste_kontrol(liste = ["ahmet", "mehmet"])
```

Daha önce anlattığımız gibi, bu fonksiyonları metin düzenleyici yerine doğrudan Python komut satırına da yazabiliriz. Eğer bu kodları komut satırına yazarsak, mesela yukarıdaki çarpma fonksiyonunu çağırırken çarpılacak değerleri isteğimize göre değiştirebiliriz. Hemen deneyelim:

Python komut satırını açıp ">>>" işaretinden hemen sonra;

```
def carpma(a, b):
```

yazıp enter'e basıyoruz. Şimdi dört kez "space" tuşuna basarak girinti veriyoruz. Girintiyi verdikten sonra şu kodu yazıyoruz:

```
print a * b
```

Hemen ardından iki kez enter tuşuna basarak ">>>" işaretinin tekrar belirmesini sağlıyoruz. Şimdi de mesela şu komutu yazarak az evvel tanımladığımız fonksiyonumuzu çalıştırıyoruz:

```
carpma(45, 56)
```

Yukarıdaki komutta 45 ile 56 sayısını çarpmış olduk. İsterseniz parantez içindeki bu değerleri, yani parametreleri, istediğimiz gibi değiştirebiliriz. Mesela şöyle yapabiliriz:

```
carpma(555, 444)
```

Şimdi şöyle bir şey deneyelim. Önce boş bir kwrite belgesi açalım ve içine şunları yazalım:

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

```
#-*- coding:utf-8 -*-  
def carpma(a, b):  
    print a * b
```

Gördüğünüz gibi bu kod yukarıda yazdığımız kodla aynı... Tek fark en sonda "carpma(124, 345)" gibi bir ifade kullanmadık. Bu dosyayı "fonk.py" adıyla kaydedelim.

Şimdi de başka bir boş kwrite belgesi açıp içine şunları yazalım:

```
#!/usr/bin/env python  
#-*- coding:utf-8 -*-  
from fonk import carpma  
carpma(34,45)
```

Bu dosyayı da istediğimiz bir adla kaydedebiliriz. Mesela "deneme.py" diyelim...

Şimdi bu "deneme.py" adlı dosyayı çalıştırdığımız zaman 34 ve 45 sayılarının birbirleriyle çarpıldığını görürüz... Halbuki deneme dosyası içinde herhangi bir çarpma işlemi tanımlamadık... Durum böyle olduğu halde çarpma yapabilmemizin nedeni daha önce tanımladığımız "fonk" isimli fonksiyonu "deneme" isimli belgemize "davet etmiş" veya "yüklemiş" olmamız... Ya da **İngilizce** olarak ifade edersek, "import etmiş" olmamız... Hatırlarsanız o "fonk" isimli fonksiyon içinde zaten bir çarpma işlemi tanımlamıştık... Dolayısıyla "deneme" isimli belgede tekrar çarpma işlemi tanımlamamıza gerek kalmadı... Peki bu "davet etme" işini nasıl yaptık?

Dikkat ederseniz, "deneme.py" isimli dosyamızın üçüncü satırı şöyle:

```
from fonk import carpma
```

Bu Python'ca ifadenin Türkçesi şu: "fonk isimli modülden çarpma isimli fonksiyonu yükle (veya davet et veya import et)"

Bizim konumuz "Python'da Fonksiyonlar" idi... Peki şimdi bu "modül" lafı da nereden çıktı... Bu durumu şöyle açıklayabiliriz:

Bu bölümün en başında, "def" parçacığını kullanarak Python'da fonksiyon tanımlamayı öğrenmiştik. Aynı belge içinde bu "def" parçacığını kullanarak pek çok fonksiyon tanımlayabiliriz. İşte bu bir veya birden fazla fonksiyon hep birlikte "modül" denen şeyi oluşturuyor... Yani aslında "modül", fonksiyonlardan oluşmuş bir bütünün adı... Python'da modül kavramını bir sonraki bölümde daha ayrıntılı bir şekilde inceleyeceğiz... Bizim örneğimizde "def" parçacığıyla

tanımlanmış "carpma" isimli blok bir fonksiyon; dosyamızın adı olan "fonk" ise bir modül olmuş oluyor...

Bu arada dikkat edin, yukarıdaki modülü ilk yüklediğimizde, çalışma klasörümüzde .pyc uzantılı bir dosya daha oluştu. Python bu dosyayı oluşturarak modülün bir dahaki sefere daha hızlı yüklenmesini sağlıyor...

Şimdi konumuza dönelim... Ne demiştik:

"fonk isimli modülden carpma isimli fonksiyonu yükle (veya davet et veya import et)"

Yani Python'ca söylemek gerekirse:

```
from fonk import carpma
```

Bu sayede fonk isimli modülün içinde yer alan carpma isimli fonksiyonu belgemizin içine davet ettik... Bu "carpma" isimli fonksiyon da carpma işlemi yaptığı için, girdiğimiz "34" ve "45" sayılarını birbiriyle çarptı...

Hatırlarsanız çok önceden yazdığımız bir hesap makinesi programında, bölme işlemlerini hatasız yapabilmek için

```
from __future__ import division
```

satırını eklemiştik... İşte şimdi yaptığımız şey de buna çok benziyor... İsterseniz /usr/lib/python2.4/ klasörünün içine bakın. Orada "\_\_future\_\_.py" adlı bir dosya göreceksiniz. Bu dosyayı açtığınızda onun içinde de "division" adlı bir satır olduğunu görürsünüz... İşte bu "from \_\_future\_\_ import division" komutuyla "\_\_future\_\_" dosyası içindeki "division" satırını kendi belgemizin içine davet etmiş oluyoruz... Dediğimiz gibi, bu konuyu bir sonraki bölümde daha ayrıntılı bir şekilde inceleyeceğiz.

Yukarıdaki fonksiyonu şu şekilde de çalıştırabiliriz:

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

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

```
import fonk
```

```
fonk.carpma(34,45)
```

Gördüğünüz gibi, "fonk" adlı modülün içinden belirli bir fonksiyonu programımıza davet etmek yerine, "fonk" adlı modülün kendisini de davet edebiliyoruz. Bu şekilde "fonk" modülü içinde ne kadar fonksiyon varsa programımıza davet edilecektir. Ancak dikkat ederseniz, bu şekilde

fonksiyonumuzu parçacık olarak kullanıyoruz. Yani "fonk.carpma()" şeklinde... Tıpkı bir önceki bölümün konusu olan "Listeler, Demetler ve Sözlükler"de gördüğümüz gibi... Hatırlarsanız orada da bazı parçacıklar kullanarak işimizi görüyorduk...

### Fonksiyon İçindeki Değişkenlerin Okunma Sırası

Şimdi şu örneği ele alalım:

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-
def merhaba(arkadas):
    print "Merhaba", arkadas
arkadas = "Kezban"
merhaba(arkadas)
```

Her şey normal... Çıktımız:

Merhaba Kezban

olacaktır.

Yukarıdaki örneğe ufak bir ekleme yapalım:

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-
def merhaba(arkadas):
    print "Merhaba", arkadas
arkadas = "Kezban"
arkadas = "Mücella"
merhaba(arkadas)
```

Sizce Python burada kimi arkadaştan sayacak? Kezban'ı mı yoksa Mücella'yı mı?

Cevap: Mücella!

Peki bunun nedeni nedir?

Cevap: Python fonksiyonlarda değişkenleri okumaya sondan başlar... Yukarıdaki örnekte arkadas = "Mücella" ifadesinin altına arkadas = "Ahmet" ifadesini eklerseniz, Python bu kez "Ahmet"i dikkate alacaktır... Çünkü bu kez en sonda "Ahmet" değeri olmuş olacak...

Şimdi şu örneğe bakın:

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-
def merhaba(arkadas):
    print "Merhaba", arkadas
    arkadas = "Kezban"
    print "Artık benim yeni arkadaşım", arkadas
arkadas = "Mücella"
merhaba(arkadas)
```

Bu örnekte aynı adla iki farklı değişken tanımladık. Burada Python'un değişkenleri okurken nasıl bir sıra takip ettiğine dikkat edin...

Ayrıca gördüğünüz gibi,

```
arkadas = "Kezban"
```

satırını,

```
arkadas = "Mücella"
```

satırından farklı olarak print ifadesinin tam altına denk getirdik. Çünkü kodumuz henüz bitmedi. Daha başka satırlar da eklemeyi düşünüyoruz kodumuza. Bu değişkeni satırın en başına alırsak Python beyin üstü betona çakılacaktır. Çünkü bu değişkeni satır başına alırsak, Python bizim artık son kodu yazdığımızı, bu kodun hemen ardından "merhaba(arkadas)" satırını ekleyip fonksiyonu çağırmak suretiyle mevcut fonksiyon bloğunu sona erdireceğimizi zanneder... Halbuki bizim yazacak birkaç satırımız daha var... İsterseniz bu ifadeyi satır başına alarak Python'un bize ne tür küfürler edeceğini görebilirsiniz...

Tabii ki biz bu iki değişkene farklı adlar vererek çok daha kolay biçimde işimizi halledebiliriz:

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

def merhaba(arkadas, ark):
    print "Merhaba", ark
    print "Artık benim yeni arkadaşım", arkadas
```

```
arkadas = "Kezban"
```

```
ark = "Mücella"
```

```
merhaba(arkadas, ark)
```

## "global" ifadesi

Şimdi kendimize şöyle bir soru soralım:

Acaba fonksiyon bloğu içinde tanımladığımız bir değişken, mevcut fonksiyon haricinde de aynı adla kullanılabilir mi? Yani diyelim ki bir fonksiyon oluşturduk ve bu fonksiyon içinde bir x değişkeni tanımladık. Daha sonra başka bir fonksiyon daha oluşturduk ve bir önceki fonksiyonda tanımladığımız x değişkenini bu yeni fonksiyon içinde de kullanmak istiyoruz. Amacımız mesela şöyle bir şey yapmak:

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

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

```
def matematik_toplama():
```

```
    soru = input("Lütfen bir sayı girin: ")
```

```
    soru2 = input("Lütfen başka bir sayı daha girin: ")
```

```
    print "Bu iki sayı toplanırsa şu çıkar:"
```

```
    print soru + soru2
```

```
matematik_toplama()
```

```
def matematik_carpma():
```

```
    print "bu iki sayı çarpılırsa şu çıkar:"
```

```
    print soru * soru2
```

```
matematik_carpma()
```

Bu kodları bu şekilde çalıştırırsak Python bize bir hata mesajı gösterecektir. Çünkü bir fonksiyon içinde tanımlanan değişkenler sadece o fonksiyonun sınırları içinde geçerlidir. Yani bir fonksiyon içinde tanımlanan bir değişken başka bir fonksiyon içinde kullanılamaz. Eğer biz bir fonksiyon içinde tanımladığımız bir değişkeni o fonksiyon dışında da kullanmak istiyorsak şöyle bir satır eklemeliyiz:

```
global [değişken_adı]
```

Bunu yukarıdaki kodlara uygularsak şöyle olmalı:

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-
def matematik_toplama():
    global soru, soru2
    soru = input("Lütfen bir sayı girin: ")
    soru2 = input("Lütfen başka bir sayı daha girin: ")
    print "Bu iki sayı toplanırsa şu çıkar:"
    print soru + soru2
matematik_toplama()

def matematik_carpma():
    print "bu iki sayı çarpılırsa şu çıkar:"
    print soru * soru2
matematik_carpma()
```

Gördüğünüz gibi, "global" ifadesi, değişkenin değerinin bir nevi "evrensel" olmasını sağlıyor...

Bir örnek daha verelim:

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-
def deneme(arkadas):
    print "Merhaba", arkadas
    arkadas = "Kezban"
    print "Benim yeni arkadaşım artık", arkadas
arkadas = "Mücella"
deneme(arkadas)
print "evet, onun arkadaşı artık", arkadas
```

Bu kodu bu haliyle çalıştırırsak çıktımız şöyle olacaktır:

```
Merhaba Mücella
Benim yeni arkadaşım artık Kezban
evet, onun arkadaşı artık Mücella
```



Son satırdaki mantık hatasını görüyorsunuz. Olması gereken "Mücella" değil, "Kezban"...

Bu durumu düzeltmek için global ifadesini kullanacağız:

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-
def deneme():
    global arkadas
    print "Merhaba", arkadas
    arkadas = "Kezban"
    print "Benim yeni arkadaşım artık", arkadas
arkadas = "Mücella"
deneme()
print "evet, onun arkadaşı artık", arkadas
```

Dikkat ederseniz, global ifadesini kullanabilmek için fonksiyon tanımlarındaki parantez içi parametreleri kaldırdık. Aksi halde Python bize bir hata mesajı gösterecektir.

### **"return" ifadesi**

Bu ifadenin ne işe yaradığını anlamak için şu örneklerle bakalım:

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-
def return_deneme(a,b):
    if a < b:
        return a
    else:
        return b
print return_deneme(34, 45)
```

Gördüğünüz gibi burada "return" ifadesi yarattığımız koşulun sonucuna göre ya a değişkenini ya da b değişkenini "döndürüyor"; yani bize çıktı olarak veriyor...

Bir de şu örneğe bakalım:

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

```
def deneme():  
    liste = ["Ayva", "Çiçek", "Açmış", "Yaz mı", "Gelecek"]  
    return liste  
print deneme()
```

Bu örnekte de "return" ifadesi listenin öğelerini ekrana döküyor.. Bu arada fonksiyonun sonuna bir "print" ifadesi eklediğimize dikkat edin.

Eğer fonksiyonumuzu aşağıdaki şekilde yazarsak, Python bize "none" diye bir çıktı verecektir:

```
#!/usr/bin/env python  
#-*- coding:utf-8 -*-  
def deneme():  
    return  
print deneme()
```

Bu "none" değeri, fonksiyonun geçerli ve doğru olduğunu, ama hiçbir şey içermediğini anlatır...

"Return" ifadesinin ikinci bir işlevi ise bir fonksiyonun işletilmesine engel olmaktır. Önceden verdiğimiz şu örneğe bir bakalım tekrar:

```
#!/usr/bin/env python  
#-*- coding:utf-8 -*-  
def matematik_toplama():  
    global soru, soru2  
    soru = input("Lütfen bir sayı girin: ")  
    soru2 = input("Lütfen başka bir sayı daha girin: ")  
    print "Bu iki sayı toplanırsa şu çıkar:"  
    print soru + soru2  
matematik_toplama()
```

```
def matematik_carpma():  
    return  
    print "bu iki sayı çarpılırsa şu çıkar:"  
    print soru * soru2  
matematik_carpma()
```

Gördüğünüz gibi ikinci fonksiyonda "return" ifadesi kullanarak bu kısmın çalıştırılmasına engel olduk.

## "pass" ifadesi

Şu örneğe bir bakalım:

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

## Modüller

### Giriş

Modül, kabaca, fonksiyonları ve sabitleri (constants) içeren, istendiğinde başka programların içine davet edilebilen, .py uzantılı bir dosyadır. Modüller, yazacağımız programlara işlevsellik katmamızı sağlar.

Bu bölümde en önemli Python modüllerinden os modülünü inceleyerek, Python'da modüllerin nasıl kullanıldığını anlamaya çalışacağız.

### Modül Çekme (importing Modules)

Python'da programımız içinde kullanacağımız modülleri birkaç farklı yöntemle çekebiliriz. Hemen kısaca bu yöntemleri görelim:

## import modül\_adı

Bu yöntemle bir modülü, bütün içeriğiyle birlikte çekebiliriz. Veya başka bir deyişle bir modülün içinde ne var ne yoksa programımız içine davet edebiliriz... Kimileri buna "import" etmek de diyor...

## from modül\_adı import \*

Bu yöntemle bir modül içinde adı "\_\_" ile başlayanlar hariç bütün fonksiyonları programımız içine çekebiliriz. Yani bu yöntem de tıpkı yukarıda anlatılan yöntemde olduğu gibi, bütün fonksiyonları alacaktır... Yalnız "\_\_" ile başlayan fonksiyonlar hariç...

## from modül\_adı import falanca, filanca

Bu yöntem ise bir modülden "falanca" ve "filanca" adlı fonksiyonları çağdırmamızı sağlayacaktır. Yani bütün içeriği değil, bizim istediğimiz fonksiyonları çekmekle yetinecektir.

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 öbür yöntemler modül içeriğinin tamamını çekmediği için programınızda işlev kaybı yaşayabilirsiniz... Ama tabii ki hangi içeriği çekmeniz gerektiğinden eminseniz o başka...

Modüller hakkında genel bir bilgi edindiğimize göre artık önemli modüllerden os modülünü incelemeye başlayabiliriz:

## os Modülü

Bu modül bize, kullanılan işletim sistemiyle ilgili işlemler yapma olacağı sunuyor. Modülün kendi belgelerinde belirtildiğine göre, bu modülü kullanan programların farklı işletim sistemleri üzerinde çalışma şansı daha fazla...

Bu modülü, yukarıda anlattığımız şekilde çekeceğiz:

## import os

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 var! Şimdi biz bu fonksiyonlardan önemli olanlarını incelemeye çalışalım...

### **name fonksiyonu**

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 "Selam Linux kullanıcısı!"
```

```
if os.name == "nt":
```

```
    a = raw_input("Bill Gates'e mesajınızı yazın:")
```

```
    print "Selam Windows Kullanıcısı!"
```

