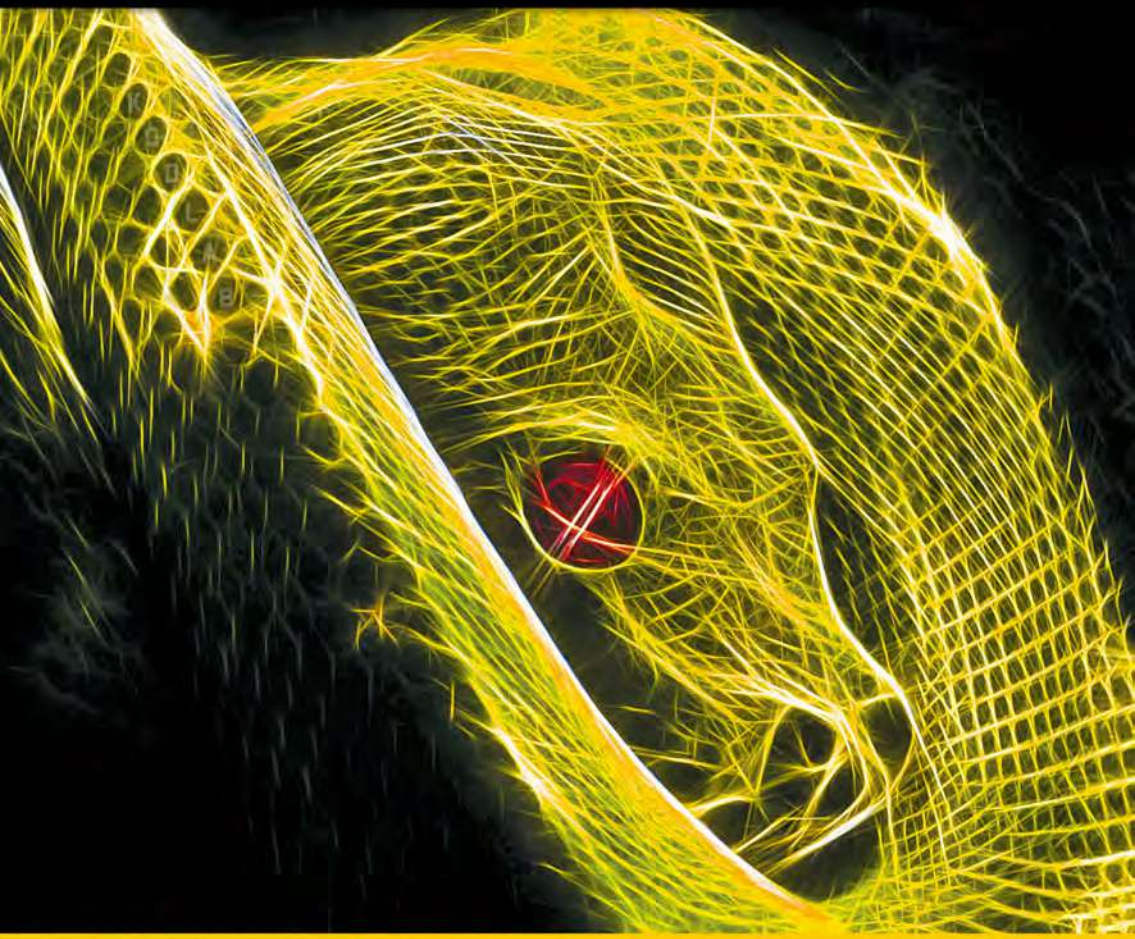
Fırat ÖZGÜL

Uzmanından!



HER YÖNÜYLE

PYTHON 3X

KODLAB®
www.kodlab.com

HER YÖNÜYLE
PYTHON

Fırat Özgül

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

HER YÖNÜYLE PYTHON

FIRAT ÖZGÜL

ISBN 978-605-2118-50-4
Yayıncılık Sertifika No: 13206

1. Baskı: Eylül 2020

Genel Yayın Yönetmeni
Hüseyin Üstünel

Grafik Tasarım
Mustafa Said Şahin

Editör
Gizem Aksan

Satış ve Pazarlama
Can Üstünel
Y. Emre Üstünel

Baskı: İnkılâp Kitabevi Yayın San. Tic. A.Ş. Sertifika No: 44066

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

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

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

Fırat Özgül

Yazarımız, 1980 yılında Adana'da doğmuştur. Boğaziçi Üniversitesi Mütercim Tercümanlık bölümünü bitiren yazar, özellikle özgür yazılım üzerine yaptığı çalışmalarla ve Python programlama dili için bir Türkçe kaynak oluşturma projesi olan istihza.com'la tanınıyor.

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

ÖN SÖZ

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

Son yıllarda ülkemizde de artık iyiden iyiye ilgi uyandırmaya başlayan bu dil, üniversitelerin müfredatında kendine yer bulabilmektedir. Her ne kadar ülkemizde henüz yeterli ilgiyi görmese de bugün; Google'dan NASA'ya, YouTube'tan Hewlett Packard'a kadar pek çok uluslararası kurum ve şirkette Python programlama dilinden yararlanılmaktadır.

Elinizdeki kitap, Python programlama diline merak duyan okurlara bu dilin temellerini sağlam ve hızlı bir şekilde öğretmeyi amaçlamaktadır. Bu kitaptan yararlanabilmek için önceden bir programlama dili biliyor olmanız gerekmez. Bu kitap, okurun programlama konusunda hiçbir bilgisi olmadığı varsayılarak ve herkesin anlayabileceği bir üslupla hazırlanmıştır. **Python'ın 3.8.3 sürümünü** temel alan bu kitap üç bölümden oluşmaktadır.

Temel Konular bölümünde, Python programlama dilinin özünü oluşturan; karakter dizileri, sayılar, listeler, demetler, sözlükler, kümeler, fonksiyonlar, dosya işlemleri ve hata yakalama gibi temel düzeydeki konular işlenmektedir. Bu bölümde ayrıca veri tabanı programlama, düzenli ifadeler ve nesne tabanlı programlama gibi orta-ileri düzey konulara da yer verilmiştir. Bu bölümü bitirdiğinizde Python'la konsol uygulamaları yazabilecek düzeye geleceksiniz.

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

Özel Konular bölümü ise Python programlama diline ilişkin bir konunun özel bir yönünü inceleyen makalelere ayrılmıştır. Bu bölümde math modülünün yanı sıra, Python'ın önbellekleme mekanizmasına ilişkin bir inceleme yazısı ve ayrıca ikili dosyalar ile Python'daki bazı önemli fonksiyonları ele alan birer makale yer almaktadır.

Fırat ÖZGÜL

TEŞEKKÜR

Türkiye'deki tüm özgür yazılım topluluklarına ve "Math Modülü" adlı makaleye katkılarından ötürü Kürşat Örsel'e teşekkürlerimi sunarım.

İÇİNDEKİLER

KISIM I: TEMEL KONULAR **1**

1 PYTHON HAKKINDA **3**

2 PYTHON NASIL KURULUR? **5**
Microsoft Windows 5
GNU/Linux 7

3 PYTHON NASIL ÇALIŞTIRILIR? **9**
Windows'ta Python'ı Çalıştırmak 9
GNU/Linux'ta Python'ı Çalıştırmak 12

4 BAZI TEMEL KAVRAMLAR **17**
print() Fonksiyonu 19
Karakter Dizileri 20
Kaçış Dizileri 21
Argümanlar 23
sep Argümanı 25
Argümanların Dağıtılması 27
end Argümanı 29
Yeni Satır Kaçış Dizisi 30

5 PYTHON'DA SAYILAR VE ARİTMETİK İŞLEMLER **35**

6 DEĞİŞKENLER **43**
f'li Karakter Dizileri 45
Çoklu Değişken Tanımlama 48
Değişkenlerin Dağıtılması 51
Dilimleme 54

7 METİN DÜZENLEYİCİ KULLANARAK PYTHON PROGRAMLAMA **61**
GNU/Linux Sistemi 61
Windows Sistemi 68

8 STANDART KÜTÜPHANEYE GİRİŞ **71**

9 KULLANICIYLA İLETİŞİM: VERİ ALIŞ-VERİŞİ **83**

10 DÖNÜŞTÜRME İŞLEMLERİ **95**

11 KAÇIŞ DİZİLERİ **107**

12 KOŞULA BAĞLI DURUMLAR **115**
If 117
else 121
elif 122

13 PYTHON'DA GİRİNTİLEME SİSTEMİ **131**

14 DÖNGÜLER **135**
whıle Döngüsü 136
for Döngüsü 145
range() Fonksiyonu 150
len() Fonksiyonu 156
break Deyimi 158
continue Deyimi 159
in işleci 161

15 ÖRNEK UYGULAMALAR **163**
Birkaç Pratik İpucu 163
Sayı Tahmin Oyunu 164
Kelime Analizi 168
Not Ortalaması Hesaplayan Program 174
Veri Tipleri 177
help() Fonksiyonu 180

16 LİSTELER **183**
Listelerin Metotları 188
append() 190
extend() 197
ınsert() 198
remove() 199
pop() 200
index() 201
sort() 201
reverse() 203
count() 203
Listelerin Dilimlenmesi 204
İç İçe Geçmiş Listeler 209
list() Fonksiyonu 210

17 DEMETLER **213**
Demetlerin Metotları 217
count() 218
index() 218
tuple() Fonksiyonu 219

18 SÖZLÜKLER **221**
Sözlüklerin Metotları 224
keys() 224
values() 224
items() 225
clear() 227
get() 227

19 KÜMELER **231**
Kümelerin Metotları 235
add() 235
remove() 236
discard() 238
clear() 238
pop() 239
update() 240
copy() 240
union() 241
difference() 241
symmetric_difference() 241
intersection() 242
difference_update() 243
symmetric_difference_update() 243
intersection_update() 244
isdisjoint() 244
issubset() 245
issuperset() 246
Küme İşleçleri 246
Birleşim Kümesi İşleci 246
Fark Kümesi İşleci 247
Simetrik Fark Kümesi İşleci 247
Kesişim Kümesi İşleci 247
Alt küme Sorgulama İşleci 247
Üst Küme Sorgulama İşleci 248
Öz Alt Küme Sorgulama İşleci 248
Öz Üst Küme Sorgulama İşleci 248

20 FONKSİYONLAR **249**
Fonksiyonları Tanımlamak 251
Fonksiyonlarda Parametre Kullanımı 256
Sıralı Parametreler 264
İsimli Parametreler 268
Zorunlu İsimli Parametreler 270
Zorunlu Sıralı Parametreler 272
Sınırsız Sayıda Sıralı Parametre 273
Sınırsız Sayıda İsimli Parametre 274
Gömülü Fonksiyonlar 275
Global Deyimi 276
return Deyimi 280
pass Deyimi 285

21 MODÜLLER **287**
Modüllerin İçe Aktarılması 288
import modül_adı 288

import modül_adı as başka_bir_ad 289
from modül_adı import * 289
From modül_adı import metot_veya_fonksiyon 292
from modül_adı import metot_veya_fonksiyon as başka_bir_ad 293
Standart Kütüphane Modülleri 293
Üçüncü Şahıs Modülleri 299
Kullanıcı Tanımlı Modüller 301
If __name__ == '__main__' 302
sys.path 312

22 DOSYA İŞLEMLERİ 315
Dosya Açmak 315
Dosyaya Yazmak 321
Dosyaların Metotları 323
Dosyayı Okumak 329
Dosya Silmek 333
Dosyaya Rastgele Satır Eklemek 334
Dosyadan Rastgele Satır Silmek 337
Dosyaları Otomatik Kapatma 339

23 HATA YAKALAMA 341
try... except... 343
pass Deyimi 347

24 KARAKTER DİZİLERİNİN METOTLARI 351
capitalize() Metodu 353
upper() Metodu 354
lower() Metodu 354
casefold() Metodu 354
swapcase() Metodu 355
title() Metodu 355
center() Metodu 356
ljust() Metodu 356
rjust() Metodu 357
zfill() Metodu 357
replace() Metodu 358
startswith() Metodu 359
endswith() Metodu 360
count() Metodu 361
isalpha() metodu 361
isdigit() Metodu 362
isalnum() metodu 362
islower() Metodu 363
isupper() Metodu 364
istitle() Metodu 364
isspace() Metodu 365
isidentifier() Metodu 365

expandtabs() Metodu 366
find() Metodu 366
rfind() Metodu 368
index() Metodu 368
rindex() Metodu 369
join() Metodu 370
translate() ve maketrans() Metotları 371
partition() Metodu 374
rpartition() Metodu 374
strip() Metodu 375
rstrip() Metodu 377
lstrip() Metodu 377
splitlines() Metodu 378
split() Metodu 378
rsplit() Metodu 380
format() Metodu 381
encode() Metodu 382
Metotlarda Türkçe Karakter Sorunu 387

25 DÜZENLİ İFADELER (REGULAR EXPRESSIONS) 393
Düzenli İfadelerin Metotları 394
match() Metodu 395
search() Metodu 400
findall() Metodu 403
Metakarakterler 405
[ ] (Köşeli Parantez) 408
. (Nokta) 411
* (Yıldız) 413
+ (Artı) 416
? (Soru İşareti) 418
{ } (Küme Parantezi) 420
^ (Şapka) 421
$ (Dolar) 425
\ (Ters Bölü) 424
| (Dik Çizgi) 427
( ) (Parantez) 428
Eşleşme Nesnelerinin Metotları 428
group() Metodu 428
groups() Metodu 431
Özel Diziler 432
Ondalık Sayıların Yerini Tutan Özel Dizi: \d 432
Alfanümerik Karakterlerin Yerini Tutan Özel Dizi: \w 433
Düzenli İfadelerin Derlenmesi 435
compile() Metodu 435
compile() ile Derleme Seçenekleri 436
re.IGNORECASE veya re.I 436
re.DOTALL veya re.S 437

Düzenli İfadelerle Metin/Karakter Dizisi Değiştirme İşlemleri 438
sub() Metodu 438
subn() Metodu 444
Sonuç 444

26 SQLITE İLE VERİ TABANI PROGRAMLAMA 445
Giriş 445
Neden Sqlite? 445
Sqlite'ın Yapısı 446
Örnek Veri Tabanı 447
Yeni Bir Veri Tabanı Oluşturmak 447
Varolan Bir Veri tabanıyla Bağlantı Kurmak 450
İmleç Oluşturma 450
Tablo Oluşturma 451
Şartlı Tablo Oluşturma 454
Tabloya Veri Girme 455
Verilerin Veri Tabanına İşlenmesi 457
Veri Tabanının Kapatılması 459
Parametreli Sorgular 460
Tablodaki Verileri Seçmek 461
Seçilen Verileri Almak 463
fetchall() Metodu 463
fetchone() Metodu 468
fetchmany() Metodu 469
Veri Süzme 470
Veri Tabanı Güvenliği 472

27 NESNE TABANLI PROGRAMLAMA 481
Neden Nesne Tabanlı Programlama? 481
Sınıflar 483
Nesneler 485
Sınıfların Kullanımı 490
Self 494
__init__ Metodu 496
Basit Bir Oyun 496

KISIM II: ÖZEL KONULAR 513

1 MATH MODÜLÜ 515
Üslü İfadeler (pow) 516
PI sayısı (pı) 516
Karekök (sqrt) 517
Euler Sabiti (e) 518
exp() Fonksiyonu 518
Logaritma (log) 519
log10() Fonksiyonu 519

degress() Fonksiyonu 519
radians() Fonksiyonu 520
Kosinüs (cos) 520
Sinüs (sin) 521
Tanjant (tan) 522

2 PYTHON'DA ID() FONKSİYONU, IS İŞLECİ VE ÖNBELLEKLEME MEKANİZMASI 525

3 İKİLİ DOSYALAR 533
İkili Dosyalarla Örnekler 535
PDF Dosyalarından Veri Alma 535
Resim Dosyalarının Türünü Tespit Etme 539
JPEG 540
PNG 544
GIF 545
TIFF 546
BMP 547

4 ÖNEMLİ GÖMÜLÜ FONKSİYONLAR 549
sorted() 549
min() 556
max() 557
sum() 559
enumerate() 560
zip() 561

KISIM III: ARAYÜZ 563

1 GRAFİK ARAYÜZ TASARIMI TEMEL BİLGİLER 565
Pencere Oluşturmak 566
Pencere Başlığı 573
Renkler 575
fg Seçeneği 575
bg Seçeneği 577
Yazı Tipleri (Fonts) 578
Metin Biçimlendirme 580
İmleçler 580
Pencere Boyutu 581
Tekrar 583

2 PENCERE ARAÇLARI (1. BÖLÜM) 587
Label Pencere Aracı 587
Button Pencere Aracı 588
Entry Pencere Aracı 599
Frame Pencere Aracı 603

3 GEOMETRİ YÖNETİCİLERİ **607**
Pack Geometri Yöneticisi 611
expand Seçeneği 612
fill Seçeneği 613
side Seçeneği 614
Grid Geometri Yöneticisi 621
row Seçeneği 622
column Seçeneği 623
Place Geometri Yöneticisi 627

4 PENCERE ARAÇLARI (2.BÖLÜM) **629**
Checkbutton Pencere Aracı 629
Toplevel Pencere Aracı 637
Listbox Pencere Aracı 640
Menü Pencere Aracı 648
Text Pencere Aracı 653
Scrollbar Pencere Aracı 659

5 TKINTER UYGULAMALARINI GÜZELLEŞTİRMEK **663**
Tkinter Programlarının Renk Şemasını Değiştirmek 663
Pencere Araçlarına Simge Eklemek 667
Pencere Araçlarına İpucu Metni (Tooltıp) Eklemek 670

6 NASIL YAPILIR? **675**
Tkınter'de Fare ve Klavye Hareketleri 675
Listbox Ögelerine Görev Atamak 689
Pencereleri Başlıksız Hâle Getirmek 691
Pencere / Ekran Koordinatları ve Boyutları 694
winfo_height() 696
winfo_width() 698
winfo_rootx() 699
winfo_rooty() 700
winfo_screenheight() 701
winfo_screenwidth() 701
Programı Tam Ekran Olarak Çalıştırmak 702
Ekranı Ortalamak 703
Pencereleri Her Zaman En Üstte Tutmak 704

7 STANDART BİLGİ PENCERELERİ **709**
Hata Mesajı Gösteren Pencere 710
Bilgi Mesajı Gösteren Pencere 718
Uyarı Mesajı Gösteren Pencere 721

KISIM I

TEMEL KONULAR

PYTHON HAKKINDA

1

Python, Guido Van Rossum adlı Hollandalı bir programcı tarafından 1990 yılında geliştirilmeye başlanmış bir programlama dilidir.

TIOBE'nin Temmuz 2020 verilerine göre C ve Java programlama dillerinin ardından **en gözde üçüncü dil konumunda bulunan Python**, son yılların revaçtaki çalışma alanlarından **veri bilimi** ve **yapay zekâ** konularında ise en tercih edilen programlama dilidir.

Özellikle programlamaya yeni başlıyorsanız; C, C++ ve Java gibi dillere kıyasla **söz dizimi daha temiz** ve **daha okunaklı** olduğu için Python programlama dilini çok daha çabuk ve kolay bir şekilde öğrenebilirsiniz.

Python, uluslararası şirketler tarafından da her zaman aranan ve ilgi gören bir dildir. Mesela Python'ın baş geliştiricisi Guido Van Rossum'un 2005 ile 2012 yılları arasında Google'da çalıştığını, 2012 yılının sonlarına doğru ise Dropbox şirketine geçtiğini söylersek bu programlama dilinin önemi ve geçerliliği herhâlde daha belirgin bir şekilde ortaya çıkacaktır.

Bu arada, her ne kadar Python programlama dili ile ilgili çoğu görsel malzemenin üzerinde bir *yılan resmi* görsek de Python kelimesi aslında çoğu kişinin zannettiğinin aksine **piton yılanı anlamına gelmiyor**. Python Programlama Dili, ismini **Monty Python** adlı altı kişilik bir İngiliz komedi grubunun **Monty Python's Flying Circus** adlı gösterisinden alıyor.

2 PYTHON NASIL KURULUR?

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

Python, Windows ve GNU/Linux işletim sistemlerine kurulma açısından farklılıklar gösterir. Biz bu bölümde Python'ın hem **GNU/Linux**'a hem de **Windows**'a nasıl kurulacağını ayrı ayrı inceleyeceğiz.

Dilerseniz ilk önce Windows'tan başlayalım.

Microsoft Windows

Python programlama dilini Windows'a kurabilmek için öncelikle *https://www.python.org/downloads/* adresini ziyaret ediyoruz. Burada, üzerinde "**Download Python 3.8.3**" yazan sarı düğmeye tıklayarak **.exe** dosyasını bilgisayarımıza indiriyoruz.



Eğer Python'ın belirli bir sürümünü arıyorsanız https://www.python.org/ftp/python/ adresinden Python'ın tüm sürüm ve mimarilerine ulaşabilirsiniz.

Python'ı indirdikten sonra **.exe** dosyasına **çift tıklayarak** kurulumu başlatıyoruz. İlk kurulum ekranında karşımıza şöyle bir pencere gelecek:



Şekil 2-1: Kurulum ekranı

Bu ekranda pencerenin alt tarafında iki tane kutucuk görüyoruz:

```
[X] Install launcher for all users (recommended)
[ ] Add Python 3.8 to PATH
```

Birinci kutucuk; Python'ı çalıştırmamızı sağlayacak programı, bilgisayarımızdaki mevcut tüm kullanıcılar için kurar. Bu kutucuk öntanımlı olarak zaten **seçili** durumdadır. Biz de bunu değiştirmeyelim.

Yukarıdaki **ikinci kutucuk** ise; Python programlama dilinin kurulu olduğu dizinin Windows işletim sistemi içindeki "**PATH**" adlı özel bir listeye eklenmesini sağlıyor. Eğer bu kutucuğu işaretlemezseniz ileride Python'la ilişkili bazı programları çalıştırmakta zorluk çekebilirsiniz. O yüzden ben size bu kutucuğu da **işaretlemenizi** tavsiye ederim.

Ayrıca bu kutucuk, Python programlama dilini bilgisayarımıza kurduktan sonra programlarımızı **hangi komutla** çalıştırabileceğimizi de belirleyecek. Eğer bu kutucuğu işaretlemezseniz Python'ı yalnızca **py** komutuyla çalıştırabileceksiniz. Eğer işaretlerseniz Python programlarınızı çalıştırmak için hem **py** hem de **python** komutunu kullanabileceksiniz. *Bu önemli bir fark değil*. Ama siz yine de bu kutucuğu **işaretlemeyi unutmayın**.

Eğer Python'ın bilgisayarınızda nereye ve nasıl kurulacağı gibi ayrıntıları düzenlemek veya değiştirmek istiyorsanız "**Customize installation**" düğmesine basarak tercihlerinizi yapabilirsiniz. Ama ben size, herhangi bir ayarı değiştirmeden **standart bir kurulum** yapmanızı tavsiye ederim. Özellikle Python'a yeni başlıyorsanız hiç başka bir yola sapmadan "**Install Now**" düğmesine basarak kurulumu tamamlayabilirsiniz.

Gelelim GNU/Linux kullanıcılarının Python'ı nereden bulacağına...

GNU/Linux

Python hemen hemen tüm **GNU/Linux** dağıtımlarında **kurulu** olarak geliyor. Mesela Ubuntu'da Python'ın kurulu olduğunu biliyoruz. O yüzden Ubuntu kullanıyorsanız Python'ı kurmanıza gerek yok.

Ama eğer herhangi bir şekilde Python'ı kurmanız gerekirse öncelikle şu adresi ziyaret edin: *https://www.python.org/downloads/*

Bu adreste, üzerinde "**Download Python 3.8.3**" yazan sarı düğmeye tıklayarak kaynak kodları içeren dosyayı bilgisayarınıza indirin.



Eğer Python'ın belirli bir sürümünü arıyorsanız https://www.python.org/ftp/python/ adresinde Python'ın tüm sürüm ve mimarilerini bulabilirsiniz.

Python'ın kaynak kodlarını bilgisayarınıza indirdikten sonra artık Python'ı **kaynaktan** derlemeye hazırsınız. Ancak Python'ı kaynaktan derlerken yeni kuracağınız Python sürümünün sistemde hâlihazırda bulunan Python sürümünü **silmemesine dikkat etmeniz gerekiyor**. Çünkü sistemdeki önemli bazı programlar bu eski Python sürümüne ihtiyaç duyuyor olabilir. Eğer sistemdeki bu Python sürümünü **yanlışlıkla silerseniz** bu sürüme ihtiyaç duyan başka programlar da **artık çalışmayacaktır**.

Daha sonra, bilgisayarınıza indirdiğiniz bu sıkıştırılmış dosyayı açın ve açılan dosyanın içine girip orada **sırasıyla aşağıdaki komutları verin**:

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

Böylelikle Python'ın **farklı bir sürümünü** sisteminize kurmuş oldunuz. Burada **`make install`** yerine **`make altinstall`** komutunu kullandığımıza özellikle dikkat edin. **`make altinstall`** komutu, Python kurulurken klasör ve dosyalara **sürüm numarasının da eklenmesini** sağlar. Böylece yeni kurduğunuz Python, sistemdeki eski Python sürümünü silip üzerine yazmamış olur ve iki farklı sürüm yan yana bulunabilir.

Bu noktada bir uyarı yapmadan geçmeyelim. Python, özellikle bazı GNU/Linux dağıtımlarında pek çok sistem aracıyla **sıkı sıkıya bağlantılıdır**. Yani Python, kullandığınız dağıtımın belkemiği durumunda olabilir. Bu yüzden Python'ı **kaynaktan kurmak bazı riskler taşıyabilir**.

Eğer yukarıda anlatıldığı şekilde sisteminize kaynaktan Python kuracaksanız karşı karşıya olduğunuz risklerin farkında olmalısınız. Ayrıca GNU/Linux üzerine kaynaktan program kurmak konusunda **tecrübeli değilseniz** kaynaktan kuracağınız programların **bağımlılıklarının** neler olduğunu, bunları nasıl kuracağınızı, programı kurduktan sonra nasıl çalıştıracağınızı bilmiyorsanız ve eğer yukarıdaki açıklamalar size kafa karıştırıcı geliyorsa kesinlikle **dağıtımınızla birlikte gelen** Python sürümünü kullanmalısınız.

Python, sürümlerini başa baş takip ettiği için ben size **Ubuntu GNU/Linux**'u denemenizi önerebilirim. Ubuntu'nun depolarında Python'ın en yeni sürümlerini rahatlıkla bulabilirsiniz. Ubuntu'nun resmî sitesine *https://www.ubuntu.com* adresinden, yerel Türkiye sitesine ise *https://www.ubuntu-tr.net* adresinden ulaşabilirsiniz.

PYTHON NASIL ÇALIŞTIRILIR?

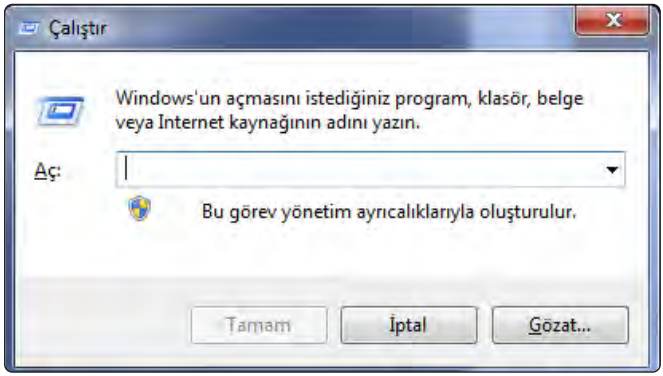
3

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

WINDOWS'TA PYTHON'I ÇALIŞTIRMAK

Python'ı geçtiğimiz bölümde verdiğimiz indirme adresinden indirip bilgisayarlarına kurmuş olan Windows kullanıcıları, Python **komut satırına** şu şekilde ulaşabilir:

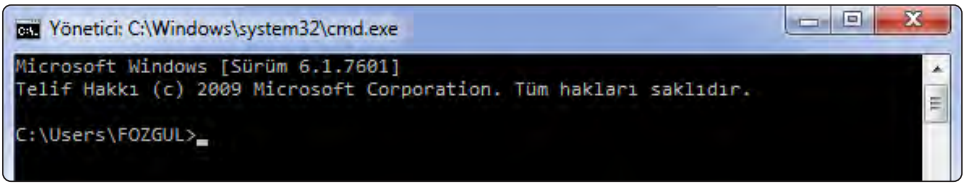
Öncelikle klavyedeki **Windows logolu tuşa** ve **R** harfine birlikte basarak "**çalıştır**" penceresini açın:



Şekil 3-1: Çalıştırma penceresi

Bu pencere üzerindeki "**Aç**" etiketli kutucuğa **cmd** yazıp **Tamam** düğmesine basın.

Karşınıza şuna **benzer bir ekran gelmeli**:



Şekil 3-2: Windows komut satırı

Bu ekrana "**Windows komut satırı**" adı veriyoruz.

Şimdi, bu ekranda şu komutu verip **ENTER** düğmesine basın:

```
py
```

Şöyle bir **çıktı** ile karşılaşacaksınız:

```
Python 3.8.3 (tags/v3.8.1:1b293b6, 14/07/2020, 12:24:55)
[MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Bu ekrana da "**etkileşimli kabuk**" diyoruz. Bu ortamdan **çıkmak için** şu komutu verip **ENTER** düğmesine basabilirsiniz:

```
quit()
```

Bu komutun bizi etkileşimli kabuktan çıkarıp Windows komut satırına attığına dikkat ediyoruz. Ayrıca etkileşimli kabukla Windows komut satırının **birbirinden farklı iki ortam** olduğunu da aklımızdan çıkarmıyoruz. Unutmayın, Windows komut satırında **Windows komutlarını**, etkileşimli kabukta ise **Python komutlarını** çalıştırıyoruz.

Etkileşimli kabuğu terk etmek için ayrıca **CTRL+Z** tuş kombinasyonunu da kullanabilirsiniz.

Eğer Python'ı kurarken daha önce bahsettiğimiz şekilde "**Add Python 3.8 to PATH**" kutucuğunu işaretlemişseniz Python programlama dilini **py** komutunun yanı sıra **python** komutuyla da başlatabilirsiniz:

```
python
```

Bu komut da size yukarıdaki **çıktıyı** verecek:

```
Python 3.8.3 (tags/v3.8.1:1b293b6, 14/07/2020, 12:24:55)
[MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Eğer **python** komutu verdiğinizde yukarıdaki **çıktı** yerine şöyle bir **hata mesajı** alıyorsanız:

```
'python' iç ya da dış komut, çalıştırılabilir
program ya da toplu iş dosyası olarak tanınmıyor.
```

... ya '**python**' kelimesini **yanlış yazıyorsunuzdur** ya da Python'ı kurarken "**Add Python 3.8 to PATH"** kutucuğunu işaretlememişsinizdir. Bu durumda Python'ı **python** yerine **py** komutuyla çalıştırmaya devam edebilirsiniz.

Ama eğer Python'ı aynı zamanda **python** komutuyla da çalıştırabilmek istiyorsanız daha önce *https://www.python.org/downloads* adresinden indirmiş olduğunuz **.exe** uzantılı Python kurulum programını **yeniden çalıştırın**.

Daha sonra **'Modify' > 'Next'** yolunu takip edin ve oradaki kutucuklar arasından "**Add Python to environment variables"** başlıklı olanı işaretleyin.

Ardından da "**Install**" düğmesine basarak Python'ı **tekrar kurun**.

Python'ı bu şekilde bir daha kurduktan sonra artık hem **py** hem de **python** komutunu vererek **Python programlama dilini başlatabilirsiniz**.

Bu iki komuttan herhangi birini verdiğimizde karşımıza çıkan ekrana baktığımızda kullandığımız Python sürümünün **3.8.3** olduğunu görüyoruz.

Ayrıca bu ekrandaki **>>>** işareti, o anda etkileşimli kabuk ortamında olduğumuzu ve Python'ın da bizden komut almaya hazır olduğunu gösteriyor. İşte komutlarımızı bu işaretten hemen sonra, **boşluk bırakmadan** yazacağız. Bunun dışında, istersek Python kodlarını bir metin dosyasına da kaydedebilir, bu kaydettiğimiz metin dosyasını **komut satırından** çalıştırabiliriz. Bu işlemin nasıl yapılacağını daha sonra konuşacağız.

Gelelim GNU/Linux kullanıcılarına...

GNU/Linux'ta Python'ı Çalıştırmak

Eğer GNU/Linux işletim sistemi üzerinde **KDE masaüstü** ortamını kullanıyorsak Python'ı çalıştırmak için **ALT+F2** tuşlarına basıp çıkan ekranda şu komutu vererek bir **konsol ekranı** açıyoruz:

```
konsole
```

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

```
gnome-terminal
```

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

Hatta bazı GNU/Linux dağıtımlarında masaüstüne **fare ile sağ tıklandığında** açılan menüde komut satırına ulaşmanızı sağlayacak bir **seçenek** de olabilir.

Kullandığınız dağıtıma uygun olan yönteme göre komut satırına ulaştıktan sonra:

```
python3
```

yazıp **ENTER** tuşuna basarak **Python programlama dilini başlatıyoruz**.

Karşımıza şuna benzer bir ekran gelmeli:

```
Python 3.8.3 (default, 14/07/2020, 12:24:55)
[GCC 9.2.1 20191008] on linux
Type “help”, “copyright”, “credits” or “license” for more
information.
>>>
```

Bu ekrana “**etkileşimli kabuk**” diyoruz. Bu ortamdan çıkmak için şu komutu verip **ENTER** düğmesine basın:

```
quit()
```

Etkileşimli kabuğu terk etmek için ayrıca **CTRL+Z** tuş kombinasyonunu da kullanabilirsiniz.

Bu komutun bizi, Python'ın etkileşimli kabuğundan çıkarıp Linux'un sistem komut satırına attığına dikkat ediyoruz. Ayrıca etkileşimli kabukla sistem komut satırının birbirinden farklı iki ortam olduğunu da aklımızdan çıkarmıyoruz. **Unutmayın**, sistem komut satırında **Linux komutlarını**, etkileşimli kabukta ise **Python komutlarını** çalıştırıyoruz.

Python'ın etkileşimli kabuğuna ulaşmak için kullandığımız komutun **`python`** değil, **`python3`** olduğuna özellikle **dikkat etmenizi istiyorum**. Pek çok GNU/Linux dağıtımında Python'ın **iki farklı sürümü** bir arada bulunur. GNU/Linux dağıtımlarında **`python`** (*veya python2*) komutu verdiğinizde genellikle Python'ın şu anda tedavülden kalkmış olan eski **2.x** sürümü, **`python3`** komutu verdiğinizde ise bizim bu kitapta anlatacağımız **yeni 3.x** sürümü çalışır.

Bazı dağıtımlarda ise Python'ın sadece **eski** veya sadece **yeni** sürümü kuruludur. Dolayısıyla kullandığınız GNU/Linux dağıtımında **`python`** ve **`python3`** komutlarının hangi **Python** sürümünü çalıştırıp çalıştırmadığı konusunda uyanık olmalısınız.

Biz bu kitapta, Python'ın yeni (*3.x*) sürümünü anlatacağımız için sisteminizde hangi Python sürümününün veya sürümlerinin kurulu olduğunu veya hangi komutun **hangi sürümü başlattığını doğru bir şekilde tespit etmeniz büyük önem** taşıyor.

Sisteminizdeki mevcut Python sürümlerini tespit etmek için komut satırında şu komutu vererek **sürüm kontrolü** yapabilirsiniz:

```
python3 --version
```



Bu komutu Python'ın etkileşimli kabuğunda değil, sistem komut satırında verdiğimize dikkat edin. Eğer bu komutu etkileşimli kabukta vermeye çalışırsanız **hata mesajı** alırsınız.

Bu komut büyük ihtimalle şuna benzer bir **çıktı** verecektir:

```
Python 3.x.y
```

Sürüm numarası **3** ile başladığı için **`python3`** komutunun yeni **`3.x`** **sürümünü başlattığını anlıyoruz**.

Bir de şu komutun **çıktısını** kontrol edin:

```
python --version
```

Bu da büyük ihtimalle şöyle bir **çıktı** verecektir:

```
Python 2.x.y
```

Buradan da **`python`** komutunun eski **`2.x`** sürümünü başlattığını anlıyoruz.

Eğer **`python3`** veya **`python`** komutunu verdiğinizde yukarıdaki ekran yerine bir hata mesajıyla karşılaşıyorsanız **iki ihtimal var:**

Birincisi, '**`python`**' kelimesini **yanlış yazmış** olabilirsiniz. Mesela yanlışlıkla '**`pyhton`**', '**`pyton`**', '**`phyton`**' veya '**`Python`**' yazmış olabilirsiniz. Doğru kelimenin **tamamen küçük harflerden** oluştuğuna özellikle dikkat etmemiz gerekiyor. Ayrıca '**`python`**' kelimesi ile **`3`** sayısının da birbirine **bitişik** olduğunu gözden kaçırmıyoruz.

İkincisi, eğer ilk maddede söylenenlerin geçerli olmadığından eminseniz çok düşük bir ihtimal olmakla birlikte Python'ın hiçbir sürümü **sisteminizde kurulu değil** demektir. Yalnız GNU/Linux dağıtımlarında Python'ın en az bir sürümünün kurulu olmama ihtimalinin **sıfıra yakın** olduğunu söyleyeyim. Bu yüzden, sisteminizde Python'ın herhangi bir sürümünün kurulu olmadığına kesinkes karar vermeden önce durumunuzun **birinci madde kapsamına** girmediğinden emin olmalısınız.

Eğer Python'ın kullandığınız GNU/Linux dağıtımında kurulu olmadığına eminseniz öncelikle kullandığınız dağıtımın **paket/yazılım depolarına bakın**. Orada Python'ın **3.x** sürümlerinden birini mutlaka bulursunuz.

Fakat aradığınız sürüm, dağıtımınızın depolarında yoksa Python'ı **kaynaktan derlemeyi** de tercih edebilirsiniz.

Bunun için öncelikle *https://www.python.org/downloads* veya *https://www.python.org/ftp/python/* adresinden Python **3.8.3** sürümünün **kaynak kodlarını** bilgisayarınıza indirin.

Daha sonra siteden indirdiğiniz sıkıştırılmış klasörü açıp orada sırasıyla `./configure`, `make` ve `make install` komutlarını verin.

Burada farklı olarak `make altinstall` yerine `make install` komutunu kullandığımıza dikkat edin. Çünkü sizin sisteminizde Python'ın hiçbir sürümü kurulu olmadığı için elle kuracağınız yeni sürümün eski bir sürümle **çakışma riski de yok**. O yüzden `make altinstall` yerine doğrudan `make install` komutunu kullanabilirsiniz.

Ama daha önce de dediğimiz gibi eğer kaynaktan program kurmak konusunda tecrübeli değilseniz ve karşılaşabileceğiniz sorunları çözemeyeceğinizden endişe ediyorsanız **sisteminizdeki** veya **paket depolarınızdaki Python3** sürümüyle devam etmeniz en doğrusu olabilir. Python'ın **3.6 ve üzeri** herhangi bir sürümü bu kitabı takip etmeniz için yeterli olacaktır.

Python'ı nasıl çalıştıracağımızı öğrendiğimize göre artık Python'a ilişkin bazı **temel kavramlardan** söz etmeye başlayabiliriz.

BAZI TEMEL KAVRAMLAR

4

Bir önceki bölümde Python'ın komut satırına nasıl ulaşacağımızı görmüştük. Bu komut satırına Pythonca'da "**etkileşimli kabuk**" veya "**yorumlayıcı**" adı veriyoruz:

```
Python 3.8.3 (tags/v3.8.1:1b293b6, 14/07/2020, 12:24:55)
[MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Python'ın etkileşimli kabuğunu **>>>** işaretinden tanıyabilirsiniz. Eğer bir komut ortamında bu işareti görüyorsanız Python'ın **etkileşimli kabuğundasınız** demektir. Dolayısıyla Python komutlarını çalıştırabilirsiniz. Ama eğer bu işareti **görmüyorsanız** içinde bulunduğunuz komut ortamı muhtemelen etkileşimli kabuk değil, **sistem komut** satırıdır. Sistem komut satırında sadece kullandığınız işletim sistemine ait komutları çalıştırabilirsiniz. Python komutlarını değil...

Özellikle Python programlama diline yeni başlayanlar bu iki ortamı zaman zaman birbirine karıştırıp Python komutlarını **sistem komut satırında**, sistem komutlarını ise **etkileşimli kabukta** çalıştırmaya kalkışabiliyor. O yüzden bu iki ortamın birbirinden **farklı** olduğunu üstüne basa basa belirtiyoruz.

Aslına bakarsanız etkileşimli kabuk bizim asıl çalışma ortamımız değildir. Yani biz normalde kodlarımızı etkileşimli kabuğa değil, **bir program dosyasına yazıp kaydedeceğiz**. Ama etkileşimli kabuk, yazdığımız Python kodlarının **sonucunu** (*yani çıktısını*) hemen verdiği için bu ortam bize kodlarımızı çok hızlı bir şekilde test etme imkânı sağlar. Dolayısıyla etkileşimli kabuk, Python'a ilişkin **temel bilgileri öğrenmek açısından son derece uygun** bir ortamdır. O yüzden kodlarımızı bir program dosyasına kaydedip yürütmeye hazır hâle gelene kadar ilk birkaç bölüm boyunca çalışmalarımızı **yalnızca etkileşimli kabuk** üzerinde yapacağız.

Şimdi, daha önce anlattığımız yöntemler arasından kendi işletim sistemimize uygun olanını kullanarak Python'ın **etkileşimli kabuğunu** açalım ve şuna benzer bir ekranla karşılaşalım:

```
Python 3.8.3 (tags/v3.8.1:1b293b6, 14/07/2020, 12:24:55)
[MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

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

Bu bölümde, Python'ın temeline ilişkin bazı **önemli kavramlardan** söz edeceğiz. Bu bölümün sonuna ulaştığımızda, Python'da daha rahat ilerleyebilmemiz için gereken hemen hemen **tüm başlangıç bilgilerini edinmiş** ve **geleceğe doğru sağlam bir adım** atmış olacağız.