Bu basit örnekte öncelikle "os" adlı modülü bütün içeriğiyle birlikte programımıza davet ettik... Daha sonra bu modül içindeki "name" fonksiyonunu kullanarak, 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... Python'da işletim sistemi isimleri için tanımlı 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 "Selam Linux kullanıcısı!"
if name == "nt":
    a = raw_input("Bill Gates'e mesajınızı yazın:")
    print "Selam Windows Kullanıcısı!"
```

Dikkat ederseniz burada "from os import name" komutuyla, os modülü içindeki name fonksiyonunu çektik yalnızca. Ayrıca program içinde kullandığımız "os.name" ifadesini de "name" şekline dönüştürdük... Çünkü "from os import name" komutuyla yalnızca "name" fonksiyonunu çektik, aslında os modülünü çekmediğimiz için, "os.name" yapısını kullanırsak Python bize "os" isminin tanımlanmadığını söyleyecektir.

### **listdir fonksiyonu**

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

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

Yukarıdaki örnekte her zamanki gibi, modülümüzü "import os" komutuyla programımızın içine çektik ilk önce. Ardından kullanım kolaylığı açısından "os.listdir" fonksiyonunu "a" adlı bir değişkene bağladık. 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

dizinin içeriğini liste olarak ekrana yazdırıyoruz.

Çıktının tipinden anladığımız gibi, elimizde olan şey, öğeleri yan yana dizilmiş bir "liste". Eğer biz dizin içeriğinin böyle yan yana değil de alt alta dizildiğinde daha yakışıklı görüneceğini düşünüyorsak, kodlarımızı şu biçime sokabiliriz:

```
import os
a = os.listdir("/home/")
for dosyalar in a:
    print dosyalar
```

Hatta eğer dosyalarımıza numara vererek listelemek istersek şöyle de yapabiliriz:

```
import os
a = os.listdir("/home/")
c = 0
for dosyalar in a:
    if c < len(a):
        c = c+1
        print c, dosyalar
```

### **getcwd fonksiyonu**

Os modülü içinde yer alan bu fonksiyon bize o anda hangi dizin içinde bulunduğumuza dair bilgi verir. İsterseniz bu fonksiyonun tam olarak ne işe yaradığını bir örnek üzerinde görelim. Bunun için, kolaylık açısından, hemen Python komut satırını açalım ve ">>>" işaretinden hemen sonra şu komutu yazalım:

```
import os
```

Bu komutu yazıp enter'e 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 çektik. Daha sonra mevcut\_dizin adında bir değişken yaratıp "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" ifadesinden sonra "for" döngüsünü kullandık... Bu işlemi, ekran çıktısı daha yakışıklı olsun diye yaptık... Eğer böyle bir kaygımız olmasaydı,

```
if mevcut_dizin == "/home/istihza/Desktop":
```

satırının hemen altına

```
print mevcut_dizin
```

yazıp işi bitirirdik...

Biz burada getcwd fonksiyonu için basit örnekler verdik, ama eminim siz yaratıcılığınızla çok daha farklı ve kullanışlı kodlar yazabilirsiniz...

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() fonksiyonları**

Bu iki fonksiyon yardımıyla dizin veya dizinler oluşturacağız. Mesela:

```
import os
os.mkdir("/test")
```

Bu kod "/" dizini altında "test" adlı boş bir klasör oluşturacaktır...

Eğer bu kodu şu şekilde yazarsak, "mevcut çalışma dizini" içinde yeni bir dizin oluşacaktır:

```
import os
os.mkdir("test")
```

Yani, mesela "mevcut çalışma dizini" masaüstü ise bu "test" adlı dizin masaüstünde oluşacaktır...

İsterseniz bu kodları şu şekilde 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:

```
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 "root" altında oluşacaktır...

## **rmdir() ve removedirs() fonksiyonları**

Bu fonksiyonlar bize mevcut dizinleri silme olanağı tanıyor.. Yalnız, burada hemen bir uyarı yapalım: Bu fonksiyonları çok dikkatli kullanmamız gerekiyor... Ne yaptığınızdan, neyi sildiğinizden emin değilseniz bu fonksiyonları kullanmayın! Çünkü Python bu komutu verdiğinizde tek bir soru bile sormadan çatır çatır silecektir belirttiğiniz dizini... Böyle bir durumda, bir çift yaşlı gözle ekrana bakakalmak istemezsiniz, değil mi? Gerçi, bu komutlar yalnızca içi boş dizinleri silecektir, ama yine de benden uyarması!

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 uçtu gitti... Haydi selamete!

Bu işlemin ardından hâlâ kendinizdeyseniz bir de şu örneğe bakın:

```
import os  
os.rmdir("/home/istihza/TEST")
```

Bu kod ise /home/kullanıcı\_adı dizini altındaki boş "TEST" dizinini uçuracaktır...

Tıpkı makedirs() 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 programcılar 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 bir muamele yaptı bize... Ayrıca kullandığımız "os.makedirs" fonksiyonunu da gördüğü için ne yapmaya çalıştığımızı anlayamadı ve kafası karıştı... Yani, "Arkadaşım, bir yandan dizin oluşturmaya çalışıyorsun, bir yandan da bölme yapmaya çalışıyorsun... Karar ver!" dedi bize Python... Peki bu meseleyi nasıl halledeceğiz?

Bu meseleyi halletmek için kullanmamız gereken başka bir fonksiyon var Python'da...

### **os.sep fonksiyonu**

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 fonksiyonu ne işe yarar? Yazdığımız kodlarda doğrudan "dizin ayracı" vermek yerine bu fonksiyonu 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 ifadesi, ilgili işletim sistemi hangisiyle ona ait olan dizin ayracının otomatik olarak yerleştirilmesini sağlayacaktır...

Bu os.sep fonksiyonu 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 fonksiyonundan 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 fonksiyonuna bir de "join" diye bir ifade ekledik. Bu ifade kendisinden sonra gelecek dizin adlarının "birleştirilmesi" emrini veriyor... Bu fonksiyon sayesinde "/" işaretine hiç 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 fonksiyonları yardımıyla, bir önceki satırda oluşturduğumuz "test1" adlı dizinin içinde "test2" adlı bir dizin daha oluşturuyoruz. Burada os.sep.join fonksiyonu "/" işaretiyle uğraşmadan dizinleri birleştirme imkanı sağlıyor bize... Hemen alttaki satırda da benzer bir işlem yapıp

kodlarımızı bitiriyoruz.

Böylece bir fonksiyon tanımlamış olduk. Şimdi bu dosyayı "deneme" 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'e 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 fonksiyonunun çıktısı birbirinden farklı oluyor. Bu örnek, os.sep fonksiyonunun, yazdığımız programların "taşınabilirliği" (portability), yani "farklı işletim sistemleri üzerinde çalışabilme kabiliyeti" açısından ne kadar önemli olabileceğini gösteriyor...

## **Dosya İşlemleri**

Bu bölümde Python programlama dilini kullanarak dosyaları nasıl yöneteceğimizi, yani nasıl yeni bir dosya oluşturacağımızı, bu dosyaya nasıl bir şeyler yazabileceğimizi ve buna benzer işlemleri öğreneceğiz. İsterseniz lafı hiç uzatmadan konumuza geçelim...

## Yeni bir dosya yaratma

Şimdi "mevcut çalışma dizini"nde yeni bir dosya yaratacağız. Öncelikle mevcut çalışma dizinimizin ne olduğunu görelim. Hemen Python komut satırını açıyoruz ve şu komutu veriyoruz:

```
os.getcwd()
```

Şimdi acaba kaç kişi bu tuzağa düştü merak ediyorum? Eğer komut satırında doğrudan bu komutu verdiyseniz hata mesajını da görmüşsünüzdür... Çünkü biliyorsunuz önce os modülünü çekmemiz gerekiyor...

```
import os
```

```
os.getcwd()
```

Şimdi oldu... Biraz sonra oluşturacağımız dosya bu komutun çıktısı olarak görünen dizin içinde oluşacaktır. Sayın ki bu dizin Masaüstü olsun...

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 çekilmesine gerek yok. Biraz önce os modülünü çekmemizin nedeni yalnızca "os.getcwd()" fonksiyonunu kullanmaktı... Bu noktayı da belirttikten sonra komutumuzu veriyoruz:

```
open("viddansiz_sabuha.txt","w")
```

Böylelikle masaüstünde "viddansiz\_sabuha" 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 yaratmaya yaradı... Parantez içindeki "viddansiz\_sabuha.txt"nin de ne olduğu açık.. Yaratacağımız dosyanın adını tırnak içine almayı unutmuyoruz. Peki bunların hepsini anladık da bu "w" ne oluyor?

Python'da dosyaları yönetirken, dosya izinlerini de belirtmemiz gerekir. Yani mesela bir dosyaya yazma yetkisi vermek için "w" kipini (mode) kullanıyoruz. Bu harf İngilizce'de "yazma" anlamına gelen "write" kelimesinin kısaltması oluyor. Bunun dışında bir de "r" kipi ve "a" kipi bulunur. "r", İngilizce'de "okuma" anlamına gelen "read" kelimesinin kısaltması... "r" kipi oluşturulan veya açılan bir dosyaya yalnızca "okuma" izni verildiğini gösterir. Yani bu dosya üzerinde herhangi bir değişiklik yapılamaz. Değişiklik yapabilmek için biraz önce gösterdiğimiz "w" kipini kullanmak gerekir. Bir de "a" kipi vardır, dedik. "a" da İngilizce'de "eklemek" anlamına gelen "append" kelimesinden geliyor... "a" kipi önceden oluşturduğumuz bir dosyaya yeni veri eklemek için kullanılır. Bu şu anlama geliyor. Örneğin "viddansiz\_sabuha.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... 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("vicdansiz_sabuha.txt", "w")
```

komutuyla "vicdansiz\_sabuha.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("vicdansiz_sabuha.txt", "w")
```

Bu arada dikkatli olun, eğer bilgisayarınızda önceden "vicdansiz\_sabuha.txt" adlı bir dosya varsa, yukarıdaki komut size hiç bir uyarı vermeden eski dosyayı silip üzerine yazacaktır...

Şimdi başka bir örnek verelim:

```
ilkdosyam=open("vicdansiz_sabuha.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 imkanımız olmaz. Ayrıca bu kip yardımıyla yeni bir dosya da oluşturamayız. Bu kip bize **varolan** bir dosyayı açma imkanı 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 hata mesajı gösterecektir.

Eğer **varolan** bir dosyayı açıp içine yeni veriler yazmak istersek şu kipi kullanmamız gerekir:

```
dosya = open("vicdansiz_sabuha.txt", "a")
```

Ayrıca "a" kipi "r" kipinin aksine bize yeni dosya oluşturma imkanı da verir.

Eğer yazdığınız kod içinde yukarıdaki üç kipten hiçbirini kullanmazsak, Python öntanımlı olarak "r" kipini kullanacaktır. Tabii "r" kipinin yukarıda bahsettiğimiz özelliğinden dolayı, bilgisayarımızda yeni bir dosya oluşturmak istiyorsak, kip belirtmemiz, yani "w" veya "a" kiplerinden birini kullanmamız gerekir...

Bu arada, yukarıdaki örneklerde biz dosyamızı "mevcut çalışma dizini" içinde oluşturduk. Tabii ki siz isterseniz tam yolu belirterek, dosyanızı istediğiniz yerde oluşturabilirsiniz. Mesela:

```
dosya = open ("/home/kullanıcı_adı/deneme.txt", "w")
```

komutu "/home/kullanıcı\_adı/" dizini altında, "yazma yetkisi verilmiş", "deneme.txt" adlı bir dosya oluşturacaktır.

Ayrıca sadece ".txt" uzantılı dosyalar değil, pek çok farklı dosya tipi de oluşturabilirsiniz. Mesela ".odt" uzantılı bir dosya oluşturarak dosyanın OpenOffice ile açılmasını sağlayabilirsiniz. Ya da ".html" uzantılı bir dosya oluşturarak internet tarayıcınızla açılacak bir dosya yaratabilirsiniz.

## Dosyaya Veri İşleme

Şimdi bilgisayarımızda halihazırda varolan veya bizim sonradan yarattığımız bir dosyaya nasıl verişi girişi yapabileceğimizi göreceğiz. Mesela "deneme.odt" adlı bir dosya oluşturarak içine "Batsın bu dünya!" yazalım... Ama bu kez komut satırında değil de metin üzerinde yapalım bu işlemi. Hemen boş bir kwrite belgesi açıp içine şunları yazıyoruz:

```
#!/usr/bin/env/ python
#-*- coding: iso-8859-9
dosya = open("deneme.odt", "w")
dosya.write("Batsın bu dünya!...")
dosya.close()
```

İlk iki satırın ne olduğunu zaten bildiğimiz için geçiyoruz... Aynen biraz önce gördüğümüz şekilde "dosya" adlı bir değişken yaratıp bu değişkenin değeri olarak "open("deneme.odt", "w")" satırını belirledik. Böylelikle "deneme.odt" adında, "yazma yetkisi verilmiş" bir dosya oluşturduk. Daha sonra "dosya.write" fonksiyonu yardımıyla "deneme.odt" dosyasının içine "Batsın bu dünya!..." yazdık... En son da "dosya.close()" emrini vererek 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 bir rivayete göre 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 "Batsın bu dünya!..." yazdığımız dosyamıza ikinci bir satır ekleyelim..

```
#!/usr/bin/env python
#-*- coding: iso-8859-9
dosya = open("deneme.odt","a")
dosya.write("\nBitsin bu rüya...")
dosya.close()
```

Gördüğünüz gibi bu kez dosyamızı "a" kipiyle açtık... Zaten "w" kipiyle açarsak eski dosyayı uçurmuş oluruz... O yüzden Python'la programlama yaparken bu tip şeylere çok dikkat etmek gerekir.

Dosyamızı "a" kipiyle açtıktan sonra "dosya.write" fonksiyonu yardımıyla "Bitsin bu rüya..." satırını eski dosyaya ekledik. Ama burada dikkat ederseniz, "\n" işaretini kullandık... Bu da daha önce bahsettiğimiz "kaçış dizileri"nden biridir; dosyaya ekleyeceğimiz ifadenin bir alt satıra yazılmasını sağlar. Eğer bunu kullanmazsak eklemek istediğimiz satır bir önceki satırın hemen arkasına getirilecektir. Bütün bunlardan sonra da "dosya.close()" fonksiyonu yardımıyla dosyamızı kapattık.

Bir de şu örneğe bakalım:

```
#!/usr/bin/env python
#-*- coding: iso-8859-9
dosya = open("şiiir.odt", "w")
dosya.write("Bütün güneşler batmadan,\nBi türkü daha söyleyeyim bu yerde\n\t\t\t\t\t--Orhan Veli--")
dosya.close()
```

Gördüğünüz gibi, "şiiir" adlı bir OpenOffice dosyası oluşturup bu dosyaya yazma yetkisi verdik. Bu dosyanın içine yazılan verilere dikkat edin. İkinci mısrayı bir alt satıra almak için "\n" ifadesini kullandık. Daha sonra "Orhan Veli" satırını sayfanın sağına doğru kaydırmak için "\t" ifadesini kullandık. Bu ifade de tıpkı "\n" gibi bir "kaçış dizisi"dir; klavyedeki "tab" tuşu gibi, cümleyi sağa kaydırır... Bizim örneğimizde "\n" ve "\t" ifadelerini yan yana kullandık. Böylece aynı cümleyi hem alt satıra almış, hem de sağa doğru kaydırmış olduk... Ayrıca birkaç tane "\t" ifadesini yan yana kullanarak cümleyi sayfanın istediğimiz noktasına getirdik...

İsterseniz yukarıdaki kodu şu şekilde kısaltabilirsiniz de:

[illegible]

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:

```
dosya = open("şiiir2.odt", "w")
dosya.writelines(["Bilmezler yalnız yaşamayanlar", "\nNasıl korku verir sessizlik insana",
"\nİnsan nasıl konuşur kendisiyle", "\nNasıl koşar aynalara bir cana hasret", "\nBilmezler..."])
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...

## Bir Dosyadan Veri Okuma

Şimdiye kadar nasıl yeni bir dosya oluşturacağımızı, bu dosyaya nasıl veri gireceğimizi ve bu dosyayı nasıl kapatacağımızı öğrendik. Şimdi de yarattığımız bir dosyadan nasıl veri okuyacağımızı öğreneceğiz. Bu iş için de **"read()"**, **"readlines()"** ve **"readline()"** fonksiyonlarından faydalanacağız.

Şu örneğe bir bakalım:

```
>>>yeni = open("Şiir.odt","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.odt")
>>>yeni.read()

'Sular  \xc3\xa7ekilmeye  ba\xc5\x9flad\xc4\xb1  k\xc3\xb6klerden...\nIs\xc4\xb1nmaz  m\xc4\xb1
```

acaba ellerimde kan?

\nAh,ne olur! B\xc3\xbc t\xbcn g\xbcne\x5\x9fler batmadan

\nBi t\xbc t\xbc daha s\xbcyleyeyim bu yerde...'