Burada göreceğimiz **temel kavramları `print()` adlı bir fonksiyon üzerinden ele alacağız**. Dilerseniz hiç vakit kaybetmeden bu **`print()`** adlı fonksiyonun ne işe yaradığını anlatarak konumuza giriş yapalım.

PRINT() FONKSİYONU

Kabaca tanımlamak gerekirse Python'da fonksiyonlar, belirli bir işlevi yerine getiren araçlardır. Python'da pek çok önemli işlevi yerine getiren çok sayıda **hazır fonksiyon bulunur**. Bu bölümde ele alacağımız `print()` fonksiyonu da bunlardan biridir. *Peki, acaba bu* `print()` *fonksiyonu, hangi işlevi yerine getirir?*

Python'da `print()` fonksiyonu, ekrana **çıktı** vermemizi sağlar. Mesela bu komutu **tek başına** kullanmayı deneyelim:

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

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

Ne oldu? Python bir satır boşluk bırakarak alt satıra geçti, değil mi? Bunu ona yapmasını biz söyledik, o da yaptı... Şimdi de Python'a, boş bir satır bırakmak yerine **ekrana bir şeyler yazmasını söyleyelim**:

```
>>> print("merhaba zalim dünya")
```

Bu satırı yazıp **ENTER** tuşuna bastıktan sonra ekranda `merhaba zalim dünya` çıktısını görmemiz gerekiyor.

`print()` fonksiyonu, Python'daki **en önemli** ve **en temel komutlardan** biridir. Python'la yazdığınız programlarda kullanıcılarınıza **herhangi bir mesaj göstermek istediğinizde** bu `print()` fonksiyonundan yararlanacaksınız.

Bu fonksiyonun çok basit bir şekilde nasıl kullanıldığını öğrendiğimize göre gelin isterseniz bir de "**karakter dizisi**" adlı bir kavramdan söz edelim.

Karakter Dizileri

Yukarıdaki örnekte, **`print()`** fonksiyonunun parantezleri arasına yazdığımız "**`merhaba zalim dünya`**" ifadesini **çift tırnak içinde** belirttiğimize dikkat edin.

Python programlama dilinde tırnak içinde gösterilen bu tür değerlere **karakter dizisi** ('**`string`**') adı verilir. Karakter dizilerinin ayırt edici özelliği, **tırnak işaretleri içinde tanımlanmış olmaları**dır.

Eğer burada tırnak işaretlerini koymazsak veya koymayı unutursak Python bize bir **hata mesajı** gösterecektir. Zaten yukarıdaki komutu verdikten sonra bir hata alıyorsanız çok büyük bir ihtimalle ya *tırnakları düzgün yerleştirmemişsinizdir* ya da **`print()`** *fonksiyonunun parantezlerinden birini koymayı unutmuşsunuzdur*. Python programlama maceramız boyunca aklımızdan çıkarmamamız gereken en önemli kurallardan biri açtığımız **parantezleri** veya **tırnakları kapatmayı asla unutmamaktır...**

Bu arada biz istersek **`print()`** fonksiyonu içindeki karakter dizisini göstermek için çift tırnak yerine **tek tırnak** (') da kullanabiliriz:

```
>>> print('merhaba zalim dünya')
```

Hatta istersek **üç tırnak** bile kullanabiliriz:

```
>>> print('''merhaba zalim dünya''')
```

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

Karakter dizilerinden ileride çok daha ayrıntılı bir şekilde söz edeceğiz. Ama şimdi dilerseniz önemli bir **temel kavramdan** daha bahsedelim.

Kaçış Dizileri

Yukarıdaki örneklerden de gördüğünüz gibi karakter dizilerini tanımlarken **tek**, **çift** veya **üç tırnak** işaretlerini kullanabiliyoruz. Ancak karakter dizilerini tanımlarken karakter dizisi içindeki başka kesme ve tırnak işaretlerine karşı da dikkatli olmalıyız.

Örneğin:

```
>>> print("İstanbul'un semtleri")
```

Bu komut bize hatasız bir şekilde **`İstanbul'un semtleri`** **çıktısını** verir.

Ancak yukarıdaki karakter dizisini çift tırnak yerine **tek tırnakla** tanımlamaya çalışırsak **hata mesajı** alırız:

```
>>> print('İstanbul'un semtleri')
```

Bunun nedeni, Python'ın, "**`İstanbul'un`**" kelimesindeki **kesme işareti**nden ötürü, karakter dizisini tanımlayan tırnakların nerede başlayıp nerede bittiğini anlayamamasıdır. Eğer mutlaka tek tırnak kullanmak istiyorsak kodu şu hâle getirmemiz gerekir:

```
>>> print('İstanbul\'un semtleri')
```

Burada `\` işareti, Python'ın, "**`İstanbul'un`**" kelimesindeki **kesme işaretine takılmadan** ilerlemesini sağlıyor.

Aynı şekilde şu kodlar da **hata** verecektir:

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

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

Bu **hatayı** da şu şekilde önleyebiliriz:

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

Burada da karakter dizisini tanımlamamızı sağlayan tırnak işaretleri ile karışma ihtimali bulunan başka tırnak işaretlerinin her birinin önüne \ işaretini yerleştirerek **Python'ın bu işaretlere takılmasını önlüyoruz**.

Sözün özü, tırnak işaretlerinin Python açısından tek bir anlamı vardır: **Karakter dizilerini tanımlamak**. Dolayısıyla, eğer siz tırnak işaretlerini karakter dizisi içinde başka bir amaçla kullanacaksanız bunların karakter dizisini tanımlamak için kullanılan tırnak işaretleri ile **birbirine karışacağını unutmamalı**, bunu önlemek için **gerekli önlemleri** almalısınız.

İşte böyle bir karışıklığı önlemek için yukarıda gösterdiğimiz \ işaretlerini karakter dizisi içindeki her bir tırnak işaretinin **sol tarafına** yerleştiriyoruz. Böylece bir bakıma tırnak işaretlerinin **anlamını değiştirmiş oluyoruz**.

Olası bir hatadan kaçmamızı veya bazı *karakterlerin/işaretlerin* anlamını değiştirmemizi sağlayan bu tür ifadelere Python dilinde **kaçış dizileri** (*'escape sequences'*) adı verilir. Kaçış dizilerini de ilerleyen derslerde çok daha ayrıntılı bir şekilde ele alacağız. Şimdilik bu kaçış dizisi kavramına takılmadan yolumuza devam edelim ve yine önemli bir kavrama giriş yapalım.

ARGÜMANLAR

Teknik olarak ifade etmemiz gerekirse **`print()`** fonksiyonunun parantezleri içine yazdığımız her bir değere '**argüman**' (*veya 'parametre'*) adı verilir. Şimdiye kadar **`print()`** fonksiyonunu hep tek argüman ile kullandık:

```
>>> print("merhaba")
merhaba

>>> print("Python, Java, C")
Python, Java, C
```

Ama eğer istersek bu fonksiyonu birden fazla argüman ile de "**çağırabiliriz**" (*yani kullanabiliriz*):

```
>>> print("Python", "programlama", "dili")
Python programlama dili

>>> print("Python", "Java", "C")
Python Java C
```

Gördüğünüz gibi **`print()`** fonksiyonunun parantezleri içine yazdığımız argümanların her birini birbirinden birer **virgül** ile ayırıyoruz.

Eğer argümanlar arasına **virgül yerleştirmezsek `print()`** fonksiyonu bu argümanları birbiriyle **bitiştirecektir**:

```
>>> print("bin" "beş" "yüz")
binbeşyüz

>>> print("mer" "haba")
merhaba
```

Eğer arzu ederseniz bu argümanların arasına **+** işareti koyarak da aynı sonucu elde edebilirsiniz:

```
>>> print("bin" + "beş" + "yüz")
binbeşyüz

>>> print("mer" + "haba")
merhaba
```

Şimdiye kadar, **`print()`** fonksiyonuna **argüman** olarak yalnızca karakter dizilerini verdik. Ama eğer istersek **sayıları** da birer argüman olarak kullanabiliriz:

```
>>> print("Yaşınız:", 40)
Yaşınız: 40
```

Ancak argümanlar arasında sayılar varsa arasına **virgül koymayı kesinlikle ihmal etmemeliyiz**. Mesela şu kod **hata** verecektir:

```
>>> print("Yaşınız:" 40)

  File "<stdin>", line 1
    print("Yaşınız:" 40)
                     ^
SyntaxError: invalid syntax
```

Gördüğünüz gibi argümanlar karakter dizisi olduğunda eğer bunlar arasına virgül koymazsak **`print()`** fonksiyonu bu argümanları birbirine bitiştiriyor ama eğer söz konusu olan şey **sayılar** ise **virgül kullanmadığımızda** birleştirme işlemi yapmak yerine **hata** veriyor:

```
>>> print(2 2)

  File "<stdin>", line 1
    print(2 2)
            ^
SyntaxError: invalid syntax
```

*Peki, sayılar arasına **+** işareti koyarsak ne olur?*

```
>>> print(2 + 2)
4
```

Burada iki sayı arasına yerleştirilen **+** işaretinin, bu iki sayının birbiriyle toplanmasını sağlaması hiç şaşırtıcı değil...

Sayılar arasında **aritmetik işlemlerin nasıl yapılacağından** ilerleyen derslerde söz edeceğiz. Ama biz şimdilik bunu bir kenara bırakıp yolumuza devam edelim.

SEP ARGÜMANI

`print()` fonksiyonunun kullanımında dikkatimizi çeken bir başka önemli nokta, bu fonksiyonun kendisine verilen argümanları ekrana yazdırırken her bir argümanın arasına bir **boşluk** yerleştiriyor oluşudur:

```
>>> print("İstanbul", "Büyükşehir", "Belediyesi")
İstanbul Büyükşehir Belediyesi

>>> print("bir", "iki", 1, 2)
bir iki 1 2
```

Ancak eğer biz istersek argümanların arasına yerleştirilecek **karakteri değiştirebiliriz**.

Bunun için **`print()`** fonksiyonuna ait **`sep`** adlı özel bir argümandan yararlanacağız.

Çok basit bir **örnek** verelim:

```
>>> print("www", "istihza", "com", sep=".")
www.istihza.com
```

Gördüğünüz gibi **sep** adlı argümanın değerini, öteki argümanlardan farklı olarak **ismiyle birlikte** kullanıyoruz. Zaten eğer **sep** argümanının ismini belirtmezsek bu argümanın gösterdiği değerin öteki argümanlardan hiçbir farkı kalmaz:

```
>>> print("www", "istihza", "com", ".")
www istihza com .
```

sep adlı argümanın öntanımlı değeri **boşluk karakteridir**. Yani **print()** fonksiyonu içinde herhangi bir **sep** değeri belirtmezsek Python bu argümanın değerini **boşluk karakteri olarak varsayacaktır**.

İşte biz yukarıdaki örnekte **sep**'in bu öntanımlı değerini değiştirip '.' yaptık. Böylece her bir argümanın arasına **boşluk karakteri yerine nokta karakteri** yerleştirilmiş oldu.

Bunu daha iyi anlamak için bir **örnek** daha verelim:

```
>>> print("elma", "armut", "çilek", "ayva", sep="-")
elma-armut-çilek-ayva
```

Burada da **sep** argümanının değerini '**-**' karakteri yaparak her bir argümanın arasına '**-**' işaretini yerleştirdik.

Hatta istersek bu argümanın değerini boş bir karakter dizisi yaparak öteki argümanlar arasındaki **boşlukları tamamen kaldırabiliriz de**:

```
>>> print("afyon", "kara", "hisar", sep='')
afyonkarahisar
```

Teknik olarak ifade etmemiz gerekirse **sep** gibi argümanlara **isimli argümanlar**, "**elma**", "**armut**", "**www**", **40** gibi argümanlara ise **isimsiz argümanlar** adı verilir. Burada dikkat edeceğimiz nokta; **isimli argümanları her zaman isimsiz argümanlardan sonra getirmektir**. Mesela şu kod **hata** verecektir:

```
>>> print(sep="-", "elma", "armut", "çilek", "ayva")
```

Birazdan **print()** fonksiyonunun başka isimli argümanlarına da değineceğiz. Ama dilerseniz şimdi **print()** fonksiyonunun başka bir özelliğinden söz edelim.

Argümanların Dağıtılması

Python'da **print()** fonksiyonunun önemli bir özelliği * işareti yardımıyla argümanları **ögelerine ayırmaya** (*veya dağıtmaya*) müsaade etmesidir. *Peki, bu tam olarak ne anlama geliyor?*

Şimdi, dikkatlice bakın:

```
>>> print(*"istihza")
i s t i h z a
```

Gördüğünüz gibi burada "**istihza**" argümanının **sol tarafına** bir adet * işareti yerleştirdiğimizde sanki **print()** fonksiyonunu şöyle yazmışız gibi bir **sonuç** ortaya çıktı:

```
>>> print("i", "s", "t", "i", "h", "z", "a")
i s t i h z a
```

Yani * işareti, "**istihza**" argümanını meydana getiren her bir ögeyi ayrı bir argüman olarak **print()** fonksiyonuna gönderdi.

Doğal olarak şu anda her bir argüman arasında **birer boşluk** var. Bunun sebebi, **sep** adlı isimli argümanın öntanımlı değerinin **boşluk karakteri** olmasıdır. Ama eğer istersek bu boşluk yerine istediğimiz **herhangi bir karakteri** yerleştirebileceğimizi biliyorsunuz:

```
>>> print(*"istihza", sep="/")
i/s/t/i/h/z/a
```

Burada **sep** adlı isimli argümanın öntanımlı değerini değiştirerek * işareti ile dağıttığımız her bir öge arasına bir adet / işareti yerleştirdik.

Şimdi de her bir argümanın arasına **ikişer boşluk** koyalım:

```
>>> print(*"istihza", sep="  ")
i  s  t  i  h  z  a
```

Bu yıldızlı yapıyı **aynı anda birden fazla argüman üzerinde** de uygulayabiliriz:

```
>>> print(*"Türkiye", *"Büyük", *"Millet", *"Meclisi", sep=".")
T.ü.r.k.i.y.e.B.ü.y.ü.k.M.i.l.l.e.t.M.e.c.l.i.s.i
```

Son bir **örnek** daha verelim:

```
>>> print(*"0123456789", sep="-")
0-1-2-3-4-5-6-7-8-9
```

Gördüğünüz gibi **sep** adlı isimli argümanın görevi, **`print()`** fonksiyonunun isimsiz argümanları arasına istediğimiz herhangi bir **karakter dizisini** yerleştirebilmemizi sağlamaktır.

Bu arada ***** işaretini sayılarla birlikte kullanamıyoruz. Yani şu kodlar **hata** verecektir:

```
>>> print(*12345)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: print() argument after * must be an iterable, not int
```

Eğer bu işareti, **sayı** değerli verilerle birlikte kullanmak istiyorsak o sayı değerli veriyi bir **karakter dizisine** dönüştürmemiz gerekir. Bunun için o sayıyı **tırnak içine** almamız yeterli olacaktır:

```
>>> print(*"12345")
1 2 3 4 5
```

Bu dönüştürme işlemlerinden daha sonra tekrar bahsedeceğiz. Ama ondan önce anlatmamız gereken başka şeyler var.

END ARGÜMANI

Bu bölümü kapatmadan önce **`print()`** fonksiyonunun bir başka isimli argümanından daha bahsedelim. Bu isimli argümanın adı **`end`** olup görevi **argümanların sonuna hangi karakterin geleceğini** belirlemektir. Mesela şu kodu inceleyelim:

```
>>> print("Osmaniye Belediyesi Park ve Bahçeler Genel Müdürlüğü")
```

Bu kodu çalıştırdığımızda şu **çıktıyı** alıyoruz:

```
Osmaniye Belediyesi Park ve Bahçeler Genel Müdürlüğü
```

Python, bu çıktıyı verdikten sonra bir **alt satıra** geçiyor. Pek belli olmasa da aslında önemli bir özelliktir bu. Zira aslında biz çıktı verildikten sonra alt satıra geçilmesini Python'dan açık açık talep etmedik. Python bizim yerimize karar vererek çıktı sonrası **bir alt satıra geçmeyi** tercih etti.

Ama bazı durumlarda bizim istediğimiz şey bu olmayabilir. Python'ın tüm argümanları yazdırdıktan sonra bir alt satıra değil, mesela iki alt satıra geçmesini de isteyebilirsiniz. Veya alt satıra geçmeden önce argümanların sonuna bir tane **nokta işareti** koymasını isteyebilirsiniz. Ya da çıktıyı verdikten sonra hiç satır değiştirmeden aynı satırda kalmasını isteyebilirsiniz. İşte bu tür ihtiyaçlarımız için **`end`** adlı bir **isimli argümandan** yararlanacağız.

Mesela argümanlar yazdırıldıktan sonra bir alt satıra değil, **iki alt satıra geçilmesini sağlayalım**:

```
>>> print("Merhaba Zalim Dünya", end="\n\n")
Merhaba Zalim Dünya

>>>
```

Hatırlarsanız yukarıda "**kaçış dizilerinden**" söz etmiştik. İşte tam bu noktada, özel bir kaçış dizisinden daha bahsetmemiz gerekiyor.

Yeni Satır Kaçış Dizisi

Daha önce de söylediğimiz gibi **end** parametresini özellikle biz değiştirmezsek öntanımlı olarak bir alt satıra geçilmesini sağlayacaktır. Programlamada "**bir alt satıra geçme**" işlemi **\n** adlı bir kaçış dizisi yardımıyla gösterilir. Yani eğer biz bir noktada yeni satıra geçilmesini istiyorsak **\n** kaçış dizisinden yararlanacağız. Dilerseniz bu kaçış dizisinin nasıl kullanıldığına dair bir iki **örnek** yapalım:

```
>>> print("bir\niki")
bir
iki
```

Gördüğünüz gibi karakter dizisi, **\n** kaçış dizisinin olduğu noktadan **bölündü** ve bu işaretten sonra gelen kısım **bir alt satıra geçti**:

```
>>> print("birinci satır\nikinci satır\nüçüncü satır")
birinci satır
ikinci satır
üçüncü satır
```

Peki ama bu nasıl oldu?

Bildiğiniz gibi normalde '**n**' harfinin görevi, ekrana '**n**' harfini **basmaktır**:

```
>>> print('klmnopr')
klmnopr
```

Ama eğer siz bu '**n**' harfinin önüne \ işaretini koyarsanız '**n**' harfinin **anlamını değiştirmiş** olursunuz:

```
>>> print('klm\nopr')
klm
opr
```

Gördüğünüz gibi '**n**' harfinin önüne \ işaretini getirerek elde ettiğimiz **\n** adlı kaçış dizisi, **bir alt satıra geçilmesini sağlıyor**.

Hatırlarsanız \ işaretini kullanarak daha önce tırnak işaretlerinin de anlamını nasıl değiştireceğimizi öğrenmiştik:

```
>>> print('İstanbul\'un semtleri')
İstanbul'un semtleri

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

Bildiğiniz gibi Python'da tırnak işaretlerinin birincil anlamı, **karakter dizilerini tanımlamaktır**. Ama eğer biz bu tırnak işaretlerini farklı bir amaçla kullanacaksak yukarıdaki örneklerde olduğu gibi önlerine \ işaretini getirerek bunların anlamını değiştirmemiz gerekir.

İşte aynı şekilde '**n**' harfinin de önüne \ işaretini getirerek bu harfin anlamını değiştirebilir ve **\n** adlı **yeni satır kaçış dizisini** elde edebiliriz.

Bu **\n** kaçış dizisini daha önce öğrendiğimiz **sep** argümanı içinde de kullanabilirsiniz:

```
>>> print("Python", "Java", "C++", sep="\n")
Python
Java
C++
```

Bu şekilde her bir isimsiz argüman arasına bir adet **yeni satır**, yani **\n** kaçış dizisi eklemiş oluyoruz. Böylece her argümandan sonra **bir alt satıra geçiliyor**. İşte **end** isimli argümanın öntanımlı değeri de budur ve eğer **istersek biz bu öntanımlı değeri değiştirebiliriz**:

```
>>> print("Merhaba Zalim Dünya", end=".")
Merhaba Zalim Dünya.>>>
```

Yalnız burada bir tuhaflık olduğunu fark etmişsinizdir. Normalde satır başında olması gereken **>>>** işareti, çıktı ile aynı satırda kaldı. Bu durumun sebebi, **end** argümanının öntanımlı değerini değiştirmiş olmamızdır. Biz bu parametrenin **\n** olan öntanımlı değerini silip yerine '**.**' karakter dizisini yerleştirdiğimiz için çıktı verildikten sonra **bir alt satıra geçilmedi**.

Hem **karakter dizisinin sonuna nokta koymak** hem de **alt satıra geçilmesini sağlamak** istiyorsak **end** argümanının değerini şöyle yapmamız gerekir:

```
>>> print("Merhaba Zalim Dünya", end=".\n")
Merhaba Zalim Dünya.
```

Bu defa **end** argümanının değerini '**.\n**' şeklinde yazdığımız için hem karakter dizisinin en sonuna bir **nokta işareti** koyuldu hem de **bir alt satıra geçildi**.

Bir **örnek** daha verelim:

```
>>> print(*"Türkiye", *"Büyük", *"Millet", *"Meclisi", sep=".",
end=".\n")
T.ü.r.k.i.y.e.B.ü.y.ü.k.M.i.l.l.e.t.M.e.c.l.i.s.i.
```

Burada **sep** argümanının değerini '**.**' olarak **end** argümanının değerini ise '**.\n**' olarak belirledik. Böylece hem her bir argümanın arasına bir **nokta işareti** yerleştirildi hem de **en son argümanın sonuna bir tane nokta koyuldu**. Ayrıca nokta ile birlikte kullandığımız yeni satır işareti sayesinde de en son karakter dizisi ekrana basıldıktan sonra **bir alt satıra geçildi**.

Bu konuyla ilgili **son örneğimiz** de şu olsun:

```
>>> print("\nPython Java C++", end="\n------ ---- ---\n")

Python Java C++
------ ---- ---
```

Bu örneği kendi kendinize inceleyerek neyi ne için yaptığımızı anlamaya çalışmanız bu bölümde öğrendiklerimizi sindirmeniz bakımından oldukça faydalı bir uğraş olacaktır.

Böylece Python'daki en önemli unsurlardan biri olan **`print()`** fonksiyonunu epey ayrıntılı bir şekilde ele almış olduk. Aynı zamanda Python programlama diline dair çok önemli bazı temel bilgiler de edindik.

Burada işlediğimiz konuların tamamı bir anda zihninizde yer etmemiş olabilir. Eğer öyleyse hiç endişe etmenize gerek yok. Zira burada incelediğimiz konuları bundan sonra da bol bol göreceğiz. Dolayısıyla burada anlatılanlar, siz istemeseniz de bir süre sonra belleğinizde yer edecektir. Ben size, **`print()`** fonksiyonunu kullanarak **kendi kendinize çeşitli deneme çalışmaları yapmanızı tavsiye ederim**. Python'ın son derece önemli ögelerinden biri olan bu fonksiyonla, bundan sonra da bol bol haşır neşir olacağız.

PYTHON'DA SAYILAR VE ARİTMETİK İŞLEMLER

5

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

Örneğin:

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

... veya:

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

... ya da:

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

... hatta:

```
>>> 6 / 2
3.0

>>> 5 / 2
2.5
```

Burada Python'ın bölme işlemlerinin sonucunu **kayan noktalı sayı** olarak verdiğine dikkat ediyoruz.

Peki, acaba "**kayan noktalı sayı**" ne demek?

Teknik olarak ifade etmemiz gerekirse ondalık kısmı olan sayılara "**kayan noktalı sayı**" ("*floating point number*") adı verilir. Ondalık kısmı olmayan sayılara ise **tam sayı** ("*integer*") denir. Birazdan bu konuyu daha ayrıntılı bir şekilde ele alacağız.

Eğer bölme işleminin sonucunu kayan noktalı sayı olarak değil de **tam sayı şeklinde** elde etmek istiyorsanız `//` işlecinden yararlanabilirsiniz:

```
>>> 10 // 2
5

>>> 5 // 2
2
```

Burada `//` **işleci hassas bölme yerine taban bölme işlemi** yapabilmemizi sağlıyor.

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

Toplama	+ işleci
Çıkarma	- işleci
Çarpma	* işleci
Hassas bölme	/ işleci
Taban bölme	// işleci

Bunların dışında işimize yarayacak **birkaç işleç daha** öğrenelim:

```
>>> 2 ** 2
4

>>> 2 ** 3
8

>>> 6 % 3
0

>>> 9 % 2
1
```

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

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

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

Bir sayının 2. kuvveti o sayının karesidir. Aynı şekilde bir sayının **`0.5`**. kuvveti de o sayının kareköküdür:

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

Yukarıda gördüğümüz **`%`** işleci ise **bölme işleminden kalan sayıyı** gösterir. Örneğin 6 sayısını 3'e böldüğümüzde, bu sayı 3'e tam bölündüğü için kalan **0**'dır. Bir sonraki örnekte gördüğümüz 9 sayısı ise 2'ye bölündüğünde kalan **1** olduğu için **`9 % 2`** işleminin değeri **`1`** olacaktır.

Bu işleci kullanarak mesela **tek** ve **çift** sayıları tespit edebilirsiniz. Eğer bir sayı 2'ye bölündüğünde kalan **0** ise o sayı **çifttir**, aksi hâlde o sayı **tektir**:

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

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

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

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

Bu arada sayıları yazarken **tırnak işaretlerini kullanmadığımıza** dikkat edin. Eğer tırnak işareti kullanırsak; Python yazdıklarımızı sayı olarak değil **karakter dizisi** olarak algılayacaktır. Bu durumu **birkaç örnekle görelim**:

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

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

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

Bu komut **25** ile **50**'yi toplamak yerine, ekrana **25 + 50** şeklinde bir **çıktı** verecektir.

Peki, şöyle bir komut verirsek ne olur?

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

Böyle bir komutla karşılaşan Python, derhal **'25'** ve **'50'** karakter dizilerini (*bu sayılar tırnak içinde olduğu için Python bunları sayı olarak algılamaz*) yan yana getirip birleştirecektir.

Yani şöyle bir şey yapacaktır:

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

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

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

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

işleminin sonucu **8.0** olacaktır. Tabii biz istersek **parantezler yardımıyla** Python'ın kendiliğinden kullandığı öncelik sırasını değiştirebiliriz:

```
>>> (2 + 6) / 3 * 5 - 4
9.333333333333332
```

Buraya kadar Python'daki verilerin 3 farklı tipte olduğunu gördük.

Bunlar:

- Karakter dizileri ('*string*')
- Tam sayılar ('*integer*')
- Kayan noktalı sayılar ('*floating point number*')

Teknik olarak ifade etmemiz gerekirse farklı tipteki bu verilerin her birine **veri tipi** ('*data type*') adı verilir.

Tek, **çift** ve **üç tırnak** içinde gösterilen her şey **birer karakter dizisi**dir:

```
>>> ‘aaa’
>>> “aaa”
>>> ‘’’aaa’’’
>>> ‘0’
>>> “1”
>>> ‘’’fd345$#&’’’
```

Hatta boşluk karakterinin kendisi de **bir karakter dizisi**dir:

```
>>> ‘ ‘
>>> “ “
>>> ‘’’ ‘’’
```

Şunlar da **içi boş karakter** dizileridir:

```
>>> ‘’
>>> “”
>>> ‘’’’’’
```

Gelelim tam sayılara... Tam sayılar, herhangi bir ondalık kısım içermeyen **negatif** ve **pozitif** sayılardır. Mesela şunlar **birer tam sayı**dır:

```
>>> 0
>>> 1
>>> 10
>>> -3
>>> -5
```

Ondalık kısım içeren **negatif** ve **pozitif** sayılara ise **kayan noktalı sayı** adı verilir:

```
>>> 0.5
>>> 5.0
>>> -3.1
>>> -19.13
```

Böylece bu bölümde hem sayılardan hem de Python'daki veri tiplerinden kısaca söz etmiş olduk. Esasında Python'da bunların dışında başka veri tipleri de bulunur. Ama biz şimdilik veri tiplerine ara verip **değişkenler** konusuna değinelim biraz.

DEĞİŞKENLER 6

Bu bölümde, genel olarak Python'daki değişkenlerden ve bu değişkenlerin çeşitli kabiliyetlerinden söz edeceğiz. Ayrıca bu bölümde özellikle **karakter dizilerinin bazı önemli yönlerine** de değineceğiz.

İlk önce **değişkenlerden** başlayalım.

Kabaca ifade etmek gerekirse; bir veriyi adlandıran veya bir veriye işaret eden birimlere **değişken** adı veriyorlar. Dilerseniz hemen basit bir **örnek** üzerinden değişkenlerin neye benzediğini anlamaya çalışalım.

Mesela şu **örneğe** bakalım:

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

Burada **n** bir değişkendir. Python komut satırında **n = 5** şeklinde değişkeni tanımladıktan sonra **n** komutunu verirsek ekrana yazdırılacak veri **5** olacaktır.

Yani:

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

Bu **n** değişkenini alıp bununla **aritmetik işlemler** yapabiliriz:

```
>>> n * 2
10

>>> n / 2
2.5

>>> n // 2
2
```

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

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

Şu **örneklere** bir göz atalım:

```
>>> a = 5
>>> b = 3
>>> a * b
15

>>> print(a, "ile", b, "sayılarını çarparsak", a * b, "elde
ederiz.")
5 ile 3 sayılarını çarparsak 15 elde ederiz.
```

Hatırlarsanız geçen bölümde **`print()`** fonksiyonunun birden fazla **argüman** (*veya parametre*) alabildiğini söylemiştik. Gördüğünüz gibi yukarıdaki **`print()`** fonksiyonu da toplam **6 argüman** almış:

```
a
"ile"
b
"sayılarını çarparsak"
a * b
"elde ederiz."
```

`print()` fonksiyonunun argümanlarını oluşturan ve birbirlerinden **virgüllerle ayrılan** bu değişken ve karakter dizilerini fonksiyon içine nasıl yerleştirdiğimize dikkat edin:

```
>>> print(a, "ile", b, "sayılarını çarparsak", a * b, "elde
ederiz.")
```

Bu yapıya alışmak için **basit birkaç örnek** daha verelim:

```
>>> a = 5
>>> b = 3
>>> print(a, "sayısı", b, "sayısından büyüktür")
5 sayısı 3 sayısından büyüktür

>>> kelime = "python"
>>> print(kelime, "kelimesi", *kelime, "harflerinden oluşur")
python kelimesi p y t h o n harflerinden oluşur
```

Gördüğünüz gibi son derece kolay bir yapı bu. Ama bu işlemleri yapmanın çok daha **pratik** bir yolu da bulunur. Şimdi, bu yolun ne olduğunu inceleyeceğiz.

f'li Karakter Dizileri

Yukarıdaki örneklerde değerlerin her birini **`print()`** fonksiyonuna ayrı birer argüman olarak verdik. Ancak bu değerleri kullanmanın başka bir yolu da **özel işaretler yardımıyla** bunları tek bir karakter dizisi içine **gömmektir**.

Şu örneğe bir bakalım:

```
>>> print(f"{a} ile {b} sayılarını çarparsak {a * b} elde ederiz.")
5 ile 3 sayılarını çarparsak 15 elde ederiz.
```

Python'da başında '**`f`**' harfi olan bu tür karakter dizilerine **f'li karakter dizileri** ('**`f-strings`**') adı verilir.

Burada öncelikle karakter dizisinin en başına bir 'f' harfi yerleştirdiğimize, değişkenleri ise süslü parantezler (**{}**) içinde gösterdiğimize dikkat edin:

```
>>> print(f"{a} sayısı {b} sayısından büyüktür")
5 sayısı 3 sayısından büyüktür
```

Eğer burada, karakter dizisinin başına getirdiğimiz '**f**' *harfini kullanmazsak* Python karakter dizisi içinde geçen değişken adlarını birer **değişken olarak algılamayacaktır:**

```
>>> print("{a} sayısı {b} sayısından büyüktür")
{a} sayısı {b} sayısından büyüktür
```

Eğer '**f**' harfini koyar ama **karakter dizisi içindeki süslü parantezleri koymazsak** yine hüsrana uğrarız:

```
>>> print(f"a sayısı b sayısından büyüktür")
a sayısı b sayısından büyüktür
```

Gördüğünüz gibi Python'ın ne yapmaya çalıştığımızı anlayabilmesi için '**f**' **harfi** ve **süslü parantezler olmazsa olmaz**:

```
>>> ad = "Fırat"
>>> soyad = "Özgül"
>>> print(f"ad={ad}, soyad={soyad}")
ad=Fırat, soyad=Özgül
```

Bu arada yukarıdaki **f**'li karakter dizisini şu şekilde **kısaltabilirsiniz**:

```
>>> print(f"{ad=}, {soyad=}")
ad='Fırat', soyad='Özgül'
```

Buradaki yapıyı çok dikkatli bir şekilde inceleyin. Gördüğünüz gibi süslü parantezler içinde kullandığımız '**=**' işareti, değişken adının aynı zamanda bir **karakter dizisi** olarak da kullanılmasına izin veriyor. Yani değişken adları zaten **ad** ve **soyad** olduğu için ayrıca '**ad**' ve '**soyad**' şeklinde karakter dizileri belirtmek zorunda kalmıyoruz.

f'li karakter dizilerinin **nasıl kullanıldığına** ilişkin bir **örnek** daha verelim:

```
>>> sayı = 12
>>> print(f"{sayı} sayısının karesi {sayı ** 2} sayısıdır.")
12 sayısının karesi 144 sayısıdır.
```

Bu yapı içinde **çeşitli işlemler yapabildiğimizi** de gözden kaçırmayın:

```
>>> print(f"2 + 3 = {2 + 3}")
2 + 3 = 5

>>> print(f"12'nin karesi = {12 ** 2}")
12'nin karesi = 144
```

Bu arada, yukarıdaki ilk örneği şu şekilde **kısaltma** imkânına da sahibiz:

```
>>> print(f"{2 + 3 = }")
2 + 3 = 5
```

Burada da '**=**' işareti sayesinde "**2 + 3**" ifadesini **iki kez** yazmak zorunda kalmıyoruz. Süslü parantezlerin dışında kalan karakter dizisi ile içinde kalan aritmetik ifade aynı olduğu için bunlardan **yalnızca birini yazmamız** yeterli oluyor. Bir **örnek** daha verelim:

```
>>> print(f"{4 * 4 = }")
4 * 4 = 16
```

Yani bu kodlarla şu ifadeyi **kısaltmış** oluyoruz:

```
>>> print(f"4 * 4 = {4 * 4}")
4 * 4 = 16
```

Gördüğünüz gibi bu yapı sayesinde hem **değişkenleri** hem de **doğrudan değerleri** kullanabiliyoruz. Ayrıca bu yapı, bir karakter dizisi içine çeşitli değerleri kolaylıkla yerleştirmemizi de sağlıyor. **f**'li karakter dizilerinden bundan sonra da bol bol bahsedeceğiz. Ama biz şimdi, değişkenlerinin önemli başka özelliklerine değinelim.

Çoklu Değişken Tanımlama

Bu bölümde, **değişkenlerin en basit şekilde şöyle kullanıldığını** gördük:

```
>>> sayı = 5
>>> isim = 'istihza'
```

Burada değeri **5** olan, **`sayı`** adında ve **tam sayı veri tipinde** bir değişken ile birlikte değeri "**`istihza`**" olan, **`isim`** adında ve **karakter dizisi tipinde** bir değişken tanımladık.

Eğer arzu edersek bu ikisini **tek hamlede** de tanımlayabiliriz.

Dikkatlice bakın:

```
>>> isim, sayı = "istihza", 5
>>> isim
'istihza'

>>> sayı
5
```

Burada dikkat edeceğimiz nokta, eşittir işaretinin **sol tarafındaki değişken adı sayısı** ile **sağ tarafındaki değer sayısının aynı olması** gerektiğidir:

```
>>> a, b, c = 3, 5, 7
>>> a
3

>>> b
5

>>> c
7

>>> şehir, semt = "İstanbul", "Kağıthane"
>>> şehir
'İstanbul'

>>> semt
'Kağıthane'
```

Sol tarafta **kaç tane değişken adı** varsa **sağ tarafta da o sayıda değer olmasına dikkat ediyoruz**. Aksi hâlde Python bize bir **hata** mesajı verecektir:

```
>>> a, b = "python"

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 2)
```

Son derece **faydalı** ve **kullanışlı** bir yapıdır bu. Mesela bu yapı sayesinde, **değişkenlerin değerini birbiriyle değiş tokuş edebiliriz**.

Önce **iki değişken** tanımlayalım:

```
>>> a, b = 1, 2
>>> a
1

>>> b
2
```

Şimdi de **a**'nın değerini **b**'ye, **b**'nin değerini de **a**'ya verelim:

```
>>> a, b = b, a
>>> a
2

>>> b
1
```

Bir **örnek** daha verelim:

```
>>> birinci, ikinci = "Python", "Java"
>>> birinci
'Python'

>>> ikinci
'Java'
```

Burada **`birinci`** ve **`ikinci`** adlı iki adet değişken tanımladık. Birinci değişkeninin değeri "**`Python`**", ikinci değişkeninin değeri ise "**`Java`**".

Şimdi, **birinci** ile **ikinci**nin **yerini değiştireceğiz**:

```
>>> birinci, ikinci = ikinci, birinci
>>> birinci
'Java'

>>> ikinci
'Python'
```

Bu özellik her programlama dilinde bulunmaz. Eğer Python'da da bu özellik olmasaydı iki değişkenin değerini birbiriyle nasıl değiş tokuş edebileceğinizi düşünerek iyi bir zihin jimnastiği yapabilirsiniz...

Değişkenlerin Dağıtılması

Değişkenlerin ve karakter dizilerinin ilgi çekici bir başka özelliği, **print()** fonksiyonunu anlatırken gördüğümüz şekilde tıpkı argümanları dağıttığımız gibi **yıldızlı yapıları kullanarak değişkenleri de dağıtabilecek olmamızdır**.

Diyelim ki elimizde şöyle bir **karakter dizisi** var:

```
>>> fiyat = "#343,428$"
```

Biz bu karakter dizisinin başındaki ve sonundaki simgeleri almayıp yalnızca **orta kısımdaki sayıları** almak istiyoruz. *Peki, bu işi nasıl yapacağız?* İşte bunun için yıldızlı yapılardan yararlanabiliriz. Şimdi, çok dikkatli bakın:

```
>>> ilk, *orta, son = fiyat
>>> ilk
#

>>> orta
['3', '4', '3', ',', '4', '2', '8']

>>> son
$
```

Gördüğünüz gibi **yıldızlı yapılar sayesinde** bir karakter dizisinin ilk karakterini ayrı bir değişkene, son karakterini ayrı bir değişkene, geri kalan karakterlerini ise ayrı bir değişkene atayabiliyoruz. Yani **değişkeni oluşturan ögelerin her birini birbirinden ayırıp dağıtabiliyoruz**.

Burada **`orta`** adlı değişkenin tuttuğu değer gözünüze biraz farklı görünmüş olabilir.

Köşeli parantezler içinde birbirinden virgüllerle ayrılmış ögeler şeklinde gösterilen bu yapı, Python'da '**`liste`**' adı verilen farklı bir veri tipine karşılık gelir.

Bu noktaya kadar Python'da **karakter dizisi**, **tam sayı** ve **kayan noktalı sayı** adlı veri tiplerinden söz etmiştik. İlerleyen derslerde **`liste`** adlı veri tipini de inceleyeceğiz. Biz şimdi bu teknik ayrıntıları bir yana bırakıp yukarıdaki sayıları görünüş olarak nasıl daha işe yarar bir şekilde gösterebileceğimize bakalım.

Esasında siz bunu nasıl yapacağınızı biliyorsunuz. Dikkatlice bakın:

```
>>> print(*orta, sep="")
343,428
```

Burada da yine yıldızlı yapılardan yararlanarak orta adlı değişkenin her bir ögesini ayrı bir argüman olarak **`print()`** fonksiyonuna gönderdik ve **`sep`** adlı isimli argüman yardımıyla da her bir isimsiz argümanın arasındaki **boşluk karakterini kaldırdık**. Böylece istediğimiz çıktıyı elde etmiş olduk.

Yıldızlı yapıları **değişik kombinasyonlar** hâlinde kullanabilirsiniz. Mesela şunlara bakın:

```
>>> ilk, *gerisi = fiyat
>>> ilk
'#'

>>> gerisi
['3', '4', '3', ',', '4', '2', '8', '$']
```

Burada **fiyat** adlı karakter dizisini **ilk** ve **gerisi** şeklinde ikiye böldük. **ilk** adlı değişken, karakter dizisinin **ilk karakterini tutarken gerisi** adlı değişken, karakter dizisinin **geri kalanını tutuyor**.

gerisi adlı değişkenle nasıl daha düzgün bir şekilde **çıktı** verebileceğimizi biliyorsunuz:

```
>>> print(*gerisi, sep="")
343,428$
Bir de şuna bakalım:
>>> ücret = "125.000€"
>>> *meblağ, para_birimi = ücret
>>> print(*meblağ, sep='')
125.000

>>> print(para_birimi)
€
```

Bu da yukarıdaki işlemin **tam tersini** yapıyor. Bu defa karakter dizisinin baş tarafını ayrı bir değişkene, son karakterini ise ayrı bir değişkene atadık.

Bu yapıyı kullanarak şöyle bir şey yapmak da mümkündür:

```
>>> a, *b, c, d, e = "bolvadin"
>>> a
'b'

>>> b
['o', 'l', 'v', 'a']

>>> c
'd'

>>> d
'i'

>>> e
'n'
```

Bununla çeşitli denemeler yaparak bu yapının tam olarak nasıl işlediğini daha iyi anlayabilirsiniz.

Şöyle bir baktığınızda, bu yapı size biraz gereksizmiş gibi gelmiş olabilir. Ama ilerleyen zamanlarda Python bilgimiz arttığında bu yapıları **daha verimli** bir şekilde ve **yerli yerinde** nasıl kullanabileceğimizi anlayacağız.

Dilimleme

Esasında yukarıda yıldızlı yapılarla hâllettiğimiz işlemi, Python'da '**dilimleme**' adı verilen bir yöntemle de gerçekleştirebiliriz.

Dilimleme, karakter dizisi gibi bir veri tipinin ögelerini **teker teker alabilmemize** izin veren bir yöntemdir. Çok basit bir **örnek** verelim:

```
>>> kardiz = "$adana"
>>> kardiz[0]
$
```

Gördüğünüz gibi bir **karakter dizisinin sıfırıncı ögesi**, **o karakter dizisinin ilk ögesi oluyor**. Bir ögenin karakter dizisi içindeki sırasını parantezler içinde gösterdiğimize ve saymaya 0'dan başladığımıza özellikle dikkat edin. Şimdi de şuna bakın:

```
>>> kardiz[1]
'a'
```

Bu örnek de karakter dizisinin ikinci ögesini nasıl '**dilimleyeceğimizi**', yani **alacağımızı** gösteriyor. Burada da karakter dizisini dilimlerken saymaya sıfırdan başladığımızı gözden kaçırmıyoruz.

Peki, bir karakter dizisinin en sonuncu ögesini nasıl alacağız, yani dilimleyeceğiz?

Mesela '**`$adana`**' karakter dizisinin **6 karakterden** oluştuğunu bildiğimiz için sonuncu ögeyi şöyle dilimleyebiliriz:

```
>>> kardiz[5]
'a'
```

Fakat bir karakter dizisinin sonuncu ögesini almanın **daha pratik** bir yolu vardır:

```
>>> kardiz[-1]
'a'
```

Bir karakter dizisinin **-1**. ögesi o karakter dizisinin **sonuncu ögesi**dir. **-2**. ögesi ise o karakter dizisinin **sondan bir önceki ögesi**dir:

```
>>> kardiz[-2]
'n'
```

Python'daki bu dilimleme sistemini kullanarak yalnızca tek bir ögeyi değil, belli **bir aralık içindeki birden fazla ögeyi de alabiliriz**. Dikkatlice bakın:

```
>>> kardiz[1:4]
'ada'
```

Bu şekilde **1**. ila **4**. sıra arasında kalan karakterleri aldık. Bu yapıyı kullanarak çeşitli kombinasyonlar yapabilirsiniz:

```
>>> kardiz[0:-1]
'$adan'
```

Bu arada, eğer dilimleme işleminin başlangıç noktası sıfırıncı öge ise bunu belirtmesek de olur:

```
>>> kardiz[:-1]
'$adan'
```

Aynı şekilde dilimleme işleminin bitiş noktası son öge ise bunu da belirtmesek olur:

```
>>> kardiz[1:]
'adana'
```

Şimdi, tekrar şu karakter dizisine bakalım:

```
>>> fiyat = "#343,428$"
```

Acaba bu karakter dizisinin ilk ve son ögeleri arasında kalan rakamları dilimleyebilir miyiz?

Dikkatlice bakın:

```
>>> fiyat[1:-1]
'343,428'
```

Bu şekilde **`fiyat`** adlı karakter dizisinin birinci (*sıfırıncı değil*) ve en son ögeleri (*son öge hariç*) arasındaki karakterleri **dilimlemiş** olduk. Yukarıdaki komut hemen hemen şuna **eş değerdir**:

```
>>> ilk, *orta, son = fiyat
>>> print(*orta, sep='')
343,428
```

Duruma ve keyfinize göre herhangi bir yöntemi seçebilirsiniz.

Gördüğünüz gibi dilimleme yapılarının şöyle bir formülü var:

```
değer[başlangıç:bitiş]
```

Buna göre **iki nokta üst üste** işaretinden önceki kısım dilimleme işleminin **başlayacağı noktayı**, sonraki kısım ise **biteceği noktayı** gösteriyor.

Dilimleme yapılarının bir başka özelliği ise karakter dizisi üzerindeki dilimleme işleminin **kaçar kaçar atlanarak yapılacağını** belirleyebilmemize de imkân tanımasıdır. Yani yukarıdaki formül bir de üçüncü bir unsura sahiptir:

```
değer[başlangıç:bitiş:adım]
```

Mesela elimizde şöyle bir **karakter dizisi** olduğunu düşünelim:

```
>>> alfabe = "abcçdefgğhıijklmnoöprsştuüvyz"
```

Amacımız; bu karakter dizisi içindeki harfleri **ikişer ikişer atlayarak dilimlemek**. Yani biz bu karakter dizisi içindeki '**acdfğıjlnörşuvz**' harflerine **erişebilmek** istiyoruz. Dikkatlice bakın:

```
>>> alfabe[0:29:2]
'acdfğıjlnörşuvz'
```

Burada dilimleme işleminin başlangıç noktası olarak **0. karakteri**, bitiş noktası olarak ise **29. karakteri** belirledik. Bu aralık boyunca ikişer ikişer atlanarak ilerlenmesi için de **adım parametresini 2** olarak belirledik.

Bu arada daha önce de söylediğimiz gibi dilimleme işleminin başlangıç noktası 0. karakter, bitiş noktası ise sonuncu karakter olduğu için yukarıdaki komutu şöyle de yazabiliriz:

```
>>> alfabe[::2]
'acdfğıjlnörşuvz'
```

Başlangıç ve bitiş noktaları için bir sınırlama olmadığından, başlangıç ve bitiş parametrelerinin değerini **boş bırakarak** yalnızca adım parametresinin değerini belirttik.

Dilimleme yapıları bize bir karakter dizisini ters çevirme olanağı da tanır. Mesela elimizde şöyle bir **karakter dizisi** var:

```
>>> rakamlar = '0123456789'
```

Amacımız bu rakamları ters çevirerek '**9876543210**' gibi bir karakter dizisi elde etmek. İşte bu işlemi şu kodla yapabiliriz:

```
>>> rakamlar[::-1]
'9876543210'
```

Bu komutla, **rakamlar** adlı değişkenin sıfırıncı ögesinden sonuncu ögesine kadar olan aralığı **geriye doğru birer birer atlayarak** dilimlemiş olduk. Bu işlem netice itibariyle bize **ilgili değerin ters çevrilmiş hâlini** vermiş oldu. Eğer arzu ederseniz aynı karakter dizisini geriye doğru **ikişer ikişer atlayarak** da dilimleyebilirsiniz:

```
>>> rakamlar[::-2]
'97531'
```

Mesela bu yapıları kullanarak **isim** ve **soyisim** bilgilerinden kullanıcı adları üretebilirsiniz:

```
>>> isim = "ahmet"
>>> soyisim = "yaldız"
>>> kullanıcı_adı = isim[0] + "_" + soyisim
>>> kullanıcı_adı
'a_yaldız'
```

Bir **örnek** daha:

```
>>> isim ="mehmet"
>>> soyisim = "ballı"
>>> kullanıcı_adı = isim[0] + "_" + soyisim
>>> kullanıcı_adı
'm_ballı'
```

Burada dilimleme yöntemini kullanarak **isim** değişkeninin sıfırıncı ögesini, "_" karakter dizisini ve **soyisim** değişkeninin değerini birleştirdik. Böylece çeşitli yerlerde kullanıcı adı olarak kullanabileceğimiz karakter dizileri (**'a_yaldız'** ve **'m_ballı'**) elde ettik.

Bununla ilgili bir **örnek** daha verelim:

```
>>> il = 'ADANA'
>>> kod = '3576'
>>> numara = il[:3] + kod[::-1]
>>> numara
'ADA6753
```

Burada da `il` değişkeninin **ilk üç karakteri ile** kod değişkeninin ters çevrilmiş hâlini birleştirerek özel amaçlarla kullanabileceğimiz bir numara elde ettik.

Böylece Python'daki önemli bir konuyu daha incelemiş olduk. Daha önce de ifade ettiğimiz gibi bu tür yapıları anlamanın en iyi yolu bunlarla kendi kendinize bol bol denemeler yapmaktır.

METİN DÜZENLEYİCİ KULLANARAK PYTHON PROGRAMLAMA

7

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

Python kodlarını yazmak için istediğiniz herhangi bir metin düzenleyiciyi kullanabilirsiniz. Ancak içine yazılan kodları ayırt edebilen, bunları **farklı renklerde** gösterebilen bir metin düzenleyici ile yola çıkmak her bakımdan hayatınızı kolaylaştıracaktır.

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

GNU/Linux Sistemi

Eğer kullandığınız sistem, GNU/Linux'ta KDE masaüstü ortamı ise başlangıç düzeyi için **Kate** adlı metin düzenleyici yeterli olacaktır.

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

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

```
kate
```

yazmaktır.

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

```
gedit
```

komutunu veriyoruz.

Aslında metin içine kod yazmak, Python komut satırına kod yazmaktan çok farklı değildir. Şimdi, aşağıda verilen satırları, aynen aşağıda göründüğü şekilde **metin düzenleyici** içine yazalım:

```
a = “kalem”
b = “pergel”
c = “çikolata”

print(f”bir {a}, bir {b}, bir de {c} alacağım”)
```

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

```
cd ~/Desktop
```

aynı işi şu komutla da yapabilirsiniz:

```
cd $HOME/Desktop
```

Bazı Türkçe GNU/Linux sistemlerinde masaüstünü gösteren dizin "**`Desktop`**" yerine "**`Masaüstü`**" adını da taşıyabilmektedir. Öyle ise tabii ki vereceğiniz komutta "**`Desktop`**" kelimesini "**`Masaüstü`**" kelimesiyle **değiştirmeniz** gerekebilir.

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

```
python3 deneme.py
```

Eğer her şey yolunda gitmişse şu **çıktıyı** almamız lazım:

```
bir kalem, bir pergel, bir de çikolata alacağım
```

Bu arada, kodları Python'ın etkileşimli kabuğuna yazmakla metin düzenleyici içine yazmak arasındaki **önemli bir fark**, **`print()`** fonksiyonuyla ilgilidir. Etkileşimli kabukta **`print()`** fonksiyonunu kullansak da kullanmasak da **değerler ekrana basılacaktır**. Ama metin düzenleyicide **`print()`** fonksiyonu içinde belirtmediğimiz ifadeler ekrana **çıktı olarak verilmez**.

Bu durumu daha iyi anlayabilmek için kendi kendinize *denemeler yapmanızı tavsiye ederim*.

Gördüğünüz gibi **`python3 deneme.py`** komutuyla programlarımızı çalıştırabiliyoruz. Peki, ama acaba Python programlarını başa **`python3`** komutu eklemeden çalıştırmanın bir yolu var mı? Elbette var, ancak bunun için öncelikle bazı işlemler yapmamız gerekiyor. Başa **`python3`** komutu getirmeden programımızı çalıştırabilmek için öncelikle programımızın **ilk satırına şu kodu ekliyoruz**:

```
#!/usr/bin/env python3
Yani programımız tam olarak şöyle görünecek:
#!/usr/bin/env python3

a = “kalem”
b = “pergel”
c = “çikolata”

print(f”bir {a}, bir {b}, bir de {c} alacağım”)
```

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

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

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

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

Bu komutla '**`deneme.py`**' adlı dosyayı çalıştırılabilir ('**`executable`**') bir dosya hâline getirmiş oluyoruz. Artık komut satırında şu komutu vererek programımızı çalıştırabiliriz:

```
./deneme.py
```

Gelin, isterseniz yukarıda yaptığımız işlemleri biraz mercek altına alalım.

Biz, bir Python programını **`python3 program_adı.py`** komutuyla çalıştırdığımızda, baştaki **`python3`** ifadesi sayesinde bu programı Python programlama dilinin çalıştıracağı anlaşılıyor. Böylece işletim sistemimiz, programımızı doğru bir şekilde çalıştırabiliyor.

Ancak başa **`python3`** ifadesini getirmediğimizde, programımızın hangi programlama dili tarafından çalıştırılacağı anlaşılmıyor. İşte programımız, içine eklediğimiz **`#!/usr/bin/env python3`** satırı, programımızı hangi programlama dilinin çalıştıracağını ve bu programlama dilinin işletim sistemi içinde hangi konumda bulunduğunu (*env adlı bir betik aracılığıyla*) **işletim sistemimize haber veriyor**.

Eğer burada **`python3`** yerine **`python`** yazarsak ve eğer kullandığımız **GNU/Linux** dağıtımında **`python`** komutu Python'ın eski **2.x** sürümünü çalıştırıyorsa programımız, sistemde kurulu olan bu **eski Python sürümüyle** çalıştırılacaktır. O yüzden bu satıra, doğru Python sürümününü yazmaya dikkat ediyoruz.

İsterseniz neler olup bittiğini daha iyi anlayabilmek için basit bir deneme çalışması yapalım. **Metin düzenleyici içine sadece şu kodları yazalım**:

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

print(f"bir {a}, bir {b}, bir de {c} alacağım")
```

Dosyamızı **`deneme.py`** adıyla kaydedelim. **Şimdi, programımızı çalıştırmayı deneyelim**:

```
./deneme.py
```

Bu komutu verdiğimizde şu **hata mesajını** alacağız:

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

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

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

Şimdi, **tekrar deneyelim**:

```
./deneme.py
```

Bu defa da şu **hata mesajını** aldık:

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

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

Şimdi, program dosyamızı açıp **ilk satıra** şu ibareyi ekleyelim:

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

Artık programımızı çalıştırabiliriz:

```
./deneme.py
```

Bu işlemlerden sonra bu '**`deneme.py`**' dosyasının, isminin sonundaki **`.py`** uzantısını kaldırıp **`./deneme`** komutuyla da **programımızı çalıştırabiliriz**.

Peki, ya biz programımızı deneme komutuyla sadece ismini yazarak çalıştırmak istersek ne yapmamız gerekiyor?

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

İşletim sistemleri, bir programı çağırdığımızda söz konusu programı çalıştırabilmek için **bazı özel dizinlerin içine bakar**. Çalıştırılabilir dosyalar eğer bu özel dizinler içinde ise **işletim sistemi bu dosyaları bulur** ve **çalıştırılmalarını sağlar**. Böylece biz de bu programları doğrudan **isimleriyle** çağırabiliriz.

Şimdi, bu konuyu daha iyi anlayabilmek için birkaç **deneme** yapalım. Hemen bir konsol ekranı açıp şu komutu veriyoruz:

```
echo $PATH
```

Bu komutun çıktısı şuna benzer bir şey olacaktır:

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

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

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

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

Şimdi, konsol ekranında:

```
deneme
```

yazınca programımızın **çalıştığını** görmemiz lazım.

Gelelim Windows kullanıcılarına...

Windows Sistemi

Eğer Windows kullanıyorsanız Python kodlarınızı, **IDLE** adlı metin düzenleyiciye yazabilirsiniz. IDLE'a ulaşmak için "**Başlat / Programlar / Python / IDLE**" yolunu takip ediyoruz. Ayrıca Windows'un arama çubuğuna "**idle**" yazarak da IDLE'a ulaşabilirsiniz.

IDLE'ı çalıştırdığımızda gördüğümüz **>>>** işaretinden de anlayabileceğimiz gibi bu ekran, aslında Python'ın etkileşimli kabuğunun ta kendisidir...

Dilerseniz, etkileşimli kabukta yaptığınız işlemleri bu ekranda da yapabilirsiniz. Ama şu anda bizim **amacımız etkileşimli kabukla oynamak değil**. Biz, Python programlarımızı yazabileceğimiz bir metin düzenleyici arıyoruz. Burada "**File**" menüsü içindeki "**New File**" düğmesine tıklayarak **boş bir sayfa** açıyoruz. İşte Python kodlarını yazacağımız yer burası. Şimdi, bu boş sayfaya şu kodları ekliyoruz:

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

print(f"bir {a}, bir {b}, bir de {c} alacağım")
```

Kodlarımızı yazdıktan sonra yapmamız gereken şey dosyayı bir yere **kaydetmek** olacaktır. Bunun için "**File / Save as**" yolunu takip ederek dosyayı '**deneme.py**' adıyla masaüstüne kaydediyoruz. Dosyayı kaydettikten sonra "**Run/Run Module**" yolunu takip ederek veya doğrudan **F5** tuşuna basarak yazdığımız programı çalıştırabiliriz. Eğer programınızı IDLE üzerinden değil de doğrudan **MS-DOS** komut satırını kullanarak çalıştırmak isterseniz şu işlemleri yapın:

Öncelikle daha önce gösterdiğimiz şekilde klavyede Windows tuşuna ve **R** harfine birlikte bastığınızda açılan pencereye '**cmd**' yazıp **ENTER** düğmesine basarak Windows'un komut satırına ulaşın (*Ama Python komut satırını çalıştırmayın*).

Şu komutu vererek masaüstüne, yani dosyayı kaydettiğiniz yere gelin:

```
cd %userprofile%/desktop
```

Masaüstüne geldikten sonra şu komutlardan herhangi birini vererek **programınızı çalıştırabilirsiniz**:

```
py deneme.py
```

veya:

```
python deneme.py
```

Eğer her şey yolunda gitmişse şu **çıktıyı** almamız lazım:

```
bir kalem, bir pergel, bir de çikolata alacağım
```

Böylece Python programlama dilinde ilerlememizi kolaylaştıracak **en temel bilgileri** edinmiş olduk. Bu temel bilgileri cebimize koyduğumuza göre artık gönül rahatlığıyla ileriye bakabiliriz.

STANDART KÜTÜPHANEYE GİRİŞ

8

Buraya gelene kadar, Python'ın temellerine ilişkin epey bilgi edindik. Bu arada çok işlevli olmasa da bazı ufak tefek kodlar da yazdık. Ama dilerseniz gelin, şimdi gerçekten **az çok işe yarar** bir şeyler yazalım. Mesela *işletim sistemimiz hakkında birtakım bilgiler veren bir program üretelim*.

Bunun için öncelikle Python'ın "**standart kütüphane**" adı verilen ve çok farklı sahalarda işlerimizi hızlandırıp kolaylaştıracak yardımcı araçlar barındıran "**modül deposundan**" üstünkörü de olsa söz etmemiz gerekiyor.

Standart kütüphaneyi, bir program yazarken ihtiyaç duyacağınız her türlü aracı içinde barındıran, oldukça **geniş bir yazılım deposu** olarak düşünebilirsiniz. Bu depo içindeki her bir yazılıma "**modül**" adı verilir.

Python'ın standart kütüphanesine *https://docs.python.org/3/library/* adresinden erişebilir, bu kütüphanede hangi modüllerin bulunduğunu inceleyebilirsiniz.

Gelin, isterseniz standart kütüphane içindeki modüllere, Python programları içinden nasıl ulaşabileceğimize dair basit bir **örnek** verelim.

Geçen bölümün başında ifade ettiğimiz gibi özellikle **küçük kod parçaları** yazıp bunları denemek için **Python komut satırı** mükemmel bir ortamdır. O yüzden biz de şimdi, çeşitli deneme çalışmaları yapmak için hemen Python'ın etkileşimli kabuğunu açalım ve orada şu komutu verelim:

```
>>> import platform
```

Bu komutla, **`platform`** adlı bir standart kütüphane modülünü "**içe aktarmış**", yani tüm içeriğini programımıza **dâhil etmiş olduk**. Böylece, bu modül içeriğindeki fonksiyon ve niteliklerden yararlanabileceğiz.

Python'ın standart kütüphanesinde yüzlerce faydalı modül bulunur ve işin güzel tarafı herhangi bir **ilave yazılım kurmamıza gerek olmadan** bütün bu modüllere her an kolayca erişebiliriz. İşte **`platform`** adlı modül de bunlardan biridir. Bu modül, kendi içinde pek çok faydalı **fonksiyon** ve **değişken** (*yani nitelik*) barındırır.

Herhangi bir modülün içeriğini (*o modülü import komutuyla içe aktardıktan sonra*) **`dir()`** adlı bir fonksiyondan yararlanarak sorgulayabiliriz.

Mesela biraz önce **`import platform`** komutuyla içe aktardığımız **`platform`** modülünün içeriğini kontrol edelim:

```
>>> dir(platform)

['_WIN32_CLIENT_RELEASES', '_WIN32_SERVER_RELEASES', '__builtins__',
'__cached__ ', '__copyright__', '__doc__', '__file__', '__loader__',
'__name__', '__package_ _', '__spec__', '__version__',
'_comparable_version', '_component_re', '_default _architecture',
'_follow_symlinks', '_ironpython26_sys_version_parser', '_ironpy
thon_sys_version_parser', '_java_getprop', '_libc_search', '_mac_
ver_xml',
'_nod e', '_norm_version', '_platform', '_platform_cache',
'_pypy_sys_version_parser', '_sys_version', '_sys_version_cache',
'_sys_version_parser', '_syscmd_file', '_ syscmd_uname', '_syscmd_
ver',
'_uname_cache', '_ver_output', '_ver_stages', 'architecture',
'collections', 'java_ver', 'libc_ver', 'mac_ver', 'machine', 'node',
'os',
```

```
'platform', 'processor', 'python_branch', 'python_build', 'python_
compile
r', 'python_implementation', 'python_revision', 'python_version',
'python_versio n_tuple', 're', 'release', 'sys', 'system', 'system_
alias',
'uname', 'uname_result', 'version', 'win32_edition', 'win32_is_iot',
'win32_ver']
```

Bu liste içinde özellikle **alt çizgi** işareti ile başlamayan **isimler** ile ilgileniyoruz. Bu isimlerin her biri önemli bir fonksiyona veya niteliğe karşılık geliyor.

Mesela bu listede **system** adlı bir isim görüyoruz. Bu ismi şu şekilde kullanabiliriz:

```
>>> platform.system()
'Linux'
```

system bir fonksiyon olduğu için bu fonksiyonun sonuna **parantez işaretlerini koymayı unutmuyoruz**. Ayrıca modül adı (**platform**) ile fonksiyon adını (**system**) birbirinden **nokta işareti** ile ayırdığımızı da gözden kaçırmayın.

Yukarıdaki komut her işletim sisteminde **farklı çıktı** verir. Mesela GNU/Linux'ta bu komut '**Linux**' çıktısı verirken Windows'ta '**Windows**', Mac'te ise '**Darwin**' **çıktısını verir**.

Bu fonksiyonu kullanarak yazdığımız programlarda, **farklı işletim sistemlerine yönelik işlemler** yapabiliriz.

platform adlı standart kütüphane modülü, aynı zamanda işletim sistemimizdeki Python sürümünü de denetleyebilmemizi sağlar. Bu iş için modülün **python_version()** adlı fonksiyonundan faydalanabiliriz.