"yeni.read()" satırına kadar olan kısmı zaten biliyoruz... Burada kullandığımız "yeni.read()" fonksiyonu "yeni" adlı değişkenin içeriğini okumamızı sağlıyor. "Yeni" adlı değişkenin değeri "Şiir.odt" adlı bir dosya olduğu için, bu fonksiyon "Şiir.odt" adlı dosyanın içeriğini bize gösterecektir. Gördüğünüz gibi bu komutun çıktısında Türkçe karakterler bozuk görünüyor... Ayrıca kullandığımız "\n" ifadesi de çıktıda yer alıyor... Esasında bu komut bize Python'un yazdığımız kodları nasıl gördüğünü gösteriyor. Eğer biz daha düzgün bir çıktı elde etmek istersek en son satırdaki komutu şu şekilde vermemiz gerekir:

```
print yeni.read()
```

Ayrıca "read()" dışında bir de "readlines()" adlı bir fonksiyon bulunur. Eğer yukarıdaki komutu

```
yeni.readlines()
```

şeklinde verecek olursak, çıktının bir liste olduğunu görürüz.

Bir de, eğer bu "readlines" fonksiyonunun sonundaki "s" harfini atıp;

```
yeni.readline()
```

şeklinde bir kod yazarsak, dosya içeriğinin yalnızca ilk satırı okunacaktır. Python'un "readline()" fonksiyonunu değerlendirirken kullandığı ölçüt şudur: "Dosyanın başından itibaren ilk '\n' ifadesini gördüğün yere kadar oku".

Bunların dışında, eğer istersek bir "for" döngüsü kurarak da dosyamızı okuyabiliriz:

```
yeni = open("Şiir.odt")
```

```
for satir in yeni:
```

```
    print satir
```

Dikkat ettiyseniz,

```
print yeni.readlines()
```

veya alternatif komutlarla dosya içeriğini okurken şöyle bir şey oluyor:

Mesela içinde

Birinci satır

İkinci satır

Üçüncü satır

yazan bir dosyamız olsun.

`dosya.readline()`

komutuyla bu dosyanın ilk satırını okuyalım. Daha sonra tekrar bu komutu verdiğimizde birinci satırın değil, ikinci satırın okunduğunu görürüz. Çünkü Python ilk okumadan sonra imleci (Evet, biz görmesek de aslında Python'un dosya içinde gezdirdiği bir imleç var!) dosyada ikinci satırın başına kaydırıyor... Eğer bir daha verirsek bu komutu, üçüncü satır okunacaktır. Son bir kez daha bu komutu verirsek, artık dosyanın sonuna ulaşıldığı için, ekrana hiç bir şey yazılmayacaktır. Böyle bir durumda dosyayı başa sarmak için şu fonksiyonu kullanıyoruz (Dosyamızın adının "dosya" olduğunu varsayıyoruz):

`dosya.seek(0)`

Böylece imleci tekrar dosyanın en başına almış olduk.

Tabii siz isterseniz, bu imleci farklı noktalara da taşıyabilirsiniz. Mesela:

`dosya.seek(10)`

komutu imleci 10. karakterin başına getirecektir (Saymaya her zamanki gibi 0'dan başlıyoruz...)

Bu "seek()" fonksiyonu aslında iki adet parametre alabiliyor. Şöyle ki:

`dosya.seek(5,0)`

komutu imleci dosyanın başından itibaren 5. karakterin bulunduğu noktaya getirir. Burada "5" sayısı imlecin kaydırılacağı noktayı, "0" sayısı ise bu işlemin dosyanın başından sonuna doğru olacağını, yani saymaya dosyanın başından başlanacağını gösteriyor...

`dosya.seek(5,1)`

komutu imlecin o anda bulunduğu konumdan itibaren 5. karakterin olduğu yere ilerlemesini sağlar. Burada "5" sayısı yine imlecin kaydırılacağı noktayı, "1" sayısı ise imlecin o anda bulunduğu konumun ölçüt alınacağını gösteriyor.

Son olarak;

`dosya.seek(-5,2)`

komutu ise saymaya tersten başlanacağını, yani dosyanın başından sonuna doğru değil de sonundan başına doğru ilerlenerek, imlecin sondan 5. karakterin olduğu yere getirileceğini gösterir.

Bu ifadeler biraz karışık gelmiş olabilir. Bu konuyu anlamanın en iyi yolu bol bol uygulama yapmak ve deneyerek görmektir...

İsterseniz, yukarıdaki okuma fonksiyonlarına da belirli parametreler vererek dosya içinde okunacak satırları veya karakterleri belirleyebilirsiniz. Mesela:

`yeni.readlines(3)`

komutu dosya içinde, imlecin o anda bulunduğu noktadan itibaren 3. karakterden sonrasını okuyacaktır.

Peki o anda imlecin hangi noktada olduğunu nereden bileceğiz? Python'da bu işlem için de bir fonksiyon bulunur:

`dosya.tell()`

komutu yardımıyla imlecin o anda hangi noktada bulunduğunu görebilirsiniz. Hatta dosyayı bir kez

`dosya.read()`

komutuyla tamamen okuttuktan sonra

`dosya.tell()`

komutunu verirsiniz imleç mevcut dosyanın en sonuna geleceği için, ekranda gördüğünüz sayı aynı zamanda mevcut dosyadaki karakter sayısına eşit olacaktır...

## **Dosyaya Değişken Yazdırma**

Python'da dosya işlemleri yaparken bilmemiz gereken en önemli noktalardan biri de şudur: "Python ancak "karakter dizileri"ni (strings) dosyaya yazdırabilir. Sayıları yazdıramaz. Eğer biz sayıları da yazdırmak istiyorsak önce bu sayıları "karakter dizisi"ne çevirmemiz gerekir. Bir örnek verelim:

```
>>>x = 50
```

```
>>>dosya = open("deneme.txt","w")
```

```
>>>dosya.write(x)
```

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 dizisi"ne çevirdik. Tabii ki bu işlemin tersi de mümkün. Eğer "x" bir karakter dizisi olsaydı, şu komutla onu sayıya çevirebilirdik:

```
int(x)
```

## Dosya Silme

Peki yarattığımız bu dosyaları nasıl sileceğiz? Python'da herhangi bir şekilde oluşturduğumuz bir dosyayı silmenin en kestirme yolu şudur:

```
import os
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")
```

## Dosyanın Herhangi Bir Yerine Satır Ekleme

Şimdiye kadar hep dosya sonuna satır ekledik. Peki ya bir dosyanın ortasına bir yere satır eklemek istersek ne yapacağız?

Şimdi:



Diyelim ki elimizde "deneme.txt" adlı bir dosya var ve içinde şunlar yazılı:

Birinci Satır

İkinci Satır

Üçüncü Satır

Dördüncü Satır

Beşinci Satır

Biz burada "İkinci Satır" ile "Üçüncü Satır" arasına "Merhaba Python!" yazmak istiyoruz. Önce bu "deneme.txt" adlı dosyayı açalım:

```
>>>kaynak = open("deneme.txt")
```

Bu dosyayı "okuma" kipinde açtık, çünkü bu dosyaya herhangi bir yazma işlemi yapmayacağız. Yapacağımız şey, bu dosyadan veri okuyup başka bir hedef dosyaya yazmak olacak... O yüzden hemen bu hedef dosyamızı oluşturalım:

```
>>>hedef = open("test.txt","w")
```

Bu dosyayı ise "yazma" modunda açtık... Çünkü kaynak dosyadan okuduğumuz verileri buraya yazdıracağız.

Şimdi de, yapacağımız "okuma işlemi"ni tanımlayalım:

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

Böylece "kaynak" dosya üzerinde yapacağımız satır okuma işlemi de tanımlamış olduk...

Şimdi kaynak dosyadaki "birinci satır" ve "ikinci satır" verilerini alıp hedef dosyaya yazdırıyoruz. Bu iş için bir "for" döngüsü oluşturacağız:

```
for satirlar in oku[:2]:  
    hedef.write(satirlar)
```

Burada biraz önce oluşturduğumuz "okuma işlemi" değişkeni yardımıyla "0" ve "1" no'lu satırları alıp hedef adlı dosyaya yazdırdık...

Şimdi eklemek istediğimiz satır olan "Merhaba Python!" satırını ekleyeceğiz:

```
hedef.write("Merhaba Python!\n")
```

Sıra geldi kaynak dosyada kalan satırları hedef dosyasına eklemeye...

```
for satirlar in oku[2:]:
```

```
hedef.write(satirlar)
```

Artık işimiz bittiğine göre hedef ve kaynak dosyaları kapatalım:

```
>>>kaynak.close()
```

```
>>>hedef.close()
```

Bu noktadan sonra eğer istersek kaynak dosyayı silip adını da hedef dosyanın adıyla değiştirebiliriz:

```
>>> os.remove("deneme.txt")
```

```
>>> os.rename("test.txt","deneme.txt")
```

Tabii bu son işlemleri yapmadan önce os modülünü çağırmayı unutmuyoruz...

## Hatalarla Başetme

### Giriş

Programcılar bir kod yazarken, yazılan kodları işletecek kullanıcıları her zaman göz önünde bulundurmalı, program çalıştırılırken kullanıcıların ne gibi hatalar yapabileceklerini kestirmeye çalışmalıdır. Çünkü kullanıcılar her zaman programcının istediği gibi davranmayabilir. Bu sözleri basit bir örnekle açıklayalım. Diyelim ki bir kod yazdık ve kullanıcıdan bir sayı girmesini istiyoruz. Eğer kullanıcı gerçekten bir sayı girerse sorun yok, ancak tabii ki kullanıcıların her zaman uslu uslu sayı girmesini bekleyemeyiz ve beklememeliyiz. Çünkü siz her ne kadar açık açık sayı girilmesini istesiniz de kullanıcı bilerek veya bilmeyerek sayı yerine başka değerler de girebilir. Hatta hiç bir giriş yapmadan "enter" tuşuna bile basabilir. Yazdığımız bu 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.

## Değişken İsmine İlişkin Hatalar (NameError)

Şu örneğe bir bakalım:

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-
while True:
    soru1 = input("Lütfen toplama işlemi için bir sayı giriniz: ")
    soru2 = input("Lütfen toplama işlemi için ikinci sayıyı giriniz: ")
    print soru1 + soru2
```

Eğer iyi huylu bir kullanıcıya denk gelirsek ne âlâ... Sevgili kullanıcımız uslu uslu iki adet sayı girecek, programımız da mutlu mesut bir şekilde bu iki sayıyı toplayıp kibarca kullanıcıya bildirecektir... Ama ne yazık ki işler her zaman böyle yürümez... Kullanıcımız programı çalıştırdıktan sonra bir sayı yerine bir harf girmeyi de tercih edebilir. Böyle bir durumda ise kullanıcı şu hatayı alır. (Diyelim ki kullanıcı "e" harfine basmış olsun):

Traceback (most recent call last):

File "deneme.py", line 4, in ?

```
soru1 = input("Lütfen toplama işlemi için bir sayı giriniz: ")
```

File "<string>", line 0, in ?

NameError: name 'e' is not defined

İşte bu noktadan sonra kullanıcı, program yazarının kulaklarını çınlatmaya başlayacak, "Ne biçim program bu. Hemen çöküyor," diye sızlanacaktır. Gerçi aldığı hata mesajı sorunun nerede olduğunu söylüyor, ama tabii ki anlayana... Sizin bu noktada iyi bir programcı olarak yapmanız gereken şey, kullanıcının hareketlerini önceden kestirip, onun alacağı hata mesajlarını anlaşılır hale getirmek olacaktır.

Şimdi ortaya çıkan hata mesajına bir bakalım. Bu mesajda önemli kısım "NameError: name 'e' is not defined" yazan yer... Demek ki Python, kullanıcının girdiği "e" harfini değişken olarak algılamış, tabii ki ortada tanımlanmış bir "e" değişkeni olmadığı için de böyle bir hata mesajı vermiş... Bu sorunu şu şekilde giderebiliriz:

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

```
try:
    soru1 = input("Lütfen toplama işlemi için bir sayı giriniz: ")
    soru2 = input("Lütfen toplama işlemi için ikinci sayıyı giriniz: ")
    print soru1 + soru2
except NameError:
    print "Sayı dedik sana! Harf değil! Tekrar dene..."
```

Artık kullanıcı sayı yerine harf girdiğinde, programımız çökmeyecek, sayı girmesi konusunda kullanıcıyı nazikçe uyararak çalışmaya devam edecektir.

İsterseniz burada yaptığımız şeyi biraz açıklayalım.. Aslında yaptığımız şey, yazdığımız ilk kodları bir **"try... except..."** bloğu içine almaktan ibaret. Yazdığımız bu kod ile Python'a kendi anlayacağı dilden şöyle demiş olduk:

"Eğer programın çalıştırılması sırasında değişken ismine ilişkin bir hatayla karşılaşırsan bu hatayı sineye çek ve ekrana, 'Sayı dedik sana!...' cümlesini yazdırıp yoluna devam et..."

Biz yukarıdaki kodla kullanıcıların yapabileceği bir hata türüyle başetmiş olduk, ama emin olun kullanıcılar çok daha başka, çok daha karmaşık hatalar da yapabilirler...

### Sözdizimine İlişkin Hatalar (SyntaxError)

Dediğimiz gibi, kullanıcıların yapabileceği hataların sınırı, hududu yok... Mesela yukarıdaki kodu çalıştıran bir başka kullanıcı sadece sayı girmek yerine, önce bir sayı girip, "enter"e basmadan bir tane de harf girmeyi uygun görebilir... Yani "3g" gibi bir şey yazabilir... O zaman da şöyle bir hata alır:

Traceback (most recent call last):

File "deneme.py", line 5, in ?

soru1 = input("Lütfen toplama işlemi için bir sayı giriniz: ")

File "<string>", line 1

3g

^

SyntaxError: unexpected EOF while parsing

Burada da önemli kısım, "SyntaxError: unexpected EOF while parsing"... Buradan anladığımıza göre, Python bir "sözdizimi hatası" vermiş... Bu hatayı da kodumuza şu şekilde ekleyebiliriz:

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-
while True:
    try:
        soru1 = input("Lütfen toplama işlemi için bir sayı giriniz: ")
        soru2 = input("Lütfen toplama işlemi için ikinci sayıyı giriniz: ")
        print soru1 + soru2
    except NameError:
        print "Sayı dedik sana! Harf değil! Tekrar dene..."
    except SyntaxError:
        print "Yazım hatası yaptınız! Lütfen bir daha deneyin..."
```

Böylelikle kullanıcıdan kaynaklanabilecek iki hata türünü öngörüp her biri için ayrı uyarı verebiliyoruz.

### **Hata Kodu Vermeden Hata Yakalama**

Tabii ki bir kullanıcının yol acabileceği bütün hataları kestirmek mümkün değildir. O yüzden, eğer hataya yönelik özel bir mesaj göstermek gibi bir kaygımız yoksa olası bütün hatalar için şu kalıbı kullanabiliriz:

```
try:
    .....
except:
    .....
```

Hemen bir örnek verelim:

```
try:
    dosya = open("deneme.txt", "r")
except:
    print "dosya açılmıyor"
    print 2 + 2
```

Gördüğünüz gibi burada herhangi bir hata kodu belirtmedik. Ama olası bir hatayı yakaladığımız için programımız çökmedi ve bir sonraki kod olan "print 2+2" işlemi yapıldı. İsterseniz aynı kodu

bir de "try...except..." bloğu olmadan deneyelim:

```
dosya = open("deneme.txt","r")  
print 2 + 2
```

Burada ise, herhangi bir hata yakalama işlemi yapmadığımız için programımız çöktü ve ikinci kod olan "print 2 + 2" işletilemedi...

Yukarıda anlattığımız, "hata kodu vermeden hata yakalama işlemi" pratik ve kolay olsa bile her zaman tercih edilmeyebilir. Çünkü bu şekilde kullanıcıya yaptığı hatayla ilgili bilgi veremiyoruz. Dolayısıyla bu kodları içeren bir programı çalıştıran kullanıcı ortada bir hata olduğunu anlayacak, ama hatanın nereden kaynaklandığını bilemeyecektir. Çünkü yukarıda, "dosya açılmıyor" diye bir hata belirttik ama hata kodu yazmadığımız için bu dosyanın neden açılmadığını belirtmedik... Zira yukarıdaki kodda dosyanın açılmamasının birkaç nedeni olabilir. Ama eğer kodumuzu şöyle verirse en azından kullanıcının bazı önlemler almasını sağlayabiliriz.. Yukarıdaki kodu çalıştırdığımızda şöyle bir hata almıştık:

Traceback (most recent call last):

```
File "deneme.py", line 3, in ?  
dosya = open("deneme.txt","r")  
IOError: [Errno 2] No such file or directory: 'deneme.txt'
```

Şimdi hatayı tespit ettiğimize göre şu kodu yazabiliriz:

try:

```
dosya = open("deneme.txt","r")
```

except IOError:

```
print "'deneme.txt' adlı dosya bulunamadı. Lütfen klasörde bu adda bir dosya olduğundan emin  
olunuz."  
print 2 + 2
```

Böylelikle kullanıcıya daha açıklayıcı bir bilgi vermiş olduk. Artık kullanıcı hatanın ne olduğunu bildiği için buna karşı önlem de alabilir. Bir de, yukarıda görünen hata mesajında dikkat ederseniz [Errno 2] diye bir ifade geçiyor. Bunun ne olduğunu anlamak için şu örneği verelim:

```
dosya = open("/usr/bin/deneme.txt","w")
```

Bu kodu çalıştırdığımızda şu hatayı alırız:

Traceback (most recent call last):

File "deneme.py", line 3, in ?

dosya = open("/usr/bin/deneme.txt", "w")

IOError: [Errno 13] Permission denied: '/usr/bin/deneme.txt'

Gördüğünüz gibi burada da "IOError" adlı hata veriliyor, ama bu kez hata kodu [Errno 13].