```
>>> platform.python_version()
'3.8.3'
```

Bu komut da çalıştırıldığı bilgisayardaki Python sürümüne göre **farklı çıktı** verecektir.

Yukarıdaki benzer bir çıktıyı ayrıca modül içindeki **`python_version_tuple()`** adlı bir fonksiyonla da elde edebiliriz. Bu fonksiyon da bize sistemimizdeki **Python sürümünü** verecek, ancak bu defa çıktının tipi **biraz daha farklı** olacaktır:

```
>>> platform.python_version_tuple()
('3', '8', '3')
```

Burada Python'ın büyük, küçük ve minik sürüm numaralarını **parantez içinde birbirinden virgül** ile ayrılmış ögeler hâlinde görüyoruz. Teknik olarak ifade etmemiz gerekirse birbirinden virgülle ayrılan, parantez içindeki ögelere "**demet**" (*tuple*) adı verilir. Demetler de tıpkı karakter dizileri ve sayılar gibi bir **veri tipi**dir. Bu veri tipinden, ilerleyen bölümlerde ayrıntılı bir şekilde söz edeceğiz. Ama biz şimdilik bunu bir kenara bırakıp yolumuza devam edelim.

Dediğimiz gibi **`platform.python_version_tuple()`** komutu bize Python'ın *büyük*, *küçük* ve *minik* sürüm numaralarını içeren bir **demet** veriyor.

Daha önce öğrendiğimiz yöntemleri kullanarak bu demetin ögelerini tek tek alabiliriz:

```
>>> sürüm = platform.python_version_tuple()
```

Öncelikle kullanım kolaylığı açısından **`platform`** modülünün **`python_version_tuple()`** fonksiyonunu, **`sürüm`** adlı değişkene atadık. Bu değişken artık Python'ın büyük, küçük ve minik sürüm numaralarını bir demet hâlinde tutuyor:

```
>>> sürüm
('3', '8', '3')
```

Şimdi, bu demetin ögelerini tek tek "**dilimleyelim**":

```
>>> büyük = sürüm[0]
>>> küçük = sürüm[1]
>>> minik = sürüm[2]
```

Demetin sıfırıncı ögesi **büyük sürüm numarasını**, birinci ögesi **küçük sürüm numarasını**, ikinci ögesi ise **minik sürüm numarasını** gösteriyor:

```
>>> büyük
'3'

>>> küçük
'8'

>>> minik
'3'
```

İstersek bu dilimleme işlemini biraz **sadeleştirebiliriz**:

```
>>> sürüm = platform.python_version_tuple()
>>> büyük, küçük, minik = sürüm[0], sürüm[1], sürüm[2]
>>> büyük
3

>>> küçük
8

>>> minik
3
```

Hatta bu işlemi **daha da sadeleştirmenin** bir yolunu biliyoruz:

```
>>> büyük, küçük, minik = platform.python_version_tuple()
>>> büyük
3

>>> küçük
8

>>> minik
3
```

Burada yerine göre **yıldızlı ifadeler** de işimize yarayabilir:

```
>>> büyük, *gerisi = platform.python_version_tuple()
>>> büyük
'3'

>>> gerisi
['8', '3']
```

Bu şekilde büyük sürüm numarasını, **sürüm numaralarının geri kalanından ayırmış olduk**.

Peki, acaba bu çıktıyı dilimleme yöntemiyle de alabilir miyiz?

Dikkatlice bakın:

```
>>> sürüm = platform.python_version_tuple()
>>> büyük, gerisi = sürüm[0], sürüm[1:]
>>> büyük
'3'

>>> gerisi
('8', '3')
```

Burada da sürüm değişkeninin sıfırıncı ögesini **`büyük`** adlı değişkene, birinci ögesinden itibaren geri kalan tüm ögeleri ise **`gerisi`** adlı değişkene atadık.

Bu yöntemlerden hangisini tercih edeceğiniz tamamen durumun neyi gerektirdiğine ve bazen de sizin isteğinize kalmış. **Uygun olan veya size kolay gelen yolu takip edebilirsiniz.**

Yıldızlı ifadelere kıyasla dilimleme yönteminin tek farkı, yıldızlı ifadelerde **`gerisi`** adlı değişkenin bir **liste** olarak, dilimleme yönteminde ise bu değişkenin bir **demet** olarak gelmesidir. İlerleyen derslerde **`liste`** ve **`demet`** adlı veri tiplerini öğrendiğimizde bu farkın ne anlama geldiğini daha iyi anlayacağız.

Kullandığımız sistemle ilgili **birden fazla bilgiyi bir arada veren** bir başka öge ise **`uname()`** adlı fonksiyondur. Bunu da şöyle kullanıyoruz:

```
>>> platform.uname()
```

Bu da kullandığınız işletim sistemine göre şuna benzer bir **çıktı** verecektir:

```
uname_result(system='Windows', node='ozgul', release='10',
version='10.0.17763', machine='AMD64',
processor='Intel64 Family 6 Model 58
Stepping 9, GenuineIntel')
```

Burada kullandığımız işletim sistemine ilişkin *isim, sürüm, mimari ve işlemci* gibi bilgileri elde edebiliyoruz. **Buradaki her bir ögeye şöyle ulaşabilirsiniz**:

```
>>> un = platform.uname()
>>> un.system
'Windows'

>>> un.release
'10'

>>> un.machine
'AMD64'

>>> un.processor
'Intel64 Family 6 Model 58 Stepping 9, GenuineIntel'
```

Tıpkı **`python_version_tuple()`** fonksiyonunu anlatırken yaptığımız gibi burada da her bir ögeyi **farklı bir değişkene atayabiliriz**:

```
>>> system, node, release, version, machine, processor = platform.
uname()
>>> system
'Windows'

>>> node
'ozgul'

>>> release
'10'

>>> version
'10.0.17763'

>>> machine
'AMD64'

>>> processor
'Intel64 Family 6 Model 58 Stepping 9, GenuineIntel'
```

Burada ayrıca **yıldızlı ifadeleri** kullanmak da işimize yarayabilir. Mesela **`platform.uname()`** çıktısındaki ilk ögeyi **tek başına**, geri kalanı ise **grup hâlinde** almak istiyor olabiliriz:

```
>>> sistem, *ayrıntı = platform.uname()
>>> sistem
'Windows'

>>> ayrıntı
['ozgul', '10', '10.0.17763', 'AMD64', 'Intel64 Family 6
Model 58 Stepping 9, GenuineIntel']
```

Aynı çıktıyı, **dilimleme** yöntemiyle de elde edebileceğimizi artık gayet iyi biliyorsunuz:

```
>>> un = platform.uname()
>>> sistem, ayrıntı = un[0], un[1:]
>>> sistem
'Windows'

>>> ayrıntı
('ozgul', '10', '10.0.17763', 'AMD64', 'Intel64 Family 6
Model 58 Stepping 9, GenuineIntel')
```

Yukarıda **dir(platform)** komutuyla elde ettiğimiz çıktıda görünen isimleri tek tek deneyerek bunların **ne tür çıktılar** verdiğini inceleyebilirsiniz. Ama biz şimdi, isterseniz bu modülü kullanarak *minik bir program* yazalım. Öncelikle **boş bir metin düzenleyici** sayfası açalım ve oraya şunları yazalım:

```
import platform

print("işletim sistemi adı:", platform.system())
print("işletim sistemi sürümü: ", platform.release())
print("işlemci mimarisi", platform.machine())
print("işlemci:", platform.processor())
print("python sürümü:", platform.python_version())
```

Burada ilk satırda gördüğümüz **import platform** komutu, **platform** adlı modülü tüm içeriğiyle birlikte programımıza dâhil etmemizi, yani **platform** modülünü **içe aktarmamızı sağlıyor**.

Daha sonra sırasıyla **işletim sistemimizin adını**:

```
platform.system()
```

sürümünü:

```
platform.release()
```

mimarisini:

```
platform.machine()
```

işlemcisini:

```
platform.processor()
```

ve sistemimizdeki Python'ın sürümünü:

```
platform.python_version()
```

veren fonksiyonları çalıştırıyoruz.

Bu programı çalıştırdığımız işletim sistemine bağlı olarak şuna benzer bir **çıktı** elde edeceğiz:

```
işletim sistemi adı: Windows
işletim sistemi sürümü:  10
işlemci mimarisi AMD64
işlemci: Intel64 Family 6 Model 58 Stepping 9, GenuineIntel
python sürümü: 3.8.3
```

Elbette yukarıdaki kodları yazmanın tek yolu bu değildir. Eğer isterseniz bu kodları şöyle de yazabilirsiniz:

```
import platform

sistem = platform.system()
is_sürüm = platform.release()
mimari = platform.machine()
işlemci = platform.processor()
py_sürüm = platform.python_version()

print("işletim sistemi adı:", sistem)
print("işletim sistemi sürümü: ", is_sürüm)
print("işlemci mimarisi", mimari)
print("işlemci:", işlemci)
print("python sürümü:", py_sürüm)
```

Burada **`platform`** modülünün ilgili fonksiyonlarını önce **ayrı ayrı değişkenlere atadık**. Daha sonra bu değişken adlarını **`print()`** fonksiyonuna **argüman** olarak gönderdik.

Hatta eğer isterseniz bu değişkenleri, **`f`**'li karakter dizileri içine de **gömebilirsiniz**:

```
import platform

sistem = platform.system()
is_sürüm = platform.release()
mimari = platform.machine()
işlemci = platform.processor()
py_sürüm = platform.python_version()

print(f"işletim sistemi adı: {sistem}")
print(f"işletim sistemi sürümü: {is_sürüm}")
print(f"işlemci mimarisi: {mimari}")
print(f"işlemci: {işlemci}")
print(f"python sürümü: {py_sürüm}")
```

Hangi tarzı kullanacağınız tamamen size kalmış. Hangisi size daha pratik geliyorsa onu kullanabilirsiniz.

Böylece kullandığımız **işletim sistemi** ve **Python sürümü** hakkında ayrıntılı sayılabilecek bilgiler veren minik bir program yazmış olduk. İlerleyen derslerde hem modüllerden hem de **içe aktarma** konusundan daha ayrıntılı bir şekilde söz edeceğiz. Biz şimdilik, bu bölüme burada bir nokta koyup bir sonraki konumuza geçelim.

KULLANICIYLA İLETİŞİM: VERİ ALIŞ-VERİŞİ

9

Bu bölümde yine Python'daki **çok önemli** bir konudan söz edeceğiz. Konumuz **kullanıcıyla iletişim**.

Dikkat ettiyseniz şimdiye dek sadece tek yönlü bir programlama faaliyeti içinde bulunduk. Yani yazdığımız kodlar, kullanıcıyla **herhangi bir etkileşim içermiyordu**. Ama artık Python'da bazı temel şeyleri öğrendiğimize göre kullanıcıyla *nasıl iletişim kurabileceğimizi de öğrenmemizin vakti geldi*.

Bu bölümde, **kullanıcılarımızla nasıl veri alış-verişinde bulunacağımızı** göreceğiz.

Python'da kullanıcıdan birtakım veriler alabilmek, yani kullanıcıyla iletişime geçebilmek için `input()` adlı bir fonksiyondan faydalanılır.

`input()` fonksiyonu da tıpkı daha önce öğrendiğimiz `print()` ve `dir()` fonksiyonları gibi Python'ın **hazır fonksiyonlarından** biridir. Bu fonksiyonu da **herhangi bir ilave işlem yapmadan** kodlarımız içinde ihtiyacımız olan her an kullanabiliriz.

`input()` fonksiyonu kullanıcılarımızın **veri girmesine** imkân tanır. İsterseniz bu fonksiyonu tarif etmeye çalışmak yerine hemen bununla ilgili bir **örnek** verelim.

Öncelikle boş bir metin belgesi açalım. Şimdi, `input()` fonksiyonuyla kullanıcıdan bazı bilgiler alacağız.

Mesela kullanıcıya bir **parola sorup kendisine teşekkür edelim**...

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

Şimdi, bu belgeyi '**deneme.py**' ismiyle kaydediyoruz. Daha sonra bir konsol ekranı açıp programımızın **kayıtlı olduğu dizine geçerek** ve **kullandığımız işletim sistemine uygun olan** komutu kullanarak programımızı çalıştırıyoruz:

GNU/Linux:

```
python3 deneme.py
```

Windows:

```
py deneme.py
```

...veya:

```
python deneme.py
```

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

- Program ilk çalıştırıldığında kullanıcıya **parola sorsun**
- Kullanıcı parolasını girdikten sonra programımız **kullanıcıya teşekkür etsin**
- Bir sonraki satırda kullanıcı tarafından girilen bu **parola ekrana yazdırılsın**
- Kullanıcı daha sonraki satırda parolanın **yanlış olduğu konusunda uyarılsın**

Şimdi, **input()** fonksiyonuyla kullanıcıya **parolasını soracağız**. Ama isterseniz bu **input()** fonksiyonunu **bir değişkene atayalım**:

```
parola = input("Lütfen parolanızı girin:")
```

Böylece **input()** fonksiyonu yardımıyla kullanıcıdan aldığımız veriyi **parola** adlı değişkene atadığımız için bu veriyi daha sonra kullanabileceğiz.

Şimdi, kullanıcıya **teşekkür ediyoruz**:

```
print("Teşekkürler!")
```

Kullanıcı tarafından girilen parolayı ekrana yazdırmak için şu satırı **ekliyoruz**:

```
print("Girdiğiniz parola: ", parola)
```

Biraz önce **`input()`** fonksiyonunu neden bir değişkene atadığımızı daha iyi anladığınızı zannediyorum. Bu sayede **`parola`** değişkenini doğrudan **`print()`** fonksiyonu içinde kullanarak kullanıcının yazdığı parolayı **ekrana dökebiliyoruz**.

Şimdi de kullanıcıya, **parolasının yanlış olduğunu** bildireceğiz:

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

Programımızın son hâli şöyle olacak:

```
parola = input("Lütfen parolanızı girin: ")

print("Teşekkürler!")
print("Girdiğiniz parola:", parola)
print("Ne yazık ki doğru parola bu değil.")
```

İsterseniz son satırı **`f`**'li **karakter dizileriyle** yazabilirsiniz:

```
print(f"Ne yazık ki doğru parola {parola} değil.")
```

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

Eğer arzu ederseniz değişkeni **karakter dizisi** içine gömmek yerine **`print()`** fonksiyonu içinde **ayrı argümanlar** hâlinde de belirtebilirsiniz:

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

Burada **`print()`** fonksiyonunu tek bir karakter dizisi hâlinde yazmak yerine, bu fonksiyonu **üç ayrı argümanla** birlikte çağırdık:

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

Hangi yöntemi kullanacağınız *tamamen sizin arzunuza kalmış*. Burada hangi yöntemi daha kolay buluyorsanız onu kullanabilirsiniz.

Aynı şekilde eğer istersek **`print("Girdiğiniz parola:", parola)`** satırını da şöyle yazabileceğimizi biliyorsunuz:

```
print(f"Girdiğiniz parola: {parola}")
```

Burada ise **`print()`** fonksiyonunu **iki farklı argümanla**:

```
"Girdiğiniz parola:"
parola
```

çağırmak yerine **`parola`** adlı değişkeni **doğrudan karakter dizisi içine gömdük**:

```
f"Girdiğiniz parola: {parola}"
```

Artık yukarıdaki **programı kaydedebilir** ve **işletim sisteminize uygun bir şekilde çalıştırabilirsiniz**.

Bu programı çalıştırdığınızda, **programınız sizi ilk önce şu cümleyle karşılayacak**:

```
Lütfen parolanızı girin:
```

Diyelim ki siz buna şu cevabı verdiniz:

```
asda
```

Bunun üzerine programınız size şu karşılığı verecek:

```
Teşekkürler!
Girdiğiniz parola: asda
Ne yazık ki doğru parola bu değil.
```

Gayet güzel...

`input()` fonksiyonunun nasıl kullanılması gerektiği konusunda artık en azından bir fikir sahibi olduğumuza göre **örnek kodlar** yazarak bilgilerimizi pekiştirebiliriz.

Dilerseniz geçen bölümde **dilimleme** konusunu anlatırken verdiğimiz **kullanıcı adı oluşturma örneğini `input()`** fonksiyonuna uyarlayalım:

```
isim = input("isminiz: ")
soyisim = input("soyisminiz: ")

kullanıcı_adı = isim[0] + '_' + soyisim

print("Bu bilgilere göre oluşturulan kullanıcı adınız: ", end='')
print(kullanıcı_adı)
```

Burada öncelikle kullanıcıdan **`isim`** ve **`soyisim`** bilgilerini aldık. Daha sonra kullanıcının isminin ilk harfini, '_' karakter dizisini ve kullanıcının soyismini birleştirerek bir **kullanıcı adı türettik**.

Daha sonra da elde ettiğimiz bu kullanıcı adı bilgisini **`print()`** fonksiyonunu kullanarak **uygun şekilde** ekrana yazdırdık. Burada ilk **`print()`** fonksiyonunda kullandığımız **`end`** parametresine dikkat edin. Bu **`print()`** fonksiyonunun çıktısı ile sonraki **`print()`** fonksiyonununun çıktısının **aynı satırda olması** için bu parametrenin değerini **`''`** olarak belirledik.

Ama elbette bu iki satır yerine şöyle bir kod da yazabilirdik:

```
isim = input("isminiz: ")
soyisim = input("soyisminiz: ")

kullanıcı_adı = isim[0] + '_' + soyisim

print("Bu bilgilere göre oluşturulan kullanıcı adınız: ", kullanıcı_
adı)
```

Duruma göre hangisini **daha uygun** buluyorsanız onu kullanabilirsiniz.

Gelin, isterseniz **`input()`** fonksiyonunun kullanımını gösteren **basit bir örnek** daha verelim:

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

print(a + b)
```

Şimdi, bu kodları çalıştıralım. Diyelim ki **ilk girilen değer** `5`, **ikinci değer** ise `4` olsun.

Ama o da ne? Program bize şöyle bir **çıktı** veriyor:

```
54
```

Gördüğünüz gibi değerler birbirleriyle toplanmadı, bunlar **sadece yan yana yazıldı**. Yani elde ettiğimiz çıktı **`5 + 4 = 9`** değil **`'5' + '4' = '54'`** oldu.

Hatırlarsanız buna benzer bir şeye, sayılardan bahsederken de karşılaşmıştık. O zaman şu **örnekleri** vermiştik:

```
>>> 25 + 50
75

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

Yukarıdaki örneklerde Python'ın **sayılara** ve **karakter** dizilerine **nasıl davrandığını** görüyoruz. Eğer Python iki adet sayıyla karşılaşırsa bu sayıları birbiriyle topluyor. Ama eğer iki adet karakter dizisiyle karşılaşırsa bu karakter dizilerini **birleştiriyor**. Bir sayı, bir de karakter dizisi ile karşılaşırsa (*sayılarla karakter dizileri arasında herhangi bir birleştirme işlemi veya aritmetik işlem yapılamayacağı için*) **hata veriyor**:

```
>>> "34" + 34

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Bütün bu açıklamalardan anlıyoruz ki **`input()`** fonksiyonu kullanıcının girdiği verileri **sayı olarak değil**, **karakter dizisi olarak görüyor**.

Bu durumu daha net kavrayabilmek için dilerseniz şöyle bir **örnek** verelim:

```
soru = "Bir sayı girin, o sayının karesini söyleyeyim:"

sayı = input(soru)

print(f"girdiğiniz sayı: {sayı}")
print(f"girdiğiniz sayının karesi: {sayı ** 2}")
```

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

```
Traceback (most recent call last):
  File "test.py", line 6, in <module>
    print(f"girdiğiniz sayının karesi: {sayı ** 2}")
TypeError: unsupported operand type(s) for ** or pow(): 'str' and
'int'
```

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

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

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

Gördüğünüz gibi biraz öncekiyle benzer bir **hata** aldık. İki karakter dizisini birbiriyle *çarpamaz*, *bölemez* ve *bu karakter dizilerini birbirinden çıkaramazsınız*. Çünkü **aritmetik işlemler ancak sayılar arasında** olur. Karakter dizileri ile aritmetik işlem yapılmaz. Yukarıdaki örneklerin düzgün çıktı verebilmesi için **karakter dizilerinin birer sayıya dönüştürülmesi** gerekir. *Peki, acaba bu dönüştürme işlemini nasıl yapacağız?*

Birazdan bu konuyu incelemek üzere bir sonraki bölüme geçeceğiz, ama isterseniz buraya kadar öğendiklerimizi kullanarak yine az çok işe yarar bir şeyler yazmaya çalışalım.