Eğer istersek, biz her iki hata kodu için ayrı mesajlar verebiliriz kullanıcıya:

try:

```
dosya = open("/usr/bin/deneme.txt", "r")
```

except IOError, (hata kodu, hataadi):

```
if hata kodu == 2:
```

```
    print "Böyle bir dosya yok"
```

```
if hata kodu == 13:
```

```
    print "Bu dosyayı okuma yetkiniz yok"
```

Burada dikkat ederseniz (hata kodu, hataadi) adında iki adet parametre oluşturduk. Bu isimleri tabii ki siz kendinize göre de belirleyebilirsiniz. Önemli olan, parantez içinde iki ayrı parametre olması... Daha sonra da "if ifadeleri" yardımıyla her iki koşul için ekrana yazdırılacak çıktıları belirledik...

## Hatalarla Başetmede "pass" İfadesi

Bazen yazdığınız program hata verse bile siz kullanıcıya herhangi bir hata mesajı göstermek istemeyebilirsiniz. Böyle bir durumda kullanıcının sebep olduğu hata sessizce geçiştirilecek, programınız çalışmaya devam edecektir.

Bir örnek verelim:

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

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

```
liste = ["elma", "armut", "karpuz", "kavun", "erik", "üzüm", "şeftali", "muz"]
```

```
while True:
```

```
    try:
```

```
        s = raw_input("Lütfen bir meyve adı söyleyiniz: ")
```

```
        p = liste.index(s) + 1
```

```
        print s, "listemizde", p, "no'lu sırada bulunuyor"
```

except ValueError:

pass

Burada öncelikle, içeriğinde bazı meyveler olan bir liste yarattık. Ardından da kullanıcılardan, "bir meyve adı söylemelerini" istedik. Daha önceki bölümlerden hatırlayacağınız "liste.index()" fonksiyonunu kullanarak kullanıcının girdiği meyve adının listede kaçınıcı sırada olduğunu sorguladık. Bildiğiniz gibi Python'da liste öğeleri sıralanırken hep sıfırdan başlanıyor... Python'un kendi iç yapısı açısından bu durum mantıklı olabilir, ama insanlar saymaya bir'den başlamayı daha mantıklı bulacakları için biz kodumuza "+1" değerini ekleyerek Python'un listedeki öğeleri sıralamaya 0'dan değil de 1'den başlamasını sağladık. Bunun ardından da, kullanıcının girdiği değer listede kaçınıcı sırada olduğunu ekrana yazdırdık. Tabii ki kullanıcı isim girerken, listede olmayan bir öğeyi de söyleyebilir. Böyle bir durumda programımızın bu hatayı sessiz sedasız geçiştirmesi için de "pass" ifadesini kullandık. Eğer "try...except..." yapısını kullanmasaydık ne olacağını biliyorsunuz:

Traceback (most recent call last):

File "deneme.py", line 8, in ?

p = liste.index(s) + 1

ValueError: list.index(x): x not in list

Gördüğünüz gibi, kullanıcı açısından tamamen anlamsız bir kelime yığını çıkıyor ortaya... Üstelik programımız da bu noktada işlevini kaybedip çöküyor...

## Karakter Dizilerinin Metotları

### Giriş

Bu bölümde, Python'daki karakter dizilerinin (strings) sahip oldukları "metot"lardan söz edeceğiz. Metotlar; Python'da bir karakter dizisinin, bir sayının, bir listenin veya sözlüğün niteliklerini kolaylıkla değiştirmemizi veya bu veritiplerine yeni özellikler katmamızı sağlayan küçük "parçacıklar"dır. Aslında bu metotları daha önceki derslerimizde de görmüştük. Örneğin listeler konusunu işlerken sözünü ettiğimiz "parçacıklar" aslında bu bölümde bahsedeceğimiz metotlara birer örnek oluşturur. Yani örneğin, daha önce "append parçacığı" olarak bahsettiğimiz öğe, 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 aşına 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"den söz ediyoruz.

Daha önce sözünü ettiğimiz "append" metodu da dikkat ederseniz bu şablona uyuyordu. Hemen bir örnek hatırlayalım:

```
liste = ["elma", "armut", "karpuz"]
```

```
liste.append("kebab")
```

```
liste
```

```
["elma", "armut", "karpuz", "kebab"]
```

Gördüğünüz gibi, noktalı gösterme biçimini uygulayarak kullandığımız "append" metodu yardımıyla listemize yeni bir öge ekledik.

İşte bu yazımızda, 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. "str", İngilizce'de karakter dizisi anlamına gelen "string" kelimesinin kısaltması oluyor. Yeri gelmişken söyleyelim: Eğer karakter dizileri yerine listelerin metotlarını listelemek isterseniz kullanacağınız biçim şu olacaktır:

```
dir(list)
```

Sözlüklerin metodlarını 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 listemek 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 veritiplerinden 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 metodlarını 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 aşındırmak için iyi bir yöntem sayılabilir:

```
dir("herhangibirkelime")
```

Dediğim 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__', '__getattr__', '__getitem__', '__getnewargs__', '__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__str__', 'capitalize', 'center', 'count', 'decode', 'encode', 'endswith', 'expandtabs', 'find', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',
```

'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

Gördüğünüz gibi, Python'da karakter dizilerinin bir hayli metodu varmış... Eğer bu listeleme biçimi gözünüze biraz karışık göründüyse, elbette çıktıyı istediğiniz gibi biçimlendirmek sizin elinizde:

Mesela önceki bilgilerinizi de kullanıp şöyle bir şey yaparak çıktıyı biraz daha okunaklı hale getirebilirsiniz:

```
for i in dir(""):
    print i
```

Hatta kaç adet metot olduğunu da merak ediyorsanız şöyle bir yol izleyebilirsiniz. ("Elle tek tek sayarım," diyenlere teessüflerimi iletiyorum...)

```
print len(dir(""))
```

Şimdi sıra geldi bu metotları 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 GÖSTERİLMİYEN 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:

### **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 = "lütfen parolanızı giriniz"
```

```
a.capitalize()
```

Tabii, burada karakter dizimiz Türkçe harfler içerdiği için komutu print ile birlikte vermemiz daha uygun olabilir:

```
print a.capitalize()
```

Bir örnek:

```
a = ["elma", "armut", "kebab", "salata"]
```

```
for i in a:
```

```
    print i.capitalize()
```

Elma

Armut

Kebab

Salata

### **upper metodu**

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

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

ENFLASYON

Yalnız bu metodun Türkçe karakterlerle ufak bir sorunu var. Yani bu metod, "şçöğü" gibi Türkçe karakterleri doğrudan büyütemiyor:

```
"şeker".upper()
```

```
'\xc5\x9fEKER'
```

Burada şu komut da işe yaramayacaktır:

```
print "şeker".upper()
```

şEKER

Bu metodun Türkçe karakterleri de büyütebilmesi için ufak bir işlem yapmamız gerekiyor.

Öncelikle Python'un "locale" adlı modülünü içe aktarıyoruz (import). Ondan sonra da dil ayarlarında küçük bir düzenlemeye gidiyoruz:

```
import locale
```

```
locale.setlocale(locale.LC_ALL, "")
```

Ardından şu komutu veriyoruz:

```
print u"şeker".upper()
```

ŞEKER

Bu şekilde Türkçe karakter sorunumuzu halletmiş olduk. Dikkat ederseniz, "şeker" karakter dizisinin başına "u" harfi koyduk. Bu Python'da "unicode" adlı bir veritipine işaret ediyor. "unicode" veritipini ilerleyen derslerimizde daha ayrıntılı olarak inceleyeceğiz. Şimdilik, karakter dizilerinin başına "u" harfi getirerek bunları "unicode" adlı bir veritipine dönüştürdüğümüzü bilmemiz yeterli olacaktır. Ufak bir test yapalım:

```
a = "şeker"
type(a)
<type 'str'>
```

```
b = u"şeker"
type(b)
<type 'unicode'>
```

Bu test sayesinde tip farklılığını açıkça görebiliyoruz.

Yukarıdaki örnekte de görüldüğü gibi, bazı durumlarda Türkçe karakterleri görüntüleyebilmek için, karakter dizilerini "unicode"ye dönüştürmemiz gerekiyor.

Birkaç örnek daha yapalım:

```
kelime = "usturupsuz"
kelime.upper()
USTURUPSUZ
```

```
kelime = u"türkçe sözcükler" #kelime değişkenini "unicode" olarak tanımladığımıza dikkat edin.
kelime.upper()
TÜRKÇE SÖZCÜKLER
```

Burada şöyle bir hatırlatma yapalım: Eğer komutlarımızı Python komut satırından (etkileşimli kabuk) veriyorsak, Türkçe karakterleri düzgün görüntüleyebilmek için her oturumda sadece bir kez şu komutu vermemiz gerekiyor:

```
import locale
locale.setlocale(locale.LC_ALL,"")
```

Dediğimiz gibi, bu komutu bir kez verdikten sonra, aynı oturum içinde bir daha bu komutu vermeye gerek kalmadan işlemlerimizi yapabiliriz.

## 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
```

Tıpkı upper metodunda olduğu gibi burada da Türkçe karakterleri düzgün görüntüleyebilmek için bir kereliğine şu işlemi yapmamız gerekiyor:

```
import locale
```

```
locale.setlocale(locale.LC_ALL, "")
```

Ancak şunu unutmayın, eğer o anda açık olan komut satırı oturumunda bu komutu önceden zaten bir kez vermişseniz, o oturum boyunca bir daha bu komutu vermenize gerek yok.

Şimdi karakter dizimizi "unicode" olarak tanımlıyoruz:

```
b = u"TÜRKÇE"
```

Artık metodumuzu kullanabiliriz:

```
print b.lower()
```

```
türkçe
```

## 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 imkanı sağlıyor. Yani, eğer o anda bir harf büyükse, bu metodu kullandığımızda o harf küçülüyor; eğer bu harf o anda küçükse, bu metot o harfi büyük harfe çeviriyor. Gördüğünüz gibi, bu metodun ne yaptığını, anlatarak açıklamak zor oluyor. O yüzden hemen birkaç örnek yapalım:

```
a = "kebab"
```

```
a.swapcase()
```

```
KEBAP
```

```
b = "KEBAP"
```

```
b.swapcase()
```

```
kebab
```

```
c = "KeBaP"  
c.swapcase()  
kEbAp
```

### **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 = u"bin atlı akınlarda çocuklar gibi şendik. bin yaya dönüşte çok sinirliydik!"  
print a.title()  
Bin Atlı Akınlarda Çocuklar Gibi Şendik. Bin Yaya Dönüşte Çok Sinirliydik!
```

Burada karakter dizimizin başına neden bir "u" harfi koyduğumuzu artık biliyorsunuz. Tekrar açıklamaya gerek yok. Ayrıca bu karakter dizisini yazdırmadan önce, mevcut konsol oturumu içinde daha önce en az bir kez şu komutların verilmiş olması gerektiğini de biliyoruz:

```
import locale  
locale.setlocale(locale.LC_ALL,"")
```

### **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 mesela 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'
```

### **zfill 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
```





```
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","kebab"]
for i in liste:
    if i.startswith("e"):
        i.replace("e","i")
'ılma'
'irik'
```

'iv'

'ilbisi'

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

### **endswith 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 metodu ile ise karakter dizisinin hangi harfle bittiğini denetleyeceğiz. Kullanımı startswith metoduna çok benzer:

```
a = "elma"
```

```
a.endswith("a")
```

```
True
```

```
b = "armut"
```

```
a.endswith("a")
```

```
False
```

Bu metot yardımıyla, cümle sonlarında bulunan istemediğiniz karakterleri ayıklayabilirsiniz:

```
kd1 = "ekmek elden su gölden!"
```

```
kd2 = "sakla samanı gelir zamanı!"
```

```
kd3 = "karga karga gak dedi..."
```

```
kd4 = "Vay vicdansızlar..."
```

```
for i in kd1,kd2,kd3,kd4:
```

```
    if i.endswith("!"):
```

```
        print i.replace("!", "")
```

```
ekmek elden su gölden
```

```
sakla samanı gelir zamanı
```

### **count metodu**

"count" metodu bize bir karakter dizisi içinde bir karakterden kaç adet bulunduğunu denetleme

imkanı verecek. Lafı uzatmadan bir örnek verelim:

```
besiktas = "Sinan Paşa Pasajı"  
besiktas.count("a")  
5
```

Demek ki "Sinan Paşa Pasajı" karakter dizisi içinde 5 adet "a" harfi varmış...

Şimdi bu metodun nerelerde kullanılabileceğine ilişkin bir örnek verelim. Bu örnekteki bazı noktaları henüz derslerimizde işlemedik. Dolayısıyla bazı kısımları anlayamazsanız dert etmeyin, zira ilerde bu konuların hepsine değineceğiz. Şimdilik aşağıdaki kodun anlayabildiğimiz kısmını anlamaya çalışıp, koddan elde ettiğimiz sonuca odaklanalım. Aşağıdaki çalışmayı komut satırına değil, bir metin düzenleyici yardımıyla dosyaya yazıyoruz:

```
#-*-coding:utf8-*-  
  
#Türkçe karakterlerin düzgün görüntülenebilmesi için şu iki satırı ekleyelim...  
import locale  
locale.setlocale(locale.LC_ALL,"")  
  
while True:  
    soru = raw_input("Bir karakter dizisi giriniz: ").replace(" ", "") #replace metodu,  
karakterlerdeki boşlukları silmemizi sağlıyor.  
    karakter = list(unicode(soru)) #unicode'nin değişkenlerle birlikte kullanımına dikkat edin  
    liste = [i for i in karakter if karakter.count(i) > 1] #burada "liste üreticileri"nden  
faýdalanyyoruz. Bu konuya ilerde değineceğiz.  
    ortak = set(karakter) | set(liste) #Python'daki "set" fonksiyonu bize kısıyoldan küme  
işlemleri yapma imkanı sağlıyor.  
    for i in ortak:  
        print "%s harfinden => %s tane"%(i, karakter.count(i))
```

Yukarıdaki çalışmada kullanıcıdan herhangi bir karakter dizisi girmesini istiyoruz. Kodlarımız bize, kullanıcının girdiği karakter dizisi içinde her bir harften kaç tane olduğunu söylüyor.

## 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
```

### **isdigit metodu**

Bu metot da isalpha metoduna benzer. Bunun yardımıyla bir karakter dizisinin "sayısal" olup olmadığını denetleyebiliriz. Sayılardan oluşan karakter dizilerine "sayı karakter dizileri" adı verilir.

Örneğin şu bir "sayı karakter dizisi"dir:

```
a = "12345"
```

Metodumuz yardımıyla bunu doğrulayabiliriz:

```
a.isdigit()
```

```
True
```

Ama şu karakter dizisi sayısal değildir:

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

Hemen kontrol edelim:

```
b.isdigit()
```

```
False
```

### **isalnum metodu**

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

Daha önce bahsettiğimiz metotlardan hatırlayacaksınız:

"Alfabetik" karakter dizileri, alfabe harflerinden oluşan karakter dizileridir.

"Sayısal" karakter dizileri, sayılardan oluşan karakter dizileridir.

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

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

İsterseniz hemen bu yeni metodumuz yardımıyla bunu doğrulayalım:

```
a.isalnum()
```

```
True
```

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

```
b = "123abc>"
```

```
b.isalnum()
```

```
False
```

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

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

### **islower metodu**

Bu metot, bize bir karakter dizisinin tamamının küçük harflerden oluşup oluşmadığını denetleme imkanı 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 = "İstanbul"
```

```
a.lower()
```

```
False
```

```
b = "ADANA"
```

```
b.lower()
```

```
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; aksihalde "False" çıktısı veriyor.

Bir de şuna bakalım:

```
c = "Gün Bugündür"  
c.istitle()  
False
```

Sizce neden böyle oldu? Evet, tahmin ettiğiniz gibi, karakter dizisinin içinde Türkçe'ye özgü harfler olduğu için metodumuz yanlış sonuç verdi. Şimdi bunu düzeltmeyi öğreneceğiz. Ama önce isterseniz Türkçe karakterlerin bazen nasıl sonuçlar doğurabileceğine bir örnek verelim:

```
d = "gün bugündür"  
print d.title()  
GüN BugüNdüR
```

Gördüğünüz gibi Türkçe karakterler, "title" metodunun tamamen sapıtmasına, kısa devre yapmasına yol açtı!... Normalde title metodunun ne yapması gerektiğini biliyoruz:

```
e = "armut bir meyvedir"  
e.title()  
'Armut Bir Meyvedir'
```

Yukarıda gördüğümüz, Türkçe karakterler barındıran örnekte ise title metodu karakter dizisi içindeki kimi harfleri büyüttü, kimi harfleri ise küçük bıraktı!...

Bu durumu nasıl düzeltereğimizi biliyorsunuz. Her oturumda en az bir kez şu komutları çalıştırmamız gerekiyor:

```
import locale  
locale.setlocale(locale.LC_ALL,"")
```

Bu komutları çalıştırdıktan sonra yukarıda gösterdiğimiz ve hata verdiğini gördüğümüz komutları tekrar çalıştırmayı deneyin. Bu defa metotların sorunsuz işlediğini göreceksiniz. Tabii ki Türkçe harfler içeren karakter dizilerimizi "unicode" olarak tanımlamayı unutmuyoruz.

## **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",başlangın_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'ıncı" konumdaymış.

İstersek şöyle bir arama yöntemi de kullanabiliriz:

```
a.find("a",1,3)
```

Bu arama yöntemi şu sonucu verecektir:

```
2
```

Bu yöntemle, "a" harfini, karakter dizisinin 1 ve 3. konumlarında arıyoruz. Bu biçimin işleyişi, daha önceki derslerimizde gördüğümüz dilimleme işlemine benzer:

```
a[1:3]
```

```
"da"
```

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")
```

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 metotlarından 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:utf8-\*-**

Bu satırı açıklamaya gerek yok. Ne olduğunu biliyoruz.

**import string**

Python modülleri arasından "string" modülünü içe aktarıyoruz (import)

**metin = "tbyksr çsn jücho elu gloglu"**

Şifreli metnimizi bir değişkene atayarak sabitliyoruz.

**kaynak= "defghijklmnoprstuvyabc"**

**hedef = "abcdefghijklmnoprstuvyz"**

Burada "kaynak", şifreli metnin yapısını; "hedef" ise alfabenin normal yapısını temsil ediyor. "kaynak" adlı değişkende "d" harfinden başlamamızın nedeni yukarıda keşfettiğimiz harfler-arası ilişkidir. Dikkat ederseniz, "hedef"teki harfleri, "kaynak"taki harflere göre, her bir harf dört sıra geride kalacak şekilde yazdık. (d -> a, e ->b, gibi...) Dikkat edeceğimiz bir başka nokta ise bunları yazarken Türkçe karakter kullanmamamız gerektiğidir.

**cevir = string.maketrans(kaynak,hedef)**

Burada ise, yukarıda tanımladığımız harf kümeleri arasında bir çevrim işlemi başlatabilmek için

"string" modülünün "maketrans" adlı fonksiyonundan yararlanıyoruz. Bu komut, parantez içinde gösterdiğimiz kaynak değişkenini hedef değişkenine çeviriyor; aslında bu iki harf kümesi arasında bir ilişki kuruyor. Bu işlem sonucunda kaynak ve hedef değişkenleri arasındaki ilişkiyi gösteren bir formül elde etmiş olacağız.

```
soncevir = metin.translate(cevir)
```

Bu komut yardımıyla, yukarıda "cevir" olarak belirlediğimiz formülü, "metin" adlı karakter dizisine uyguluyoruz.

```
print soncevir
```

Bu komutla da son darbeyi vuruyoruz.

Şimdi bu komutlara topluca bir bakalım:

```
#-*-coding:utf8-*-  
import string  
metin = "tbyksr çsn jücho elu gloglu"  
kaynak= "defghijklmnoprstuvyabc"  
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 = "Adana'nın yolları taştan\nSen çıkardın beni baştan"  
print satir.splitlines()  
["Adana'nın yolları taştan", 'Sen çıkardın beni baştan']
```

### **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 yanısıra 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 ögesi "Debian", öteki ögesi de "Pardus, Ubuntu, SuSe" oldu. Bunu şu şekilde daha açık görebiliriz:

```
c[0]
'Debian'
```

```
c[1]
'Pardus, Ubuntu, SuSe'
```

Gördüğünüz gibi listenin 0. ögesi Debian'ken; listenin 1. ögesi "Pardus, Ubuntu, Suse" üçlüsü. Yani bu üçlü tek bir karakter dizisi şeklinde tanımlanmış.

Yukarıda tanımladığımız "yeni" adlı listeye "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 Metotlarını bitirmiş olduk. Dikkat ederseniz metot listesi içindeki iki metodu anlatmadık. Bunlar, *encode* ve *decode* metotları... Bunları "Python'da Unicode" konusunu işlerken anlatmak üzere şimdilik bir kenara bırakıyoruz.

Bu konuyu iyice sindirebilmek için kendi kendinize bolca örnek ve denemeler yapmanızı, bu konuyu arada sırada tekrar etmenizi öneririm.

## Nesne Tabanlı Programlama – OOP (NTP)

Bu yazımızda çok önemli bir konuyu işlemeye başlayacağız: *Python'da "Nesne Tabanlı Programlama"* (*Object Oriented Programming*). Yabancılar bu ifadeyi "OOP" olarak kısaltıyor.

Gelin isterseniz biz de bunu Türkçe'de NTP olarak kısaltalım...

Şimdilik bu "Nesne Tabanlı Programlama"nın ne olduğu ve tanımı bizi ilgilendirmiyor. Biz şimdilik

işin teorisiyle pek uğraşmayıp pratiğine bakacağız. NTP'nin pratikte nasıl işlediğini anlarsak, teorisini araştırıp öğrenmek de daha kolay olacaktır.

### **Neden Nesne Tabanlı Programlama?**

İsterseniz önce kendimizi biraz yüreklendirip cesaretlendirelim. Şu soruyu soralım kendimize: Nesne Tabanlı Programlama'ya hiç girmesem olmaz mı?

Bu soruyu cevaplandırmadan önce bakış açımızı şöyle belirleyelim. Daha doğrusu bu soruyu iki farklı açıdan inceleyelim: **NTP'yi öğrenmek** ve **NTP'yi kullanmak**...

Eğer yukarıdaki soruya, "NTP'yi kullanmak" penceresinden bakarsak, cevabımız, "Evet," olacaktır. Yani, "Evet, NTP'yi kullanmak zorunda değilsiniz". Bu bakımdan NTP'yle ilgilenmek istemeyebilirsiniz, çünkü Python başka bazı dillerin aksine NTP'yi dayatmaz. İyi bir Python programcısı olmak için NTP'yi kullanmasanız da olur. NTP'yi kullanmadan da gayet başarılı programlar yazabilirsiniz. Bu bakımdan önünüzde bir engel yok.

Ama eğer yukarıdaki soruya "NTP'yi öğrenmek" penceresinden bakarsak, cevabımız, "Hayır", olacaktır. Yani, "Hayır, NTP'yi öğrenmek zorundasınız!". Bu bakımdan NTP'yle ilgilenmeniz gerekir, çünkü siz NTP'yi kullanmasanız da başkaları bunu kullanıyor. Dolayısıyla, NTP'nin bütün erdemlerini bir kenara bıraksak dahi, sırf başkalarının yazdığı kodları anlayabilmek için bile olsa, elinizi NTP'yle kirletmeniz gerekecektir... Bir de şöyle düşünün: Gerek internet üzerinde olsun, gerekse basılı yayınlarda olsun, Python'a ilişkin pek çok kaynakta kodlar bir noktadan sonra NTP yapısı içinde işlenmektedir. Bu yüzden özellikle başlangıç seviyesini geçtikten sonra karşınıza çıkacak olan kodları anlayabilmek için bile NTP'ye bir aşinalığınızın olması gerekir.

Dolayısıyla en başta sorduğumuz soruya karşılık ben de size şu soruyu sormak isterim:

"Daha nereye kadar kaçacaksınız bu NTP'den?"

Dikkat ederseniz, bildik anlamda NTP'nin faydalarından, bize getirdiği kolaylıklardan hiç bahsetmiyoruz. Zira şu anda içinde bulunduğumuz noktada bunları bilmenin bize pek faydası dokunmayacaktır. Çünkü daha NTP'nin ne olduğunu dahi bilmiyoruz ki cicili bicili cümlelerle bize

anlatılacak "faydaları" özümseyebilelim... NTP'yi öğrenmeye çalışan birine birkaç sayfa boyunca "NTP şöyle iyidir, NTP böyle hoştur," demenin pek faydası olmayacaktır. Çünkü böyle bir çaba, konuyu anlatan kişiyi ister istemez okurun henüz bilmediği kavramları kullanarak bazı şeyleri açıklamaya çalışmaya itecektir. Bu da okurun zihninde birtakım fantastik cümlelerin uçuşmasından başka bir işe yaramayacaktır. Dolayısıyla, NTP'nin faydalarını size burada bir çırpıda saymak yerine, öğrenme sürecine bırakıyoruz bu "özümseme" işini... NTP'yi öğrendikten sonra, bu programlama tekniğinin Python deneyiminize ne tür bir katkı sağlayacağını, size ne gibi bir fayda getireceğini kendi gözlerinizle göreceksiniz.

En azından biz bu noktada şunu rahatlıkla söyleyebiliriz: NTP'yi öğrendiğinizde Python Programlama'da bir anlamda "boyut atlamış" olacaksınız. Sonunda özgüveniniz artacak, orada burada Python'a ilişkin okuduğunuz şeyler zihninizde daha anlamlı izler bırakmaya başlayacaktır.

## Sınıflar

NTP’de en önemli kavram “sınıflar”dır. Zaten NTP denince ilk akla gelen şey de genellikle “sınıflar” olmaktadır. Sınıflar yapı olarak “fonksiyonlara” benzetilebilir. Hatırlarsanız, fonksiyonlar yardımıyla farklı değişkenleri ve veri tiplerini, tekrar kullanılmak üzere bir yerde toplayabiliyorduk. İşte sınıflar yardımıyla da farklı fonksiyonları, veri tiplerini, değişkenleri, metotları gruplandırabiliyoruz.

## Sınıf Tanımlamak

Öncelikle bir sınıfı nasıl tanımlayacağımıza bakmamız gerekiyor. Hemen, bir sınıfı nasıl tanımlayacağımızı bir örnekle görmeye çalışalım:

Python’da bir sınıf oluşturmak için şu yapıyı kullanıyoruz:

**class İlkSınıf:**

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 İlkSınıf:**

- Tıpkı fonksiyonlarda olduğu gibi, isim olarak herhangi bir kelimeyi seçebiliriz. Mesela yukarıdaki fonksiyonda “deneme” adını seçmiştik. Yine yukarıda gördüğümüz sınıf örneğinde de “İlkSınıf” 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 Sınıf" veya "class HerhangiBirKelime". Gördüğünüz gibi sınıf adı birden fazla kelimedenden 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 İlkSınıf:**

**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 çalıştıracığımıza bakalım:

Herhangi bir Python programını nasıl çalıştırıyorsak sınıfları da öyle çalıştırabiliriz. Yani pek çok farklı yöntem kullanabiliriz. Örneğin yazdığımız şey arayüzü olan bir Tkinter programıysa "python programadı.py" komutuyla bunu çalıştırabilir, yazdığımız arayüzü görebiliriz. Hatta gerekli ayarlamaları yaptıktan sonra programın simgesine çift tıklayarak veya GNU/Linux sistemlerinde konsol ekranında programın sadece adını yazarak çalıştırabiliriz programımızı. Eğer komut satırından çalışan bir uygulama yazdıysak, yine "python programadı.py" komutuyla programımızı çalıştırıp konsol üzerinden yönetebiliriz. Ancak bizim şimdilik yazdığımız kodun bir arayüzü yok.

Üstelik bu sadece NTP'yi öğrenmek için yazdığımız, tam olmayan bir kod parçasından ibaret. Dolayısıyla sınıfımızı tecrübe etmek için biz şimdilik doğrudan Python komut satırı içinden çalışacağız.

Şu halde herkes kendi platformuna uygun şekilde Python komut satırını başlatsın! Python'u başlattıktan sonra bütün platformlarda şu komutu vererek bu kod parçasını çalıştırılabilir duruma getirebiliriz:

```
>>>from sinif import *
```

Burada sizin bu kodları “sinif.py” adlı bir dosyaya kaydettiğinizi varsaydım. Dolayısıyla bu şekilde dosyamızı bir modül olarak içe aktarabiliyoruz (import). Bu arada Python'un bu modülü düzgün olarak içe aktarabilmesi için komut satırını, bu modülün bulunduğu dizin içinde açmak gerekir. Python içe aktarılacak modülleri ararken ilk olarak o anda içinde bulunulan dizine bakacağı için modülümüzü rahatlıkla bulabilecektir.

GNU/Linux kullanıcıları komut satırıyla daha içli dışlı oldukları için etkileşimli kabuğu modülün bulunduğu dizinde nasıl açacaklarını zaten biliyorlardır... Ama biz yine de hızlıca üzerinden geçelim...(Modülün masaüstünde olduğunu varsayıyoruz):

ALT+F2 tuşlarına basıp açılan pencereye "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:

```
C:/python25/python
```

Tabii kullandığınız Python sürümünün 2.5 olduğunu varsaydım. Sizde sürüm farklıysa komutu ona



göre değiştirmelisiniz.

Eğer herhangi bir hata yapmadıysanız karşınıza şuna benzer bir ekran gelmeli:

```
C:\Documents and Settings\Isminiz>c:/python25/Python
```

```
Python 2.5.1 (r251:54863, Apr 18 2007, 08:51:08) [MSC v.1310 32 bit (Intel)] on  
win32
```

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

```
>>>
```

Şimdi bu ekrandaki ">>>" satırından hemen sonra şu komutu verebiliriz:

```
from sinif import *
```

Artık sınıfımızı çalıştırmamızın önünde hiç bir engel kalmadı sayılır. Bu noktada yapmamız gereken tek bir işlem var: 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” (instance) diyoruz. Buna göre, burada “deneme” adlı değişken, “IlkSinif” adlı sınıfın bir örneğidir (“*deneme*” is an instance of the class “*IlkSinif*”). Daha soyut bir ifadeyle, örnekleme işlemi “Class” (sınıf) nesnesini etkinleştirmeye yarar. Yani sınıfın bütününe alır ve onu paketleyip, istediğimiz şekilde kullanabileceğimiz bir nesne haline getirir. Şöyle de diyebiliriz:

Biz bir sınıf tanımlıyoruz. Bu sınıfın içinde birtakım değişkenler, fonksiyonlar, vb. olacaktır. Hayli kaba bir benzetme olacak ama, biz bunları bir internet sayfasının içeriğine benzetebiliriz. İşte biz bu

sınıfı "örneklediğimiz" zaman, sınıf içeriğini bir bakıma erişilebilir hale getirmiş oluyoruz. Tıpkı bir internet sayfasının, "www..." şeklinde gösterilen adresi gibi... Mesela `www.python.quotaless.com` adresi içindeki bütün bilgileri bir sınıf olarak düşünürsek, "`www.python.quotaless.com`" ifadesi bu sınıfın bir örneğidir... Durum tam olarak böyle olmasa bile, bu benzetme, "örnekleme" işlemine ilişkin en azından zihnimizde bir kıvılcım çakmasını sağlayabilir.

Daha yerinde bir benzetme şöyle olabilir: "İnsan"ı büyük bir sınıf olarak kabul edelim. İşte "siz" (yani Ahmet, Mehmet, vb...) bu büyük sınıfın bir örneği, yani ete kemiğe bürünmüş hali oluyorsunuz... Buna göre "insan" sınıfı insanın ne tür özellikleri olduğuna dair tanımlar (fonksiyonlar, veriler) içeriyor. "Mehmet" örneği (instance) ise bu tanımları, nitelikleri, özellikleri taşıyan bir "nesne" oluyor...

### 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 bir çok programlama dilinde) yazdığımız programlar içindeki "işe yaramayan" veriler bellekten silinir. Böylece etkili bir hafıza yönetimi uygulanmış ve programların performansı artırılmış olur. Mesela:

```
a = 5
```

```
a = a + 6
```

```
print a
```

```
11
```

Burada "a" değişkeninin gösterdiği "5" verisi, daha sonra gelen "`a = a + 6`" ifadesi nedeniyle boşa düşmüş, ıskartaya çıkmış oluyor. Yani "`a = a + 6`" ifadesi nedeniyle, "a" değişkeni artık "5" verisini göstermiyor. Dolayısıyla "5" verisi o anda bellekte boşu boşuna yer kaplamış oluyor. Çünkü "`a = a + 6`" ifadesi yüzünden, "5" verisine gönderme yapan, onu gösteren, bu veriye bizim ulaşmamızı sağlayacak hiç bir işaret kalmamış oluyor ortada. İşte Python, bir veriye işaret eden hiç bir referans kalmadığı durumlarda, yani o veri artık işe yaramaz hale geldiğinde, otomatik olarak "çöp toplama" işlemini devreye sokar ve bu örnekte "5" verisini çöpe gönderir. Yani artık o veriyi bellekte tutmaktan vazgeçer. İşte eğer biz de yukarıda olduğu gibi sınıflarımızı "örneklemezsek", bu sınıflara hiçbir yerde işaret edilmediği, yani bu sınıfı gösteren hiçbir "referans" olmadığı için, sınıfımız oluşturulduğu anda çöp toplama işlemine tabi tutulacaktır. Dolayısıyla artık bellekte

tutulmayacaktır.

"Çöp Toplama" işlemini de kısaca anlattığımıza göre artık kaldığımız yerden yolumuza devam edebiliriz...