Hatırlarsanız geçen bölümde *Python'ın standart kütüphanesinden* söz etmiş ve bu kütüphanede yer alan **`platform`** adlı bir modülden söz etmiştik. Şimdi de yine standart kütüphanede yer alan **bir başka modülden** bahsedeceğiz. Bu modülün adı **`webbrowser`**.

`webbrowser` modülü, sistemdeki öntanımlı İnternet tarayıcı kullanılarak web sayfalarının kullanıcıya gösterilmesini sağlayan bir modüldür. Dilerseniz bu modülün nasıl kullanıldığını anlamak için yine Python'ın **etkileşimli kabuğunu açarak** birtakım *deneme çalışmaları* yapalım. Her zaman dediğimiz gibi **etkileşimli kabuk** bu tür deneme, alıştırma ve öğrenme çalışmaları için birebirdir.

Geçen bölümde de bahsettiğimiz gibi bir modülü kullanabilmek için öncelikle o modülü **içe aktarmamız** gerekiyor:

```
>>> import webbrowser
```

Bu komut bize **webbrowser** modülü içindeki tüm fonksiyonları kullanabilme imkânı veriyor.

Bakalım bu modül içinde hangi fonksiyonlar varmış:

```
>>> dir(webbrowser)

['BackgroundBrowser', 'BaseBrowser', 'Chrome', 'Chromium', 'Elinks',
'Error', 'Galeon', 'GenericBrowser', 'Grail', 'Konqueror',
'Mozilla',
'Netscape', 'Opera', 'UnixBrowser', 'WindowsDefault', '__all__',
'__builtins__', '__cached__', '__doc __', '__file__', '__loader__',
'__name__', '__package__', '__spec__', '_browsers ', '_lock',
'_os_preferred_browser', '_synthesize', '_tryorder', 'get', 'main',
'open',
'open_new', 'open_new_tab', 'os', 'register', 'register_X_browsers',
're
gister_standard_browsers', 'shlex', 'shutil', 'subprocess', 'sys',
'threading']
```

Bu listede yer alan isimler arasından özellikle **open()** fonksiyonu bizi ilgilendiriyor. Bu fonksiyon yardımıyla İnternet sayfalarını **Python programları** üzerinden açabileceğiz. Ufak bir deneme yapalım. Mesela *https://www.google.com* adresini açalım:

```
>>> webbrowser.open("www.google.com")
```

Bu komutu verip **ENTER** düğmesine bastığınızda İnternet tarayıcınızın otomatik bir şekilde açılarak *https://www.google.com* adresine gittiğini göreceksiniz.

Şimdi, bu bilgileri kullanarak yine ufak bir program yazalım. Programımız kullanıcıya aramak istediği kelimeyi soracak ve girilen kelimeyi içeren arama sonuçlarını *https://www.google.com* üzerinde gösterecek:

```
import webbrowser

sorgu = input("aradığınız kelime:")
adres = f"https://www.google.com/search?q={sorgu}"

webbrowser.open(adres)
```

Burada öncelikle `webbrowser` modülünü **içe aktarıyoruz**:

```
import webbrowser
```

Bu modülün bize sunduğu özelliklerden faydalanabilmek için **bu modülü içe aktarmamız şart**.

Daha sonra kullanıcıya **aramak istediği kelimeyi soruyoruz**:

```
sorgu = input("aradığınız kelime:")
```

Ardından kullanıcının yazdığı kelimeyi **Google'ın sorgu sayfasına yönlendiriyoruz**:

```
adres = f"https://www.google.com/search?q={sorgu}"
```

Burada **f**'li karakter dizilerinden yararlanarak kullanıcıdan aldığımız sorgu değişkenini *https://www.google.com/search?q=* adresinin **içine gömdük**.

Elbette her sitenin **url** yapısı birbirinden farklı olacaktır. İlgilendiğiniz web sitesinin **url** yapısının ne olduğunu araştırarak veya kendi kendinize denemeler yaparak bulabilirsiniz.

Bu kodun ardından **webbrowser** modülünün **open()** fonksiyonunu kullanarak adres değişkenini öntanımlı İnternet tarayıcısı üzerinden görüntüledik.

Dediğimiz gibi her web sitesinin url yapısı **birbirinden farklıdır**. Mesela *https://www.tureng.com* adresindeki **TURENG** sözlüğünde kelime arayan bir program yazıyor olsaydık **adres** değişkenini şu şekilde yazmamız gerekirdi:

```
adres = f"https://tureng.com/en/turkish-english/{sorgu}"
```

Benzer bir aramayı YouTube'da yapmak istediğimizde ise bu değişkeni şöyle yazmamız gerekecekti:

```
adres = f"https://www.youtube.com/results?search_query={sorgu}"
```

İlgilendiğiniz web sitesini birazcık kurcalayarak **doğru url yapısını** kolayca bulabilirsiniz.

Dilerseniz bu bölümü de burada kapatalım. Buraya kadar epey bilgi edindik. Bunları sindirebilmek için kendi kendinize *çeşitli deneme çalışmaları* yapabilirsiniz.

10 DÖNÜŞTÜRME İŞLEMLERİ

Programcılık maceranız sırasında, *verileri birbirine dönüştürmeniz* gereken durumlarla sıkça karşılaşacaksınız. Mesela pek çok durumda **bir sayıyı karakter dizisine** ve eğer mümkünse **bir karakter dizisini de sayıya** dönüştürmek zorunda kalacaksınız.

Şimdi, dilerseniz bu duruma çok basit bir **örnek** verelim:

```
>>> a = “23”
```

Bildiğiniz gibi yukarıdaki **a** değişkeni bir **karakter dizisi**dir. Dolayısıyla mesela bu karakter dizisini şöyle bir sayı ile **toplayamayız**:

```
>>> a = “23”
>>> b = 50
>>> a + b

Traceback (most recent call last):
  File “<stdin>”, line 1, in <module>
TypeError: can only concatenate str (not “int”) to str
```

a ve **b** değişkenleri arasında aritmetik işlemler yapabilmemiz için **her iki değerin de sayı olması** gerekir. O yüzden öncelikle bir karakter dizisi olan **a** değişkenini sayıya dönüştürmemiz gerekir.

Bunun için **`int()`** adlı bir fonksiyondan yararlanacağız. **`int()`** fonksiyonu da tıpkı **`print()`**, **`dir()`** ve **`input()`** fonksiyonları gibi Python'ın **hazır fonksiyonlarından biridir**:

```
>>> int(a)
23
```

Böylece '**23**' karakter dizisini **sayıya** çevirmiş olduk. Ancak tahmin edebileceğiniz gibi **her karakter dizisi sayıya çevrilemez**. **`int()`** fonksiyonu, yalnızca **sayı değerli karakter dizilerini sayıya dönüştürebilir**:

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

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

Gördüğünüz gibi '**`elma`**' karakter dizisi **sayı** olarak temsil edilemeyeceği için Python bize bir **hata mesajı** gösteriyor. Ama '**23**' gibi bir karakter dizisini sayı olmaktan çıkaran tek şey **tırnak işaretleri** olduğu için bu karakter dizisi sayı olarak temsil edilebiliyor.

Dilerseniz bu dönüştürme işlemlerinin önemini daha iyi anlayabilmek için geçen bölümün sonunda verdiğimiz **şu kodlara tekrar bakalım**:

```
soru = "Bir sayı girin, o sayının karesini söyleyeyim:"

sayı = input(soru)

print(f"girdiğiniz sayı: {sayı}")
print(f"girdiğiniz sayının karesi: {sayı ** 2}")
```

Bu kodları bu şekilde çalıştırdığımızda hata alacağımızı biliyoruz. Zira **`input()`** fonksiyonu bize **karakter dizisi** verecek ve karakter dizileri ile de **aritmetik işlem yapmamız mümkün olmayacaktır**. **`input()`** fonksiyonunun verdiği karakter dizilerini artık şu şekilde dönüştürmemiz gerektiğini biliyoruz:

```
soru = "Bir sayı girin, o sayının karesini söyleyeyim:"

sayı = int(input(soru))

print(f"girdiğiniz sayı: {sayı}")
print(f"girdiğiniz sayının karesi: {sayı ** 2}")
```

Burada yaptığımız tek değişiklik, ikinci satırda **`input()`** fonksiyonunu **`int()`** adlı dönüştürücü fonksiyonun parantezleri içine almaktan ibarettir. Böylece **`input()`** fonksiyonundan gelecek **karakter dizisini bir sayıya dönüştürmüş** oluyoruz.

Eğer isterseniz kullanıcıdan veri alma ve bu veriyi sayıya dönüştürme işlemlerini **tek satırda** da hâlledebilirsiniz. Yani yukarıdaki kodları isterseniz şöyle de yazabilirsiniz:

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

print(f"girdiğiniz sayı: {sayı}")
print(f"girdiğiniz sayının karesi: {sayı ** 2}")
```

Burada **`input()`** ve **`int()`** fonksiyonlarından gelen veriyi **tek hamlede `sayı`** adlı bir değişkene **gönderdik**.

Gelin, isterseniz yukarıda öğrendiğimiz bilgileri şu **örneğe** de uygulayalım:

```
ilk_sayı = int(input("İlk sayıyı girin: "))
ikinci_sayı = int(input("İkinci sayıyı girin: "))
toplam = ilk_sayı + ikinci_sayı
print(f"{toplam = }")
```

Gördüğünüz gibi burada yaptığımız şey çok basit. **`input()`** fonksiyonunu tümden **`int()`** fonksiyonu içine aldık. Yani teknik olarak ifade etmek gerekirse **`input()`** fonksiyonunu **`int()`** fonksiyonuna **argüman** olarak verdik:

```
int(input("İlk sayıyı girin: "))
```

Burada özellikle **kapanış tırnaklarını** ve **kapanış parantezlerini eksik yazmamaya özen gösteriyoruz**. Python'a yeni başlayanların en sık yaptığı hatalardan biri de **açılmış tırnakları ve açılmış parantezleri kapatmayı unutmalarıdır**.

İsterseniz bir **örnek** daha vererek ne ile karşı karşıya olduğumuzu daha iyi anlamaya çalışalım:

```
sayı = int(input('''Bir sayı girin. Ben size bu sayının
istediğiniz kuvvetini hesaplayayım: ''''))

kuvvet = int(input(f'''Şimdi de {sayı} sayısının kaçıncı
kuvvetini hesaplamak istediğinizi söyleyin: ''''))

print(f"{sayı} sayısının {kuvvet}. kuvveti {sayı ** kuvvet} olur.")
```

Burada yazdığımız kodların nasıl işlediğine dikkat etmenin yanı sıra, karakter dizilerini tanımlamak için **tek** veya **çift tırnak** yerine **üç tırnak** kullandığımıza da dikkat edin. Python'da özellikle uzun karakter dizileri tanımlamamız gerektiğinde **üç tırnak işareti son derece kullanışlı bir araç sunar**. Mesela program çıktısını zevkinize göre düzenleyebilmek için bu üç tırnak işaretlerinden faydalanabilirsiniz:

```
sayı = int(input('''
  Bir sayı girin. Ben size bu sayının
  istediğiniz kuvvetini hesaplayayım: ''''))

kuvvet = int(input(f'''
  Şimdi de {sayı} sayısının kaçıncı
  kuvvetini hesaplamak istediğinizi söyleyin: '''))

print(f'''
  {sayı} sayısının {kuvvet}. kuvveti {sayı ** kuvvet} olur.''')
```

Üç tırnak işaretleri, karakter dizilerini istediğimiz noktadan **bölüp** bir alt satıra geçebilmemize müsaade ediyor. Tek ve çift tırnaklarla bu işi bu şekilde yapamayız. Eğer aynı işi tek ve çift tırnaklarla yapmak istersek daha önce öğrendiğimiz **** ve **\n kaçış dizilerinden** yararlanmamız gerekir:

```
sayı = int(input('\
Bir sayı girin. Ben size bu sayının\n\
istediğiniz kuvvetini hesaplayayım: '))

kuvvet = int(input(f"\
Şimdi de {sayı} sayısının kaçıncı\n\
kuvvetini hesaplamak istediğinizi söyleyin: "))

print(f"\
{sayı} sayısının {kuvvet}. kuvveti {sayı ** kuvvet} olur.")
```

Gördüğünüz gibi karakter dizilerini tanımlamak için tek ve çift tırnaklar kullandığımızda **uzun bir satırı bölmek üzere alt satıra geçmek istiyorsak** tam alt satıra geçmek istediğimiz noktaya bir adet **\n** ve **\ işareti yerleştirip ENTER düğmesine basmamız gerekiyor**. Burada \ işaretini koyduktan sonra **herhangi bir boşluk** veya **karakter getirmemeye** özellikle dikkat etmemiz gerekiyor. Bu işareti koyar koymaz **ENTER** ile alt satıra geçmeliyiz:

```
print("Merhaba.\n\
Ben çok uzun bir karakter\n\
dizisiyim. Beni bölebilmek için\n\
kaçış dizilerinden yararlanmanız\n\
gerekiyor.")
```

Böylece, bir karakter dizisini **int()** fonksiyonu yardımıyla nasıl karakter dizisine çevirebileceğimizi öğrendik.

Peki, yukarıda yaptığımız şeyin tersi mümkün mü? Yani acaba bir sayıyı karakter dizisine çevirebilir miyiz? Bu sorunun yanıtı **evettir**. Bu işlem için de **`str()`** adlı bir fonksiyondan yararlanacağız:

```
>>> a = 23
>>> str(a)
'23'
```

Böylece **23** tam sayısını, "**23**" **karakter dizisine** dönüştürmüş olduk. Gördüğünüz gibi **`int()`** ve **`str()`** adlı fonksiyonlar yardımıyla karakter dizileri ve tam sayılar arasında **dönüştürme** işlemi yapabiliyoruz.

Eğer bir sayıyı veya sayı değerli karakter dizisini **kayan noktalı sayıya** dönüştürmek istersek de **`float()`** adlı bir fonksiyondan yararlanacağız:

```
>>> a = 23
>>> float(a)
23.0

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

Bütün bu bilgileri edindikten sonra, dilerseniz yine işe yarar bir şeyler yazmaya çalışalım. Geçen bölümlerde Python'daki **`platform`** ve **`webbrowser`** modüllerini incelemiştik. Bu bölümde ise **`datetime`** adlı bir modülden söz edelim.

Tıpkı **`platform`** ve **`webbrowser`** modülleri gibi **`datetime`** modülü de bir **standart kütüphane modülü** olup tarihlerle ilgili işlemler yapabilmemizi sağlayan birtakım **fonksiyon** ve **nitelikler** (*yani değişkenler*) barındırır.

Tıpkı daha önce öğrendiğimiz modüllerde olduğu gibi bu modülü de kullanabilmek için öncelikle bunu **`import`** komutuyla **içe aktarmamız** lazım:

```
>>> import datetime
```

İsterseniz bu modülün içeriğini de **`dir()`** fonksiyonuyla kontrol edebilirsiniz:

```
>>> dir(datetime)

['MAXYEAR', 'MINYEAR', '__builtins__', '__cached__', '__doc__', '__
file__',
'__loader__', '__name__', '__package__', '__spec__', 'date',
'datetime',
'datetime_CAPI', 'sys', 'time', 'timedelta', 'timezone', 'tzinfo']
```

Bu listede gördüğümüz **`datetime`** adlı sınıf, özellikle önemlidir.



'**Sınıf**' kavramından, ileride ayrıntılı bir şekilde söz edeceğiz. Biz şimdilik, buna takılmadan yolumuza devam edelim.

`datetime` modülü içindeki **`datetime`** adlı sınıf da kendi içinde oldukça faydalı birtakım **fonksiyon** ve **nitelikler** barındırır:

```
>>> dir(datetime.datetime)

['__add__', '__class__', '__delattr__', '__dir__', '__doc__', '__
eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__le__', '__lt__', '__ne__', '__
new__',
'__radd__', '__reduce__', '__reduce_ex__', '__repr__', '__rsub__',
'__setattr__', '__sizeof__', '__str__', '__sub__', '__
subclasshook__',
'astimezone', 'combine', 'ctime', 'date', 'day', 'dst', 'fold',
'fromisocalendar', 'fromisoformat', 'fromordinal', 'fromtimestamp',
'hour',
'isocalendar', 'isoformat', 'isoweekday', 'max', 'microsecond',
'min',
'minute', 'month', 'now', 'replace', 'resolution', 'second',
'strftime',
'strptime', 'time', 'timestamp', 'timetuple', 'timetz', 'today',
'toordinal', 'tzinfo', 'tzname', 'utcfromtimestamp', 'utcnow',
'utcoffset',
'utctimetuple', 'weekday', 'year']
```

Burada da özellikle **now()** adlı fonksiyon bizim şu anda ilgimizi çekiyor. Şimdi, **şu koda dikkatlice bakın**:

```
>>> şuan = datetime.datetime.now()
```

Bu kod yardımıyla **datetime** modülünün **datetime** sınıfının **now()** adlı fonksiyonunu **şuan** adlı değişkene atamış olduk.

Şimdi, **şuan** değişkeninin **değerini kontrol edelim**:

```
>>> şuan
datetime.datetime(2020, 1, 30, 10, 26, 0, 839841)
```

Bu çıktı ilk anda gözünüze *anlamsız* gelmiş olabilir. Ancak **şuan** değişkeninin verdiği **çıktı**, tam şu andaki **tarih ve saate ilişkin ihtiyacımız olan tüm bilgileri** içinde barındırıyor. Eğer arzu edersek bu değişkenin mevcut içeriğini de **dir()** fonksiyonuyla kontrol edebiliriz:

```
>>> dir(şuan)

['__add__', '__class__', '__delattr__', '__dir__', '__doc__', '__
eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__init_su bclass__', '__le__', '__lt__', '__ne__', '__
new__',
'__radd__', '__reduce__', '_ _reduce_ex__', '__repr__', '__rsub__',
'__setattr__', '__sizeof__', '__str__', ' __sub__', '__
subclasshook__',
'astimezone', 'combine', 'ctime', 'date', 'day', ' dst', 'fold',
'fromisocalendar', 'fromisoformat', 'fromordinal', 'fromtimestamp' ,
'hour',
'isocalendar', 'isoformat', 'isoweekday', 'max', 'microsecond',
'min',
'minute', 'month', 'now', 'replace', 'resolution', 'second',
'strftime',
'strptime', 'time', 'timestamp', 'timetuple', 'timetz', 'today',
'toordinal', 'tzinfo' , 'tzname', 'utcfromtimestamp', 'utcnow',
'utcoffset',
'utctimetuple', 'weekday' , 'year']
```

Burada çok önemli birtakım nitelikler görüyoruz.

Mesela şu anda **ayın hangi günündeyiz?**

```
>>> şuan.day
30
```

Hangi aydayız?

```
>>> şuan.month
1
```

Hangi yıldayız?

```
>>> şuan.year
2020
```

Saat kaç?

```
>>> şuan.hour
10
```

Dakika kaç?

```
>>> şuan.minute
26
```

Saniye kaç?

```
>>> şuan.second
0
```

Gördüğünüz gibi **`datetime`** modülü içindeki **`datetime`** adlı bir sınıfın **`now()`** fonksiyonuna ait olan **`day`**, **`month`**, **`year`**, **`hour`**, **`minute`** ve **`second`** adlı nitelikleri kullanarak sırasıyla **gün**, **ay**, **yıl**, **saat**, **dakika** ve **saniye** bilgilerini elde ettik.

Bu arada bir modül içindeki fonksiyonları çağırırken parantezleri kullandığımıza ama **nitelikleri çağırırken parantez kullanmadığımıza** dikkat edin. Modüller içindeki nitelikler, **değişkenlere çok benzer**. Değişkenleri nasıl *parantezsiz* kullanıyorsak nitelikleri de aynı şekilde *parantezsiz* olarak kullanıyoruz.

Artık bu bilgileri kullanarak mesela bugünün tarihini **gün**, **ay**, **yıl**, **saat** ve **dakika** şeklinde veren **basit bir program yazabiliriz**:

```
import datetime

şuan = datetime.datetime.now()

gün = şuan.day
ay = şuan.month
yıl = şuan.year
saat = şuan.hour
dakika = şuan.minute
saniye = şuan.second

print(f"{gün}/{ay}/{yıl} {saat}:{dakika}:{saniye}")
```

Bu programı çalıştırdığınızda o sırada tarih ve saat ne ise o görüntülenecektir.