Bu arada dikkat ettiyseniz sınıfımızı örneklerken parantez kullandık. Yani şöyle yaptık:

```
deneme = IlkSinif()
```

Eğer parantezleri kullanmazsak, yani "deneme = IlkSinif" gibi bir şey yazarsak, yaptığımız şey "örnekleme" olmaz. Böyle yaparak sınıfı sadece kopyalamış oluruz... Bizim yapmak istediğimiz bu değil. O yüzden, "parantezlere dikkat!" diyoruz...

Artık şu komut yardımıyla, sınıf örneğimizin "niteliklerine" ulaşabiliriz:

```
>>>deneme.mesele
```

Olmak ya da olmamak

### 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
```

1. İlk satırda "Toplama" adlı bir sınıf tanımladık. Bunu yapmak için "class" parçacığından yararlandık.
2. 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 (örneği biraz sonra tanımlayacağız) 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 “örneklıyoruz”. Bu işleme “örnekleme” (instantiation) adı veriyoruz. “sonuc” kelimesine ise Python'cada “örnek” (instance) adı veriliyor. Yani “sonuc”, “Toplama” sınıfının bir örneğidir, diyoruz...

Artık,

```
sonuc.a
```

```
sonuc.b
```

```
sonuc.c
```

biçiminde, "sonuc" örneğinin niteliklerine tek tek erişebiliriz.

Peki kodları şöyle çalıştırırsak ne olur?

```
import matematik
```

Eğer modülü bu şekilde içe aktarırsak (import), sınıf örneğinin niteliklerine ulaşmak için şu yapıyı kullanmamız gerekir:

```
matematik.sonuc.a
```

```
matematik.sonuc.b
```

```
matematik.sonuc.c
```

Yani her defasında dosya adını (ya da başka bir ifadeyle “modülün adını”) da belirtmemiz gerekir.

Bu iki kullanım arasında, özellikle sağladıkları güvenlik avantajları/dezavantajları açısından başka bazı temel farklılıklar da vardır, ama şimdilik konumuzu dağıtmamak için bunlara girmiyoruz...

Ama temel olarak şunu bilmekte fayda var: Genellikle tercih edilmesi gereken yöntem "from modül

import \*" yerine "import modül" biçimini kullanmaktır. Eğer "from modül import \*" yöntemini kullanarak içe aktardığınız modül içindeki isimler (değişkenler, nitelikler), bu modülü kullanacağınız dosya içinde de bulunuyorsa isim çakışmaları ortaya çıkabilir... Esasında, "from modül import \*" yapısını sadece ne yaptığımızı çok iyi biliyorsak ve modülle ilgili belgelerde modülün bu şekilde içe aktarılması gerektiği bildiriliyorsa kullanmamız yerinde olacaktır. Mesela Tkinter ile programlama yaparken rahatlıkla "from Tkinter import \*" yapısını kullanabiliriz, çünkü Tkinter bu kullanımda problem yaratmayacak şekilde tasarlanmıştır. Yukarıda bizim verdiğimiz örnekte de "from modül import \*" yapısını rahatlıkla kullanıyoruz, çünkü şimdilik tek bir modül üzerinde çalışıyoruz. Dolayısıyla isim çakışması yaratacak başka bir modülümüz olmadığı için "ne yaptığımızı biliyoruz!"...

Yukarıda anlattığımız kod çalıştırma biçimleri tabii ki, bu kodları komut ekranından çalıştırdığınızı varsaymaktadır. Eğer siz bu kodları IDLE ile çalıştırmak isterseniz, bunları hazırladıktan sonra F5 tuşuna basmanız, veya “Run > Run Module” yolunu takip etmeniz yeterli olacaktır. F5’e bastığınızda veya “Run > Run Module” yolunu takip ettiğinizde IDLE sanki komut ekranında “from matematik import \*” komutunu vermişsiniz gibi davranacaktır.

Veya GNU/Linux sistemlerinde sistem konsolunda

`python -i sınıf.py`

komutunu vererek de bu kod parçalarını çalıştırılabilir duruma getirebiliriz. Bu komutu verdiğimizde "from sınıf 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`

1. “Toplama” adlı bir sınıf tanımlıyoruz.
2. Sınıfımızın içine istediğimiz kod parçalarını ekliyoruz. Biz burada üç adet değişken ekledik. Bu

değişkenlerin her birine, “nitelik” adını veriyoruz.

3. Bu kodları kullanabilmek için Python komut satırında şu komutu veriyoruz:

```
from matematik import *
```

Burada modül adının (yani dosya adının) matematik olduğunu varsaydık.

4. Şimdi yapmamız gereken şey, Toplama adlı sınıfı “örnekleme” (instantiation). Yani bir nevi, sınıfın kendisini bir değişkene atamak. Bu değişkene biz Python’cada “örnek” (instance) adını veriyoruz. Yani, “sonuc” adlı değişken, “Toplama” adlı sınıfın bir örneğidir diyoruz (*sonuc is an instance of Toplama*).

```
>>>sonuc = Toplama()
```

5. Bu komutu verdikten sonra niteliklerimize erişebiliriz:

```
>>>sonuc.a
```

```
>>>sonuc.b
```

```
>>>sonuc.c
```

Dikkat ederseniz, niteliklerimize erişirken “örnek”ten (instance), yani “sonuc” adlı değişkenden yararlanıyoruz.

Şimdi bir an bu sınıfımızı örneklemediğimizi düşünelim. Dolayısıyla bu sınıfı şöyle kullanmamız gerekecek:

```
>>>Toplama().a
```

```
>>>Toplama().b
```

```
>>>Toplama().c
```

Ama daha önce de anlattığımız gibi, siz "Toplama().a" der demez sınıf çalıştırılacak ve 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 terketmesi sağlanacaktır. Bu yüzden bunu her çalıştırdığınızda yeniden belleğe yüklemiş olacaksınız sınıfı. Bu da bir hayli verimsiz bir çalışma şeklidir.

Böylelikle zor kısmı geride bırakmış olduk. Artık önümüze bakabiliriz. Zira en temel bazı kavramları gözden geçirdiğimiz ve temelimizi oluşturduğumuz için, daha karışık şeyleri anlamak kolaylaşacaktır.

## `__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:utf8 -*-
```

```
class Oyun:
```

```
    def __init__(self):
        enerji = 50
        para = 100
        fabrika = 4
        isci = 10
        print "enerji:", enerji
        print "para:", para
        print "fabrika:", fabrika
        print "işçi:", isci
```

```
macera = Oyun()
```

Gayet güzel. Dikkat ederseniz "örnekleme" (instantiation) işlemini doğrudan dosya içinde hallettik. Komut satırına bırakmadık bu işi.

Şimdi bu kodları çalıştıracakız. Bir kaç seçeneğimiz var:

1. 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.

2. GNU/Linux sistemlerinde başka bir seçenek olarak, ALT+F2 tuşlarına basıyoruz ve açılan

pencerde "konsole" (KDE) veya "gnome-terminal" (GNOME) yazıp enter'e bastıktan sonra açtığımız komut satırında şu komutu veriyoruz:

`python -i deneme.py`

3. Eğer Windows'ta IDLE üzerinde çalışıyorsak, F5 tuşuna basarak veya *"Run>Run Module"* yolunu takip ederek kodlarımızı çalıştırıyoruz.

Bu kodları yukarıdaki seçeneklerden herhangi biriyle çalıştırdığımızda, `__init__` fonksiyonu içinde tanımlanmış olan bütün değişkenlerin, yani "niteliklerin", ilk çalışma esnasında ekrana yazdırıldığını görüyoruz. İşte bu niteliklerin başlangıç değeri olarak belirlenebilmesi hep `__init__` fonksiyonu sayesinde olmaktadır. Dolayısıyla şöyle bir şey diyebiliriz:

Python'da bir programın ilk kez çalıştırıldığı anda işlemlerini istediğimiz şeyleri bu `__init__` fonksiyonu içine yazıyoruz. Mesela yukarıdaki ufak oyun çalışmasında, oyuna başlandığı anda bir oyuncunun sahip olacağı özellikleri `__init__` fonksiyonu içinde tanımladık. Buna göre bu oyunda bir oyuncu oyuna başladığında;

enerjisi 50,  
parası 100  
fabrika sayısı 4,  
işçi sayısı ise 10 olacaktır.

Yalnız hemen uyaralım: Yukarıdaki örnek aslında pek de düzgün sayılmaz. Çok önemli eksiklikleri var bu kodun. Ama şimdi konumuz bu değil... Olayın iç yüzünü kavrayabilmek için öyle bir örnek vermemiz gerekiyordu. Bunu biraz sonra açıklayacağız. Biz okumaya devam edelim...

Bir de Tkinter ile bir örnek yapalım. Zira sınıflı yapıların en çok ve en verimli kullanıldığı yer arayüz programlama çalışmalarımızdır:

```
from Tkinter import *  
class Arayuz:  
    def __init__(self):  
        pencere = Tk()  
        dugme = Button(text="tamam")  
        dugme.pack()
```



## uygulama = Arayuz()

Bu kodları da yukarıda saydığımız yöntemlerden herhangi biri ile çalıştırıyoruz. Tabii ki bu kod da eksiksiz değildir. Ancak şimdilik amacımıza hizmet edebilmesi için kodlarımızı bu şekilde yazmamız gerekiyordu. Ama göreceğiniz gibi yine de çalışıyor bu kodlar... Dikkat ederseniz burada da örnekleme işlemini dosya içinden hallettik. Eğer örnekleme satırını dosya içine yazmazsak, Tkinter penceresinin açılması için komut satırında "uygulama = Arayuz()" gibi bir satır yazmamız gerekir.

Buradaki `__init__` fonksiyonu sayesinde “Arayuz” adlı sınıf her çağrıldığında bir adet Tkinter penceresi ve bunun içinde bir adet düğme otomatik olarak oluşacaktır. Zaten bu `__init__` fonksiyonuna da İngilizce’de çoğu zaman “constructor” (oluşturan, inşa eden, meydana getiren) adı verilir. Gerçi `__init__` fonksiyonuna "constructor" demek pek doğru bir ifade sayılmaz, ama biz bunu şimdi bir kenara bırakalım. Sadece aklımızda olsun, `__init__` fonksiyonu gerçek anlamda bir "constructor" değildir, ama ona çok benzer...

Şöyle bir yanlış anlaşılma olmamasına dikkat edin:

“`__init__`” fonksiyonunun, “varsayılan değerleri belirleme”, yani “inşa etme” özelliği konumundan kaynaklanmıyor. Yani bu `__init__` fonksiyonu, işlevini sırf ilk sırada yer aldığı için yerine getirmiyor. Bunu test etmek için, isterseniz yukarıdaki kodları “`__init__`” fonksiyonunun adını değiştirerek çalıştırmayı deneyin. Aynı işlevi elde edemezsiniz... Mesela `__init__` yerine `__simit__` deyin. Çalışmaz...

`__init__` konusuna biraz olsun ışık tuttuğumuza göre artık en önemli bileşenlerden ikincisine gelebiliriz: `self`

## self Nedir?

Bu küçük kelime Python’da sınıfların en can alıcı noktasını oluşturur. Esasında çok basit bir işlevi olsa da, bu işlevi kavrayamazsak neredeyse bütün bir sınıf konusunu kavramak imkansız hale gelecektir. Self’i anlamaya doğru ilk adımı atmak için yukarıda kullandığımız kodlardan faydalanarak bir örnek yapmaya çalışalım. Kodumuz şöyleydi:

## class Oyun:

```
def __init__(self):
    enerji = 50
    para = 100
    fabrika = 4
    isci = 10
    print "enerji:", enerji
    print "para:", para
    print "fabrika:", fabrika
    print "işçi:", isci
```

```
macera = Oyun()
```

Diyelim ki biz burada “enerji, para, fabrika, işçi” değişkenlerini ayrı bir fonksiyon içinde kullanmak istiyoruz. Yani mesela “göster” adlı ayrı bir fonksiyonumuz olsun ve biz bu değişkenleri ekrana yazdırmak istediğimizde bu “göster” fonksiyonundan yararlanalım. Kodların şu anki halinde olduğu gibi, bu kodlar tanımlansın, ama doğrudan ekrana dökülmesin. Şöyle bir şey yazmayı deneyelim. Bakalım sonuç ne olacak?

```
class Oyun:
```

```
    def __init__(self):
        enerji = 50
        para = 100
        fabrika = 4
        isci = 10

    def goster():
        print "enerji:", enerji
        print "para:", para
        print "fabrika:", fabrika
        print "işçi:", isci
```

```
macera = Oyun()
```

Öncelikle bu kodların sahip olduğu “niteliklere” bir bakalım:

```
enerji, para, fabrika, işçi ve goster()
```

Burada “örneğimiz” (instance) “macera” adlı değişken. Dolayısıyla bu niteliklere şu şekilde ulaşabiliriz:

```
macera.enerji
```

```
macera.para
macera.fabrika
macera.isci
macera.goster()
```

Hemen deneyelim. Ama o da ne? Mesela “macera.goster()” dediğimizde şöyle bir hata alıyoruz:

```
Traceback (most recent call last):
File "<pyshell#0>", 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 "<pyshell#0>", line 1, in <module>
macera.goster()
File "xxxxxxxxxxxxxxxxxxxxxx", 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. Yani mesela, “enerji” yerine “self.enerji” diyerek, bu “enerji” adlı değişkenin yalnızca içinde bulunduğu fonksiyonda değil, o fonksiyonun dışında da kullanılabilmesini sağlıyoruz. İyice somutlaştırmak gerekirse, “\_\_init\_\_” fonksiyonu içinde tanımladığımız “enerji” adlı değişken, bu haliyle “goster” adlı fonksiyonun içinde kullanılamaz. Daha da önemlisi bu kodları bu haliyle tam olarak çalıştıramayız da. Mesela şu temel komutları işletemeyiz:

```
macera.enerji
macera.para
macera.isci
macera.fabrika
```

Eğer biz “enerji” adlı değişkeni “goster” fonksiyonu içinde kullanmak istersek değişkeni sadece “enerji” değil, “self.enerji” olarak tanımlamamız gerekir. Ayrıca bunu “goster” adlı fonksiyon içinde

kullanırken de sadece “enerji” olarak değil, “self.enerji” olarak yazmamız gerekir. Üstelik mesela "enerji" adlı değişkeni herhangi bir yerden çağırmak istediğimiz zaman da bunu önceden "self" olarak tanımlamış olmamız gerekir.

Şimdi tekrar deneyelim:

```
macera.goster
```

```
enerji: 50
```

```
para: 100
```

```
fabrika: 4
```

```
işçi: 10
```

```
macera.enerji
```

```
50
```

```
macera.para
```

```
100
```

```
macera.fabrika
```

```
4
```

```
macera.isci
```

```
10
```

Sınıfın niteliklerine tek tek nasıl erişebildiğimizi görüyorsunuz... Bu arada, isterseniz "self'i, "macera" örneğinin yerini tutan bir kelime olarak da kurabilirsiniz zihninizde. Yani kodları çalıştırırken "macera.enerji" diyebilmek için, en başta bunu "self.enerji" olarak tanımlamamız gerekiyor... Bu düşünme tarzı işimizi biraz daha kolaylaştırabilir.

Bir de Tkinter’li örneğimize bakalım:

```
from Tkinter import *
```

```
class Arayuz:
```

```
    def __init__(self):
```

```
        pencere = Tk()
```

```
        dugme = Button(text="tamam")
```

```
        dugme.pack()
```

```
uygulama = Arayuz()
```

Burada tanımladığımız düğmenin bir iş yapmasını sağlayalım. Mesela düğmeye basılınca komut ekranında bir yazı çıksın. Önce şöyle deneyelim:

```
from Tkinter import *
```

```
class Arayuz:
```

```
    def __init__(self):
```

```
        pencere = Tk()
```

```
        dugme = Button(text="tamam",command=yaz)
```

```
        dugme.pack()
```

```
    def yaz():
```

```
        print "Hadi eyvallah!"
```

```
uygulama = Arayuz()
```

Tabii ki bu kodları çalıştırdığımızda şöyle bir hata mesajı alırız:

```
Traceback (most recent call last):
File "xxxxxxxxxxxxxxxxxxxxxx", line 13, in <module>
uygulama = Arayuz()
File "xxxxxxxxxxxxxxxxxxxxxx", line 7, in __init__
dugme = Button(text="tamam",command=yaz)
```

NameError: global name 'yaz' is not defined

Bunun sebebini bir önceki örnekte öğrenmiştik. Kodlarımızı şu şekilde yazmamız gerekiyor:

```
from Tkinter import *
```

```
class Arayuz:
```

```
    def __init__(self):
```

```
        pencere = Tk()
```

```
        dugme = Button(text="tamam",command=self.yaz)
```

```
        dugme.pack()
```

```
    def yaz(self):
```

```
        print "Hadi eyvallah!"
```

```
uygulama = Arayuz()
```

Gördüğünüz gibi, eğer programın farklı noktalarında kullanacağımız değişkenler veya fonksiyonlar varsa, bunları “self” öneki ile birlikte tanımlıyoruz. “def self.yaz” şeklinde bir fonksiyon tanımlama

yöntemi olmadığına göre bu işlemi “def yaz(self)” şeklinde yapmamız gerekiyor. Bu son örnek aslında yine de tam anlamıyla kusursuz bir örnek değildir. Ama şimdilik elimizden bu kadarı geliyor. Daha çok bilginiz olduğunda bu kodları daha düzgün yazmayı da öğreneceğiz.

Bu iki örnek içinde, “self”lerle oynayarak olayın iç yüzünü kavramaya çalışın. Mesela yaz() fonksiyonundaki self parametresini silince ne tür bir hata mesajı alıyorsunuz, “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:

```
from Tkinter import *  
class Arayuz:  
    def __init__(armut):  
        pencere = Tk()  
        dugme = Button(text="tamam",command=armut.yaz)  
        dugme.pack()  
  
    def yaz(armut):  
        print "Hadi eyvallah!"
```

```
uygulama = Arayuz()
```

Ama siz böyle yapmayın. “self” kelimesinin kullanımı o kadar yaygınlaşmış ve yerleşmiştir ki, sizin bunu kendi kodlarınızda dahi olsa değiştirmeye kalkmanız pek hoş karşılanmayacaktır. Ayrıca sizin kodlarınızı okuyan başkaları, ne yapmaya çalıştığınızı anlamakta bir an da olsa tereddüt edecektir. Hatta birkaç yıl sonra dönüp siz dahi aynı kodlara baktığınızda, “Ben burada ne yapmaya çalışmışım,” diyebilirsiniz... O yüzden, “self” iyidir, “self” kullanın!...

Sizi “self” kullanmaya ikna ettiğimizi kabul edersek, artık yolumuza devam edebiliriz.

Hatırlarsanız yukarıda ufacık bir oyun çalışması yapmaya başlamıştık. Gelin isterseniz oyunumuzu biraz ayrıntılandıralım. Elimizde şimdilik şunlar vardı:

```
class Oyun:  
    def __init__(self):
```

```
self.enerji = 50
self.para = 100
self.fabrika = 4
self.isci = 10
```

```
def goster(self):
    print "enerji:", self.enerji
    print "para:", self.para
    print "fabrika:", self.fabrika
    print "işçi:", self.isci
```

macera = Oyun()

Buradaki kodlar yardımıyla bir oyuncu oluşturduk. Bu oyuncunun oyuna başladığında sahip olacağı enerji, para, fabrika ve işçi bilgilerini de girdik. Kodlarımız arasındaki “goster()” fonksiyonu yardımıyla da her an bu bilgileri görüntüleyebiliyoruz.

Şimdi isterseniz oyunumuza biraz hareket getirelim. Mesela kodlara yeni bir fonksiyon ekleyerek oyuncumuza yeni fabrikalar kurma olanağı tanıyalım:

class Oyun:

```
def __init__(self):
    self.enerji = 50
    self.para = 100
    self.fabrika = 4
    self.isci = 10

def goster(self):
    print "enerji:", self.enerji
    print "para:", self.para
    print "fabrika:", self.fabrika
    print "işçi:", self.isci
```

```
def fabrikakur(self,miktar):
    if self.enerji > 3 and self.para > 10:
```



```
self.fabrika = miktar + self.fabrika
self.enerji = self.enerji - 3
self.para = self.para - 10
print miktar, "adet fabrika kurdunuz! Tebrikler!"
```

else:

```
print "Yeni fabrika kuramazsınız. Çünkü 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()
```

Bu komut şu çıktıyı verecektir:

```
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. Çünkü 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? Tabii ki hayır. O halde nasıl yapacağız bunu? İşte burada imdadımıza Python sınıflarının “miras alma” özelliği yetişiyor. Yabancılar bu kavrama “inheritance” adını veriyorlar. Yani, nasıl Mısır’daki dedenizden size miras kaldığında dedenizin size bıraktığı mirasın nimetlerinden her yönüyle yararlanabiliyorsanız, bir sınıf başka bir sınıftan miras 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 alduralım. Bunun için sınıfımızı şöyle tanımlamamız gerekiyor:

**class Dusman(Oyun):**

Böylelikle daha en başta tanımladığımız “Oyun” adlı sınıfı, bu yeni oluşturduğumuz “Dusman” adlı sınıfa miras verdik. Dusman sınıfının durumunu Python’cada şöyle ifade edebiliriz:

*“Dusman sınıfı Oyun sınıfını miras aldı” (Dusman inherits from Oyun)*

Bu haliyle kodlarımız henüz eksik. Şimdilik şöyle bir şey yazıp sınıfımızı kitabına uyduralım:

**class Dusman(Oyun):**

**pass**

**dsman = Dusman()**

Yukarıda “pass” ifadesini neden kullandığımızı biliyorsunuz. Sınıfı tanımladıktan sonra iki nokta üst üstenin ardından aşağıya bir kod bloğu yazmamız gerekiyor. Ama şu anda oraya yazacak bir kodumuz yok. O yüzden idareten oraya bir pass ifadesi yerleştirerek gerekli kod bloğunu geçiştirmiş oluyoruz. O kısmı boş bırakamayız. Yoksa sınıfımız kullanılamaz durumda olur. Daha sonra oraya yazacağımız kod bloklarını hazırladıktan sonra oradaki “pass” ifadesini sileceğiz.

Şimdi bakalım bu sınıfla neler yapabiliyoruz?

Bu kodları, yazının başında anlattığımız şekilde çalıştıralım. Dediğimiz gibi, “Dusman” adlı sınıfımız daha önce tanımladığımız “Oyun” adlı sınıfı miras alıyor. Dolayısıyla “Dusman” adlı sınıf “Oyun” adlı sınıfın bütün özelliklerine sahip. Bunu hemen test edelim:

```
dsman.goster()
```

```
enerji: 50
```

```
para: 100
```

```
fabrika: 4
```

```
işçi: 10
```

Gördüğünüz gibi, Oyun sınıfının bir fonksiyonu olan “goster”i “Dusman” sınıfı içinden de çalıştırabildik. Üstelik Dusman içinde bu değişkenleri tekrar tanımlamak zorunda kalmadan... İstersek bu değişkenlere teker teker de ulaşabiliriz:

```
dsman.enerji
```

```
50
```

```
dsman.isci
```

```
10
```

Dusman sınıfı aynı zamanda Oyun sınıfının “fabrikakur” adlı fonksiyonuna da erişebiliyor:

```
dsman.fabrikakur(4)
```

4 adet fabrika kurdunuz! Tebrikler!

Gördüğünüz gibi düşmanımız kendisine 4 adet fabrika kurdu!.. Düşmanımızın durumuna bakalım:

```
dsman.goster()
```

```
enerji: 47
```

```
para: 90
```

```
fabrika: 8
```

```
işçi: 10
```

Evet, düşmanımızın fabrika sayısı artmış, enerjisi ve parası azalmış. Bir de kendi durumumuzu kontrol edelim:

```
macera.goster()
```

```
enerji: 50
```

```
para: 100
```

```
fabrika: 4
```

```
işçi: 10
```

Dikkat ederseniz, Oyun ve Dusman sınıfları aynı değişkenleri kullandıkları halde birindeki değişiklik öbürünü etkilemiyor. Yani düşmanımızın yeni bir fabrika kurması bizim değerlerimizi değişikliğe uğratmıyor.

Şimdi şöyle bir şey yapalım:

Düşmanımızın, oyuncunun özelliklerine ek olarak bir de “ego” adlı bir niteliği olsun. Mesela düşmanımız bize her zarar verdiğinde egosu büyüsün!...

Önce şöyle deneyelim:

```
class Dusman(Oyun):  
    def __init__(self):  
        self.ego = 0
```

Bu kodları çalıştırdığımızda hata alırız. Çünkü burada yeni bir “\_\_init\_\_” fonksiyonu tanımladığımız için, bu yeni fonksiyon kendini Oyun sınıfının \_\_init\_\_ fonksiyonunun üzerine yazıyor. Dolayısıyla Oyun sınıfından miras aldığımız bütün nitelikleri kaybediyoruz. Bunu önlemek için şöyle bir şey yapmamız gerekir:

```
class Dusman(Oyun):  
    def __init__(self):  
        Oyun.__init__(self)  
        self.ego = 0
```

Burada “Oyun.\_\_init\_\_(self)” ifadesiyle “Oyun” adlı sınıfın “\_\_init\_\_” fonksiyonu içinde yer alan bütün nitelikleri, “Dusman” adlı sınıfın \_\_init\_\_ fonksiyonu içine kopyalıyoruz. Böylece “self.ego” değişkenini tanımlarken, “enerji, para, vb.” niteliklerin kaybolmasını engelliyoruz.

Aslında bu haliyle kodlarımız düzgün şekilde çalışır. Kodlarımızı çalıştırdığımızda biz ekranda göremesek de aslında “ego” adlı niteliğe sahiptir düşmanımız. Ekranda bunu göremememizin nedeni tabii ki kodlarımızda henüz bu niteliği ekrana yazdıracak bir “print” deyiminin yer almaması... İsterseniz bu özelliği daha önce de yaptığımız gibi ayrı bir fonksiyon ile halledelim:

```
class Dusman(Oyun):  
    def __init__(self):  
        Oyun.__init__(self)  
        self.ego = 0  
  
    def goster(self):  
        Oyun.goster(self)  
        print "ego:", self.ego
```

```
dsman = Dusman()
```

Tıpkı “\_\_init\_\_” fonksiyonunda olduğu gibi, burada da “Oyun.goster(self)” ifadesi yardımıyla “Oyun” sınıfının “goster()” fonksiyonu içindeki değişkenleri “Dusman” sınıfının “goster()” fonksiyonu içine kopyaladık. Böylece “ego” değişkenini yazdırırken, öteki değişkenlerin de yazdırılmasını sağladık.

Şimdi artık düşmanımızın bütün niteliklerini istediğimiz şekilde oluşturmuş olduk. Hemen deneyelim:

```
dsman.goster()
```

```
enerji: 50
```

```
para: 100
```

```
fabrika: 4
```

```
işçi: 10
```

```
ego: 0
```

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 tavana vurdu!"
```

```
dsman = Dusman()
```

Dikkat ederseniz, "fabrikayik" fonksiyonu içindeki değişkeni "macera.fabrika" şeklinde yazdık.

Yani bir önceki "Oyun" adlı sınıfın "örneğini" (instance) kullandık. "Dusman" sınıfının değil...

Neden? Çok basit. Çünkü kendi fabrikalarımızı değil oyuncunun fabrikalarını yıkmak istiyoruz!..

Burada, şu kodu çalıştırarak oyuncumuzun kurduğu fabrikaları yıkabiliriz:

```
dsman.fabrikayik(2)
```

Biz burada "2" adet fabrika yıkmayı tercih ettik...

Kodlarımızın en son halini topluca görelim isterseniz:

```
class Oyun:
```

```
    def __init__(self):
```

```
        self.enerji = 50
```

```
        self.para = 100
```

```
        self.fabrika = 4
```

```
        self.isci = 10
```

```
    def goster(self):
```

```
        print "enerji:", self.enerji
```

```
        print "para:", self.para
```

```
        print "fabrika:", self.fabrika
```

```
        print "işçi:", self.isci
```

```
    def fabrikakur(self,miktar):
```

```
        if self.enerji > 3 and self.para > 10:
```

```
            self.fabrika = miktar + self.fabrika
```

```
            self.enerji = self.enerji - 3
```

```
            self.para = self.para - 10
```

```
            print miktar, "adet fabrika kurdunuz! Tebrikler!"
```

```
        else:
```

```
            print "Yeni fabrika kuramazsınız. Çünkü yeterli enerjiniz/paranız yok!"
```

```
macera = Oyun()
```

```
class Dusman(Oyun):
    def __init__(self):
        Oyun.__init__(self)
        self.ego = 0

    def goster(self):
        Oyun.goster(self)
        print "ego:", self.ego

    def fabrikayik(self,miktar):
        macera.fabrika = macera.fabrika - miktar
        self.ego = self.ego + 2
        print "Tebrikler. Oyuncunun", miktar, "adet fabrikasını yıktınız!"
        print "Üstelik egonuz da tavana vurdu!"

dsman = Dusman()
```

En son oluşturduğumuz fonksiyonda nerede “Oyun” sınıfını doğrudan adıyla kullandığımıza ve nerede bu sınıfın “örneğinden” (instance) yararlandığımıza dikkat edin. Dikkat ederseniz, fonksiyon başlıklarını çağırırken doğrudan sınıfın kendi adını kullanıyoruz (mesela “Oyun.\_\_init\_\_(self)”). Bir fonksiyon içindeki değişkenleri çağırırken ise (mesela “macera.fabrika”), “örneği” (instance) kullanıyoruz. Eğer bir fonksiyon içindeki değişkenleri çağırırken de sınıf isminin kendisini kullanmak isterseniz, ifadeyi “Oyun().\_\_init\_\_(self)” şeklinde yazmanız gerekir. Ama siz böyle yapmayın... Yani değişkenleri çağırırken örneği kullanın.

Artık kodlarımız didiklenmek üzere sizi bekliyor. Burada yapılan şeyleri iyice anlayabilmek için kafanıza göre kodları değiştirin. Neyi nasıl değiştirdiğinizde ne gibi bir sonuç elde ettiğinizi dikkatli bir şekilde takip ederek, bu konunun zihninizde iyice yer etmesini sağlayın.

Aslında yukarıdaki kodları daha düzenli bir şekilde de yazmamız mümkün. Örneğin, “enerji, para, fabrika” gibi nitelikleri ayrı bir sınıf halinde düzenleyip, öteki sınıfların doğrudan bu sınıftan miras almasını sağlayabiliriz. Böylece sınıfımız daha derli toplu bir görünüm kazanmış olur. Aşağıdaki kodlar içinde, isimlendirmeleri de biraz değiştirerek standartlaştırdığımıza dikkat edin:



```
class Oyun:
    def __init__(self):
        self.enerji = 50
        self.para = 100
        self.fabrika = 4
        self.isci = 10

    def goster(self):
        print "enerji:", self.enerji
        print "para:", self.para
        print "fabrika:", self.fabrika
        print "işçi:", self.isci

oyun = Oyun()

class Oyuncu(Oyun):
    def __init__(self):
        Oyun.__init__(self)

    def fabrikakur(self,miktar):
        if self.enerji > 3 and self.para > 10:
            self.fabrika = miktar + self.fabrika
            self.enerji = self.enerji - 3
            self.para = self.para - 10
            print miktar, "adet fabrika kurdunuz! Tebrikler!"
        else:
            print "Yeni fabrika kuramazsınız. Çünkü yeterli enerjiniz/paranız yok!"
```

```
oyuncu = Oyuncu()
```

```
class Dusman(Oyun):
    def __init__(self):
```

```
Oyun.__init__(self)
self.ego = 0
```

```
def goster(self):
    Oyun.goster(self)
    print "ego:", self.ego
```

```
def fabrikayik(self,miktar):
    oyuncu.fabrika = oyuncu.fabrika - miktar
    self.ego = self.ego + 2
    print "Tebrikler. Oyuncunun", miktar, "adet fabrikasını yıktınız!"
    print "Üstelik egonuz da tavana vurdu!"
```

```
dusman = Dusman()
```

Bu kodlar hakkında son bir noktaya daha değinelim. Hatırlarsanız oyuna başlarken oluşturulan niteliklerde değişiklik yapabiliyorduk. Mesela yukarıda “Dusman” sınıfı için “ego” adlı yeni bir nitelik tanımlamıştık. Bu nitelik sadece “Dusman” tarafından kullanılabilirdi, Oyuncu tarafından değil. Aynı şekilde, yeni bir nitelik belirlemek yerine, istersek varolan bir niteliği iptal de edebiliriz. Diyelim ki Oyuncu’nun oyuna başlarken “fabrika”ları olsun istiyoruz, ama Dusman’ın oyun başlangıcında fabrikası olsun istemiyoruz. Bunu şöyle yapabiliriz:

```
class Dusman(Oyun):
    def __init__(self):
        Oyun.__init__(self)
        del self.fabrika
        self.ego = 0
```

Gördüğünüz gibi “Dusman” sınıfı için “\_\_init\_\_” fonksiyonunu tanımlarken “fabrika” niteliğini “del” komutuyla siliyoruz. Bu silme işlemi sadece “Dusman” sınıfı için geçerli oluyor. Bu işlem öteki sınıfları etkilemiyor. Bunu şöyle de ifade edebiliriz;

“del komutu yardımıyla fabrika adlı değişkene Dusman adlı bölgeden erişilmesini engelliyoruz.”

Dolayısıyla bu değişiklik sadece o “bölgeyi” etkiliyor. Öteki sınıflar ve daha sonra oluşturulacak yeni sınıflar bu işleminden etkilenmez. Yani aslında “del” komutuyla herhangi bir şeyi sildiğimiz yok! Sadece “erişimi engelliyoruz”.

Küçük bir not: Burada “bölge” olarak bahsettiğimiz şey aslında Python’cada “isim alanı” (namespace) olarak adlandırılıyor.

Şimdi bir örnek de Tkinter ile yapalım. Yukarıda verdiğimiz örneği hatırlıyorsunuz:

```
from Tkinter import *  
class Arayuz:  
    def __init__(self):  
        pencere = Tk()  
        dugme = Button(text="tamam",command=self.yaz)  
        dugme.pack()  
  
    def yaz(self):  
        print "Hadi eyvallah!"
```

uygulama = Arayuz()

Bu örnek gayet düzgün çalışsa da bu sınıfı daha düzgün ve düzenli bir hale getirmemiz mümkün:

```
#!/usr/bin/env python  
#-*-coding:utf-8-*-  
from Tkinter import *  
class Arayuz(Frame):  
    def __init__(self):  
        Frame.__init__(self)  
        self.pack()  
        self.pencerearaclari()  
  
    def pencerearaclari(self):  
        self.dugme = Button(self,text="tamam",command=self.yaz)  
        self.dugme.pack()
```

```
def yaz(self):  
    print "Hadi eyvallah!"
```

```
uygulama = Arayuz()  
uygulama.mainloop()
```

Burada dikkat ederseniz, Tkinter'in "Frame" adlı sınıfını miras aldık. Buradan anlayacağımız gibi, "miras alma" (inheritance) özelliğini kullanmak için miras alacağımız sınıfın o anda kullandığımız modül içinde olması şart değil. Burada olduğu gibi, başka modüllerin içindeki sınıfları da miras alabiliyoruz. Yukarıdaki kodları dikkatlice inceleyin. Başta biraz karışık gibi görünse de aslında daha önce verdiğimiz basit örneklerden hiç bir farkı yoktur.

## Eski ve Yeni Sınıflar

Şimdiye kadar verdiğimiz sınıf örneklerinde önemli bir konudan hiç bahsetmedik. Python'da iki tip sınıf vardır: Eski tip sınıflar ve yeni tip sınıflar. Ancak korkmanızı gerektirecek kadar fark yoktur bu iki sınıf tipi arasında. Ayrıca hangi sınıf tipini kullanırsanız kullanın sorun yaşamazsınız. Ama tabii ki kendimizi yeni tipe alıştırmakta fayda var, çünkü muhtemelen Python'un sonraki sürümlerinden birinde (büyük ihtimalle Python 3.0'da) eski tip sınıflar kullanımdan kaldırılacaktır.

Eski tip sınıflar ile yeni tip sınıflar arasındaki en büyük fark şudur:

Eski tip sınıflar şöyle tanımlanır:

```
class Deneme:
```

Yeni tip sınıflar ise şöyle tanımlanır:

```
class Deneme(object)
```

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 içeriğini 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 imkanına kavuşacak ve okuduğunu daha kolay anlayacaktır. Bir sonraki bölümde de NTP konusunu işlemeye devam edeceğiz. Ama bu kez biraz daha ayrıntıya girerek...

## Grafik Arayüz Tasarımı // Temel Bilgiler

### Giriş

Bir programlama dilini ne kadar iyi bilerseniz 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 gocunmayacaktır, hatta sizi takdir de edecektir, ama "son kullanıcı"

denen kesime hitap etmediğiniz sürece dikkati çekmeniz güç... Bu dikkati çekme meselesi de çoğunlukla ancak cicili bicili arayüzler vasıtasıyla aşılabiliyor. Diyelim ki konsol üzerinde muhteşem işler yapan dehşetengiz bir program yazdınız ve piyasanın kucağına attınız... "Son kullanıcı", programınızı kurup çalıştırmayı denediğinde alacağınız ilk tepki: "Ama bu program çalışmıyoo!" şeklinde olacaktır... Kızısak da kızmasak da ne yazık ki durum bu.. Dost acı söyler!

Madem arayüz denen şey bu kadar önemli şu halde biz de bu bölümde Python'da nasıl grafik arayüzler tasarlayacağımızı inceleyelim.

Arayüzler tasarlamak üzere geliştirilmiş değişik alternatifler arasında biz en yaygını olan Tkinter'i kullanacağız. Aslında Tkinter Python içinde bir modül... Neyse... Lafı daha fazla uzatmadan işe koyulalım:

## Pencere Oluşturmak

Arayüz denilen şey tabii ki penceresiz olmaz. Dolayısıyla arayüz programlamanın ilk adımı çalışan bir pencere yaratmak olacaktır.

Başta söylediğimiz gibi, arayüz tasarlarken Tkinter modülünden faydalanacağız. Daha önceki yazılardan, Python'da nasıl modül "import" edildiğini hatırlıyorsunuz. Şimdi hemen Python komut satırında 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: Gidip paket yöneticisinden "python-tk" adlı paketi kuracağız.

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? Ne yalan söyleyeyim, ben bu pencereyi ilk gördüğümde çok heyecanlanmışım... Neyse... Konumuza dönelim...

Gördüğünüz gibi "Tk()" komutuyla önümüze atlayan 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 eksi işareti bütün işlevselliğiyle karşımızda duruyor. Ayrıca farkedeceğimiz gibi bu pencereyi istediğimiz gibi boyutlandırmamız da mümkün.

Bu komutları hangi platformda verdiğinize bağlı olarak, pencerenin görünüşü farklı olacaktır. Yani mesela bu komutları Windows'ta verdiyseniz, Windows'un renk ve şekil şemasına uygun bir pencere oluşacaktır. Eğer Gnome kullanıyorsanız, pencerenin şekli şemali, 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/python2.4/lib-tk/` klasörü içindeki Tkinter.py modülü içinde “class Tk” satırını arayarak bunun ne menem bir şey olduğuna bakabilir (*Gereksiz bilgi: "ne menem" ifadesinin doğrusu aslında "ne mene"... Ama hep "ne menem" biçiminde kullanıla kullanıla, yanlış kullanım doğru kullanım halini almış...*)

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 kwrite belgesi 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()
```

Buradaki “mainloop()” satırı, bir “döngü” yaratarak, kendisinden önce gelen kodlarla belirlenen özelliklere göre bir pencere oluşturulmasını sağlıyor. Bu satırla yaratılan “döngü” sayesinde de oluşturulan pencere (aksi belirtilmedikçe veya kullanıcı pencereyi kapatmadıkça) ekranda hep açık kalabiliyor.

Şimdiye kadar Python'da Tkinter yardımıyla boş bir pencere oluşturmayı öğrendik... Bu heyecan



içinde bu kodları elli defa çalıştırıp elli 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 kwrite belgesi 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 ibaret... Biz burada “pencere” adını tercih ettik.. Tabii ki siz isterseniz emmoğlunuzun adını da verebilirsiniz... Hiç problem değil... Hatta hiç isim vermeseniz de olur. Ama kullanım kolaylığı 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. İstedığınız ismi kullanmakta serbestsiniz. Ancak değişkenin ismi önemli olmasa da, içeriği önemli... "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 faydalıyoruz. (Bu arada "label" kelimesi İngilizce'de "etiket" anlamına geliyor) "Label" ifadesini bu adla aklımıza kazımanız gerekiyor. 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 kazımanız 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 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 Zalim Dünya!")
etiket.pack()
mainloop()
```

Gördüğünüz gibi yine gayet şık, içinde "Elveda Zalim Dünya!" yazan bir pencere elde ettik.

Yukarıda yazdığımız kodları alternatif olarak şu şekilde kısaltmamız da mümkün:

```
#!/usr/bin/env python
#-*-coding:utf-8-*-
from Tkinter import *
etiket = Label(Tk(), text = "GUI dediğin her eve lazım!")
etiket.pack()
mainloop()
```

Burada dikkat ederseniz "Tk()" satırını alıp doğrudan "Label()" ifadesi içine bir parametre olarak gömdük. Bunu yaparken "text" değişkeniyle arasına bir virgül koyduğumuza dikkat edin...

## **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. Mesela şu anda baktığınız pencerenin en tepesinde, "Python (10. Bölüm) – Wiki.Pardus-Linux.Org – Mozilla Firefox" yazıyor. 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.pack()
etiket= Label(text="...ama en azından bir başlığım var!")
mainloop()
```

Gördüğümüz gibi pencereye başlık eklemek son derece kolay... Tek bir satırla işi halledebiliyoruz. Üstelik herhangi bir "paketleme" işlemi yapmamız da gerekmiyor.

Pencerelerle ilgili daha karmaşık özelliklere geçmeden önce isterseniz burada bir durup şimdiye kadar öğrendiklerimizi çok kısa bir şekilde özetleyelim:

1. Öncelikle "from Tkinter import \*" diyerek gerekli modülü yüklememiz gerekiyor.
2. Ardından " Tk()" yardımıyla boş bir pencere oluşturuyoruz. Kullanım kolaylığı açısından " Tk()" ifadesini isimlendirerek bir değişkene atamayı tercih edebilirsiniz. Örneğin; kebab = Tk()
3. 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!")
4. Eğer istersek en baştaki "Tk()" ifadesini doğrudan "Label()" aracının bir parametresi olarak da atayabiliriz. Şöyle ki: oyun = Label(Tk(), text = "Sabit diski silmemiz gerekecek!")
5. 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()
6. Bütün bu yazdığımız kodların işlevli hale gelebilmesi için ise en sona "mainloop()" ifadesini yerleştirmemiz gerekiyor.

7. 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 eşek şakaları 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:

1. .py uzantılı dosyamızı, yani kodlarımızı hazırlarken ilk satıra `#!/usr/bin/env python` ifadesini mutlaka yazıyoruz.
2. Kaydettiğimiz .py uzantılı dosyaya sağ tıklayarak "özellikler"e giriyoruz
3. Burada "izinler" sekmesinde "çalıştırılabilir" ifadesinin yanındaki kutucuğu işaretliyoruz
4. "Tamam" diyerek çıkıyoruz.

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 "seçenek"tir... 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, önalın) 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>

### **bg seçeneği**

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="red", bg="black")
etiket.pack()
mainloop()
```

## Yazı Tipleri (Fonts)

Tkinter bize renklerin yanısıra yazı tipleriyle de oynama imkanı 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 ise şunları kullanabilirsiniz:

Century Schoolbook L

DejaVu Sans

DejaVu Sans

DejaVu Sans Condensed

DejaVu Sans Mono

DejaVu Serif

DejaVu Serif

Dingbats

FreeMono

FreeSans

FreeSerif

Nimbus Mono L

Nimbus Roman No9 L

Nimbus Sans L

Standard Symbols L

URW Bookman L

URW Chancery L

URW Gothic L

URW Palladio L

pardus

Bu listede birden fazla kelimedenden 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](http://www.dil.univ-mrs.fr/~garreta/PythonBBSG/docs/Tkinter_ref.pdf)

Burada 10. ve 11. sayfalarda hoşunuza gidebilecek imleç isimlerini seçebilirsiniz.

## 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 küçük. 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 engellemek ve penceremizi istediğimiz boyutta oluşturabilmemiz için, Tkinter bize gayet kullanışlı bir imkan 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...

Böylelikle Arayüz Tasarımı konusunun "Temel Bilgiler" kısmını bitirmiş oluyoruz. Bir sonraki bölümde Tkinter'de "Pencere Araçlarını" (widgets) kullanmayı öğreneceğiz.

## **Grafik Arayüz Tasarımı // Pencere Araçları (Widgets)**

### **Giriş**

Herhangi bir pencere üzerinde bulunan düğmeler, etiketler, sağa-sola, yukarı-aşağı kayan çubuklar, kutular, 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:

### **"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! Belleği vurmamı ister misiniz?")
```

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 ("button": "düğme")

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 *
def olustur():
    dosya = open("deneme.txt", "w")
pencere = Tk()
dugme = Button(text = "oluştur", command=olustur)
dugme.pack()
mainloop()
```

Gördüğünüz gibi, Tkinter modülünü çağırdıktan sonra bir fonksiyon oluşturduk. Tkinter dışından bir komut çalıştırmak istediğimizde bu şekilde bir fonksiyon tanımlamamız gerekir.

Daha sonra her zaman yaptığımız gibi, "Tk()" komutuyla penceremizi oluşturduk. Button() yardımıyla da 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 oluştursanız, "deneme.txt" adlı dosya düğmeye henüz basmadan oluşacaktır...

İsterseniz bu arayüze bir de "çıkış" düğmesi ekleyebiliriz:

```
#!/usr/bin/env python
#-*-coding:utf-8-*-
from Tkinter import *
def olustur():
    dosya = open("deneme.txt", "w")
pencere = Tk()
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 *
def olustur():
    dosya = open("deneme.txt", "w")
pencere = Tk()
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.

## "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 imkanı 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 *
def sil():
    giris.delete(0,END)
pencere = Tk()
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 *
```

```
def olustur():
    dosya = open("deneme.txt","w")
    metin = giris.get()
    dosya.write(metin)

pencere = Tk()
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

def bas():
    a = random.randint(1,100)
    giris.delete(0,END)
    giris.insert(0, a)

pencere = Tk()
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. 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.

### "Checkbutton" Pencere Aracı

"Checkbutton" denen şey, bildiğimiz "onay kutusu"... 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 isteriz:

```
#!/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. Eğer bu konuda ısrarcıysak bu noktada farklı bir bilgiye ihtiyacımız var.

Bildiğiniz ve gördüğünüz gibi, herhangi bir pencere aracı oluşturduktan sonra "pack" ifadesi yardımıyla pencere aracımızı paketliyorduk. Aslında bu "pack" ifadesi Tkinter'de "Geometri Yöneticisi" denen araçlardan bir tanesidir. Geometri Yöneticileri bir pencere aracının nasıl görüneceğini, nerede duracağını, yani kısaca o aracın şeklini şemalini belirlerler. Mesela "pack" adlı Pencere Yöneticisine, yukarıda gördüğümüz gibi bir "side=LEFT" parametresi eklediğimizde ilgili pencere aracı sola yaslanıyor... Tkinter'de üç adet Geometri Yöneticisi bulunur:

- + Pack Geometri Yöneticisi
- + Grid Geometri Yöneticisi
- + Place Geometri Yöneticisi

Bu Geometri Yöneticilerinin 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 uzmandır. 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.

Dediğimiz gibi, pencere araçlarını istediğimiz şekilde dizmek için en uygun geometri yöneticisi "place" adlı olan... Şimdi bununla ilgili bir örnek görelim.. Mesela bir tane onay kutusu oluşturup bunu en sola yaslayalım:

```
from Tkinter import *
pencere = Tk()
onay = Checkbutton(text="Pardus")
onay.place(relx = 0.0, rely = 0.1)
mainloop()
```

Gördüğünüz gibi, bu place adlı yönetici de tıpkı pack gibi çalışıyor. Burada dikkatimizi çeken "relx" ve "rely" seçenekleri onay kutusunun koordinatlarını gösteriyor. 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="Pardus")
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 "Pardus"u "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="Pardus")
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()
```

Pardus'u 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 mesela 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="Pardus", 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 "seçilmemiş" haldedir. 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="Pardus", 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 "Pardus" olsun.
d.set("Pardus")
# 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)

# Aşağıda "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)
# Aşağıda "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. Anladığımız gibi, eğer aksi belirtilmemişse, pencere araçları otomatikman 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" seçeneğini 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 imkanı 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 tamamımını 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 yaratıcılığınızı kullanarak çok daha çekici görünüm ve renkler elde edebilirsiniz...

Şimdi liste kutumuza bazı öğeler ekleyelim. Bunun için şu basit ifadeyi kullanacağız:

```
liste.insert(END,"öge 1")
```

Tabii ki 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,"öge 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:

```
linux_dagitimlari = ["Pardus", "Debian", "Ubuntu", "PclinuxOS", "TruvaLinux", "Gelecek Linux"]  
for i in linux_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()  
linux_dagitimlari = ["Pardus", "Debian", "Ubuntu", "PclinuxOS", "TruvaLinux", "Gelecek Linux"]  
for i in linux_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()
linux_dagitimleri = ["Pardus", "Debian", "Ubuntu", "PclinuxOS", "TruvaLinux", "Gelecek Linux"]
for i in linux_dagitimleri:
    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()
linux_dagitimlari = ["Pardus", "Debian", "Ubuntu", "PclinuxOS", "TruvaLinux", "Gelecek Linux"]
for i in linux_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 "linux\_dagitimlari" adında bir liste oluşturduk ve tek tek öğelerini yazdık... Hemen alt satırda bir "for döngüsü" yardımıyla linux\_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. İsmi 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 kutusu"na ekleniyor... Ayrıca ilk veri girişinin ardından "Entry" aracı içindeki alan tamamen boşaltılıyor ki kullanıcı rahatlıkla ikinci veriyi girebilsin... Çok hoş değil mi?!

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" koşulu 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 *
```

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

```
#-*-coding=utf-8-*-
```

```
from Tkinter import *
```

```
pencere = Tk()
```

```
liste = Listbox(bg="white")
```

```
liste.pack()
```

```
linux_dagitimleri = ["Pardus", "Debian", "Ubuntu", "PclinuxOS", "TruvaLinux", "Gelecek Linux"]
```

```
for i in linux_dagitimleri:
```

```
    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"... ikinci "menu" ifadesi ise yukarıda bizim "Menu" aracına kendi verdiğimiz isim... Bu isim herhangi bir kelime olabilirdi... Yani en başta menünün adını "kebab" olarak belirleseydik, burada "menu=kebab" 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 "menu"nın 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 gördüğümüz "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 epeyce 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 aracı"na bağlıyoruz.

Üçüncü satırda "inen menü"yü 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 şöyle bir bölüm ekliyoruz kodlarımız arasına:

```
#!/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 yerleştirmedığımız 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. (İngilizce'de "END" kelimesi "SON" anlamına geliyor). 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. Mesela 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
```

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

```
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
```

```
#!/usr/bin/python3
```

```
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.