Gayet güzel... Gelin, isterseniz aynı bilgileri kullanarak bir program daha yazalım. Bu kez programımız kullanıcıya **doğduğu yılı sorsun** ve **aldığı cevaba göre kullanıcının kaç yaşında olduğunu** söylesin.

```
import datetime

doğum_tarihi = int(input("doğum yılınızı girin: "))

şuan = datetime.datetime.now()

gün = şuan.day
ay = şuan.month
yıl = şuan.year
saat = şuan.hour
dakika = şuan.minute
saniye = şuan.second
```

```
print(f"{gün}/{ay}/{yıl} {saat}:{dakika}:{saniye}",
      f"itibariyle {yıl-doğum_tarihi} yaşındasınız!")
```

Burada kullanıcının doğum tarihi ile aritmetik işlem yapacağımız için **`input()`** ile gelen karakter dizisi tipindeki veriyi **`int()`** fonksiyonu yardımıyla **tam sayıya çevirdiğimize dikkat edin**:

```
doğum_tarihi = int(input("doğum yılınızı girin: "))
```

Kodların geri kalanı ise tamamen bildiğimiz şeylerden oluşuyor. Burada dikkatinizi çekmek istediğim tek nokta sondaki **`print()`** fonksiyonu:

```
print(f"{gün}/{ay}/{yıl} {saat}:{dakika}:{saniye}",
      f"itibariyle {yıl-doğum_tarihi} yaşındasınız!")
```

Burada **`print()`** fonksiyonu içinde yazdığımız cümlenin sağa doğru çirkin bir şekilde uzamaması için cümleyi **uygun bir yerinden böldüğümüzü** ve **`print()`** fonksiyonunu **tek argümanla değil**, **iki argümanla çalıştırdığımızı görüyorsunuz**:

```
f"{gün}/{ay}/{yıl} {saat}:{dakika}:{saniye}"
f"itibariyle {yıl-doğum_tarihi} yaşındasınız!"
```

Yukarıdaki **`print()`** fonksiyonunun içeriğini nasıl düzenleyeceğiniz tamamen size kalmış. Elinizdeki **verileri kullanıcıya nasıl sunacağınız tamamen sizin isteğinize bağlı**. Mesela bu fonksiyonun içeriğini şöyle de yazabilirdiniz:

```
print(f"\nTarih   : {gün}/{ay}/{yıl}",
      f"\nSaat    : {saat}:{dakika}:{saniye}",
      f"\nYaşınız : {yıl-doğum_tarihi}\n")
```

Burada yeni satır kaçış dizisini (**`\n`**) kullanarak **çıktıya istediğimiz biçimi nasıl verdiğimize dikkat edin**. Eğer bu kaçış dizisini kullanmasaydık veya farklı yerlerde kullansaydık çıktımızın nasıl olacağını görmek için *kendi kendinize denemeler yapın*.

Bu arada yukarıdaki satırları ayrı birer **`print()`** fonksiyonu şeklinde de tasarlayabilirdik:

```
print(f"Tarih   : {gün}/{ay}/{yıl}")
print(f"Saat    : {saat}:{dakika}:{saniye}")
print(f"Yaşınız : {yıl-doğum_tarihi}")
```

Bir programı yazmanın **çoğu zaman tek bir yolu yoktur**. Duruma ve ihtiyaçlarına göre kendinize has kod yazma yöntemleri geliştirebilirsiniz.

Böylece Python programlama dilinde dönüştürme işlemleri konusunu tamamlamış olduk. Artık Python'da bir başka önemli konuyu daha incelemeye geçebiliriz.

KAÇIŞ DİZİLERİ

11

Hatırlarsanız **`print()`** fonksiyonunu anlattığımız bölümde şöyle bir **örnek** vermiştik:

```
>>> print('İstanbul\'un semtleri')
```

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

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

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

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

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

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

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

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

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

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

Yukarıdaki örneklerden gördüğünüz gibi \ işareti, **tek** ve **çift** tırnak işaretleri ile birleşerek bu işaretleri, **karakter dizileri içinde farklı anlamlarda kullanmamızı sağlıyor**.

\ adlı kaçış dizisi **başka karakterlerle de birleşebilir**. Örneğin, önceki derslerimizde **print()** fonksiyonunun **sep** ve **end** argümanlarını anlatırken yeni satıra geçmemizi sağlayan **\n** adlı bir karakterden söz etmiştik. İşte bu da aslında bir başka **kaçış dizisi**dir. \ kaçış dizisini **n** harfiyle birleştirdiğimizde, **yeni satır** adlı bir kaçış dizisini elde ediyoruz. **\n** adlı kaçış dizisinin şöyle kullanıldığını hatırlıyorsunuz:

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

Bu kodları çalıştırdığımızda şöyle bir **çıktı** elde ediyoruz:

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

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

\ kaçış dizisinin, birleştiğinde özel bir anlam kazanmasını sağladığı bir başka karakter de '**t**' karakteridir. Bildiğiniz gibi '**t**' karakterinin görevi normalde ekrana '**t**' harfini basmaktır. Ama eğer biz bu karakteri \ işareti ile birleştirirsek ortaya **sekme** adlı bir kaçış dizisi çıkar.

Hemen bir **örnek** verelim:

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

Eğer arzu ederseniz sekme karakterini **print()** fonksiyonunun **sep** argümanı ile birlikte de kullanabilirsiniz:

```
>>> print("uzak...", "çok uzak", sep="\t")
uzak...	çok uzak
```

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

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

>>> print("uzak...", "çok uzak", sep="\t\t")
uzak...         çok uzak
```

Tam bu noktada size şöyle bir soru sormama izin verin: *Acaba aşağıdaki komut neden hata veriyor?*

```
>>> print("C:\dosyalar\nisan\tablolar\")

File "<stdin>", line 1
  print("C:\dosyalar\nisan\tablolar\")
                                     ^
SyntaxError: EOL while scanning string literal
```

veya şu komut neden böyle garip bir **çıktı** veriyor:

```
>>> print("C:\dosyalar\nisan\tablolar")
C:\dosyalar
isan    ablolar
>>>
```

Bu örnekler, **karakter dizileri içinde geçen bazı karakterlere karşı** neden dikkatli olmamız gerektiğini gayet net bir şekilde gösteriyor bize. Buradaki sorun, Python'ın karakter dizileri içinde geçen **\n**, **\t** ve **** karakterlerini birer **kaçış dizisi** olarak yorumlamasından kaynaklanıyor. Bu sorunu aşmak için '**nisan**' ve '**tablolar**' kelimelerinin ilk harflerinin kaçış dizisi olarak yorumlanmasına yol açan **** işaretini **çift** yazmamız gerekiyor:

```
>>> print("C:\dosyalar\\nisan\\tablolar\\")
C:\dosyalar\nisan\tablolar\
```

Peki, şuradaki sorun nedir?

```
>>> print("falanca\")

File "<stdin>", line 1
  print("falanca\")
                ^
SyntaxError: EOL while scanning string literal
```

Burada da sorun, karakter dizisini kapatan çift tırnaklardan önceki \ işaretinin, **kapanış tırnağını bloke ederek** Python'ın bu tırnağı görmesini engellemesi, böylece Python'ın sanki karakter dizisini **kapatan bir tırnak işareti yokmuş gibi davranmasına yol açmasıdır**.

Sözün özü, eğer bir karakter dizisi içinde herhangi bir şekilde \ işareti geçiyorsa bu işaretin **sağında** hangi harfin olduğuna çok dikkat etmeliyiz. Zira bu işaretin yan yana geldiği bazı harfler Python programlama dili açısından **özel bir anlam** ifade eder. Eğer bu tür denk gelişlere karşı dikkatli olmazsanız programlarınız hiç beklemediğiniz noktalarda **hata** verip çökebilir.

Elbette Python'daki kaçış dizileri yukarıdakilerden ibaret değildir, ama bunlar arasında en önemlileri yukarıda verdiklerimizdir. Dolayısıyla özellikle ilk aşamada burada verdiğimiz **kaçış dizilerini** öğrenmeniz işlerinizin büyük bölümünü hâlletmeniz açısından yeterli olacaktır.

Böylece üstünkörü de olsa kaçış dizilerini öğrenmiş olduk. Dilerseniz şimdi, yine *ufak tefek program yazma denemeleri yapalım*. Bir yandan da standart kütüphanedeki modülleri tanımaya devam edelim. Mesela bu bölümde de `random` adlı modülü ele alalım.

`random` adlı modül, bize rastgele sayılar veya karakterler üretme imkânı verir. Mesela bu modülü kullanarak bir karakter dizisi içinden **rastgele harf** seçebiliriz:

```
>>> import random
>>> random.choice("abcçdefgğhıijklmnoöprsştuüvyz")
z
```

Bu kodu her çalıştırdığınızda, **random** modülünün **choice()** adlı fonksiyonuna argüman olarak verdiğiniz karakter dizisi içinden **bir harf rastgele seçilerek** size gösterilecektir.

random modülünün bir de **randrange()** adlı bir fonksiyonu bulunur. Bu fonksiyonu kullanarak istediğiniz bir sayı aralığı içinden rastgele sayılar seçebilirsiniz:

```
>>> random.randrange(1, 100)
32
```

Burada 1 ile 100 arasında rastgele bir sayı seçtik. Gelin, isterseniz bu fonksiyonun kullanımını **örneklendiren** minik bir program yazalım:

```
import random

print("""Bu programı kullanarak bir sayı aralığı içinden
rastgele sayılar seçebilirsiniz.""")

print("""Programın üreteceği rastgele sayı hangi sayılar
arasında olsun?""")

ilk_sayı = int(input("İstediğiniz aralığın ilk sayısını girin: "))
ikinci_sayı = int(input("İstediğiniz aralığın ikinci sayısını girin:
"))

print(f"{ilk_sayı} ile {ikinci_sayı} arasında",
"rastgele bir sayı üretiliyor. Lütfen bekleyiniz...")

print("Üretilen sayı: ", random.randint(ilk_sayı, ikinci_sayı))
```

Burada ilk olarak **random** adlı modülü **import** komutuyla içe aktardık. **Aksi hâlde bu modülü programımızda kullanmamız mümkün olmayacaktır**:

```
import random
```

Daha sonra, programımızın nasıl kullanılacağı konusunda **birtakım bilgiler** veriyoruz:

```
print("""Bu programı kullanarak bir sayı aralığı içinden
rastgele sayılar seçebilirsiniz.""")

print("""Programın üreteceği rastgele sayı hangi sayılar
arasında olsun?""")
```

Kullanıcılarımıza **bu tür mesajlar gösterebilmek için `print()`** fonksiyonundan yararlanmamız gerektiğini biliyoruz.

Ardından kullanıcıdan **iki adet sayı** istiyoruz:

```
ilk_sayı = int(input("İstediğiniz aralığın ilk sayısını girin: "))
ikinci_sayı = int(input("İstediğiniz aralığın ikinci sayısını girin:
"))
```

Bildiğiniz gibi **`input()`** fonksiyonu bize **her zaman** bir **karakter dizisi** veriyor. **`random`** modülünün **`randint()`** fonksiyonu ise bizden **iki adet sayı** girmemizi bekliyor. O yüzden **`int()`** fonksiyonunu kullanarak **`input()`** fonksiyonundan gelen karakter dizilerini **birer sayıya** dönüştürüyoruz.

Artık **`randint()`** fonksiyonunu kullanarak kullanıcıdan aldığımız sayılar arasından **rastgele bir sayı üretilmesini sağlayacak** kodlarımızı yazabiliriz:

```
print("üretilen sayı: ", random.randint(ilk_sayı, ikinci_sayı))
```

Böylece **`random`** modülü üzerinden hem yeni bilgiler edinmiş hem de şimdiye kadar öğrendiğimiz bilgileri pekiştirmiş olduk. Dolayısıyla artık gönül rahatlığıyla bir sonraki bölüme geçebiliriz.

KOŞULA BAĞLI DURUMLAR

12

Python'la program yazma maceramıza ilk olarak **`print()`** fonksiyonu ile başladık. Bu fonksiyon bize **tek yönlü** birtakım programlar yazabilme olanağı sağladı:

```
>>> print("Guido Van Rossum")
```

Bu fonksiyon yardımıyla her türlü veriyi komut satırına **çıktı** olarak verebiliyoruz. Ancak bu kod tek yönlü olduğu için **herhangi bir kaynaktan veri almamıza izin vermiyor**.

Daha sonra ise **`input()`** fonksiyonunu öğrenmemiz sayesinde programımızı kullanan kişilerden **birtakım veriler alabilecek duruma geldik**:

```
>>> input("Bir değer girin: ")
```

Burada **`input()`** fonksiyonu aracılığıyla kullanıcıdan bir değer girmesini istiyoruz. Ancak kullanıcıdan gelen bu değerin *türüne*, *yapısına* veya *niteliğine* göre programımızın akışını nasıl değiştireceğimizi henüz bilmiyoruz.

Örneğin bu fonksiyon yardımıyla kullanıcıdan bir **parola** girmesini isteyebiliriz. Fakat şu ana kadarki bilgilerimiz bu **parolanın doğru mu yoksa yanlış mı** olduğunu kontrol etmemize **izin vermiyor**.

Dolayısıyla parola doğru olduğunda **bir işlem**, yanlış olduğunda ise **başka bir işlem yapamıyoruz**. Yani girilen veriye göre programımızın akışını değiştiremiyoruz.

İşte bu bölümde, herhangi bir kaynaktan elde ettiğimiz bir verinin değerine göre programımız içinde **farklı işlemleri nasıl yapabileceğimizi**, yani program akışını nasıl yönlendireceğimizi öğreneceğiz.

Python'da bir programın akışını birtakım "**koşul deyimleri**" yardımıyla yönlendiriyoruz.

İsterseniz tüm bu anlattıklarımızın ne demek olduğunu daha somut bir örnekle açıklayalım.

Diyelim ki Gmail'den, aldığınız e-posta hesabınıza gireceksiniz. Gmail'in ilk sayfasında size bir **kullanıcı adı** ve **parola** sorulur. Siz de kendinize ait kullanıcı adını ve parolayı sayfadaki kutucuklara yazarsınız. Eğer yazdığınız kullanıcı adı ve parola doğruysa hesabınıza erişebilirsiniz. Yok eğer kullanıcı adınız ve parolanız doğru değilse **hesabınıza erişemezsiniz**. Yani e-posta hesabınıza erişmeniz kullanıcı adı ve parolayı **doğru girme koşuluna** bağlıdır.

Ya da şu örneği düşünelim: Diyelim ki Ubuntu'da konsol ekranından güncelleme işlemi yapacaksınız. `sudo apt-get upgrade` komutunu verdiğiniz zaman, güncellemelerin listesi size bildirilecek, bu güncellemeleri yapmak isteyip istemediğiniz size sorulacaktır. Eğer **evet** cevabı verirseniz güncelleme işlemi başlayacak, eğer **hayır** cevabı verirseniz güncelleme işlemi başlamayacaktır. Yani güncelleme işleminin başlaması kullanıcının evet cevabı vermesi koşuluna bağlıdır.

Biz de şimdi, Python'da bu tip koşullu durumların nasıl oluşturulacağını öğreneceğiz. Dediğimiz gibi bu iş için birtakım **koşul deyimlerinden** yararlanacağız. Python'da bu tür işlemler için kullanabileceğimiz üç tane koşul deyimi var: `if`, `else` ve `elif`

İlk önce `if` deyimi ile başlayalım.

IF

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

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

Bunun anlamı şudur:

```
"eğer a ile b birbirine eşit ise..."
```

Biraz daha açarak söylemek gerekirse:

```
"eğer a değişkeninin değeri b değişkeninin değeriyle aynı ise..."
```

Bu arada, kod içindeki **==** (*çift eşittir*) işaretini birbirine **bitişik** olarak yazmamız gerektiğine dikkat edin. Yani bu çift eşittir işaretini **ayrı yazmamaya özen göstermeliyiz**. Aksi hâlde yazdığımız program **hata** verecektir.

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

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

Yukarıda yazdığımız kod şu anlama geliyor:

```
>>> Eğer a değişkeninin değeri b değişkeninin değeriyle aynı ise:
...     ekrana 'a ile b birbirine eşittir,' 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 hâliyle çalıştırmaya kalkışırsanız Python size, **a** ve **b** değişkenlerinin tanımlanmadığını söyleyen bir **hata** mesajı gösterecektir.

Biraz sonra bu yarım yamalak kodu eksiksiz bir hâle nasıl getireceğimizi göreceğiz. Ama şimdi, burada bir **parantez açalım** ve Python'da girintileme işleminden ve **kodların içine nasıl açıklama ekleneceğinden** bahsedelim kısaca. Öncelikle **girintilemeden** bahsedelim. Çünkü bundan sonra girintilerle bol bol haşır neşir olacaksınız.

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

Ancak bu kuralı uygularken TAB veya SPACE tuşlarına basma seçeneklerinden **yalnızca birini uygulayın**. Yani bir yerde TAB tuşuna başka yerde SPACE tuşuna basıp Python'ın kafasının karışmasına yol açmayın. Python'daki girintileme sistemi konusuna biraz sonra tekrar döneceğiz. Ama şimdi, bunu bir kenara bırakıp Python'da kodlar içine nasıl **açıklama/yorum eklenir**, biraz da kısaca bundan bahsedelim.

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

Peki, programımıza bu açıklayıcı notları nasıl ekleyeceğiz?

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

Hemen bir **örnek** verelim:

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

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

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

```
a = 23
b = 23
```

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

Programımızı yazmaya devam edelim:

```
a = 23
b = 23

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

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

Ama bu program ahım şahım bir şey olmasa da en azından bize **if** deyiminin **nasıl kullanılacağı hakkında** önemli bir fikir verdi. Artık bilgilerimizi bu programın bize sağladığı temel üzerine inşa etmeye devam edebiliriz.

Her zamanki gibi **boş bir Kate** (*veya Gedit ya da IDLE*) belgesi açıyoruz ve içine şunları yazıyoruz:

```
parola = "python"

cevap = input("Lütfen parolanızı giriniz: ")

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

Gördüğünüz gibi burada öncelikle **parola** adlı bir değişken oluşturduk. Bu **parola** adlı değişkenin değeri, **kullanıcının girmesi gereken parolanın kendisi** oluyor. Ardından **cevap** adlı başka bir değişken daha tanımlayıp **input()** fonksiyonunu bu değişkene atadık. Daha sonra da **if** deyimi yardımıyla, "**Eğer cevap değişkeninin değeri parola değişkeninin değeriyle aynı ise ekrana 'Parola onaylandı! Programa hoş geldiniz!' yazdır**" dedik. Bu programı çalıştırdığımızda, eğer kullanıcının girdiği kelime "**python**" ise **parola onaylanacaktır**. Yok, eğer kullanıcı başka bir kelime yazarsa program derhâl **kapanacaktır**.

Aynı programı şu şekilde **kısaltarak** da yazabiliriz:

```
parola = input("Lütfen parolanızı giriniz: ")

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

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

ELSE

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

```
isim = input("Senin ismin ne?")

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

else:
    print(f"{isim} ismini pek sevmem!")
```

Burada yaptığımız şey şu: Öncelikle kullanıcıya, "**Senin ismin ne?**" diye soruyoruz. Bu soruyu, **isim** adı verdiğimiz bir değişkene atadık. **Daha sonra şu cümleyi Pythonca'ya çevirdik**:

"Eğer isim değişkeninin değeri Ferhat ise ekrana '**Ne güzel bir isim bu!**' cümlesini yazdır. Yok eğer **isim** değişkeninin değeri **Ferhat** değil de başka herhangi bir şeyse ekrana **isim** değişkeninin değerini ve '**adını pek sevmem!**' cümlesini yazdır."

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

```
parola = input("Lütfen parolanızı giriniz: ")

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

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

ELIF

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

```
s = input("Bir meyve adı yazınız: ")

if s == "elma":
    print("elma bir meyvedir")

elif s == "armut":
    print("armut bir meyvedir")

else:
    print(f"{s} bir meyve değildir!")
```

Burada, şu Türkçe ifadeyi **Pythonca'ya çevirdik:**

"Kullanıcıya, bir meyve ismi yazmasını söyle. Eğer kullanıcının yazdığı isim **`elma`** ise ekrana '**`elma bir meyvedir`**' çıktısı verilsin. Yok eğer kullanıcının yazdığı isim **`elma`** değil, ama **`armut`** ise ekrana '**`armut bir meyvedir`**' çıktısı verilsin. Eğer kullanıcının yazdığı isim bunlardan hiçbiri değilse ekrana **`meyve`** değişkeninin değeri ve '**`bir meyve değildir`**' çıktısı yazılsın."

Bu arada, yukarıdaki son satırı şu şekilde yazabileceğimizi de biliyorsunuz:

```
print(s, "bir meyve değildir!")
```

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

```
sayı = 100

if sayı == 100:
    print("sayı 100'dür")

if sayı <= 150:
    print("sayı 150'den küçüktür")

if sayı > 50:
    print("sayı 50'den büyüktür")

if sayı <= 100:
    print("sayı 100'den küçüktür veya 100'e eşittir")
```

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

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

Eğer bu programı **`elif`** deyimini kullanarak yazarsak **sonuç şu olacaktır:**

Öncelikle kodumuzu görelim:

```
sayı = 100

if sayı == 100:
    print("sayı 100'dür")

elif sayı <= 150:
    print("sayı 150'den küçüktür")

elif sayı > 50:
    print("sayı 50'den büyüktür")
```

```
elif sayı <= 100:
    print("sayı 100'den küçüktür veya 100'e eşittir")
```

Bu kodların **çıktısı** ise şöyle olacaktır:

```
sayi 100'dür
```

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

Buraya kadar Python'da pek çok şey öğrenmiş olduk. **if**, **elif** ve **else** deyimlerini de öğrendiğimize göre artık **çok basit bir hesap makinesi** yazabiliriz:

```
seçenek1 = "(1) toplama"
seçenek2 = "(2) çıkarma"
seçenek3 = "(3) çarpma"
seçenek4 = "(4) bölme"

print(seçenek1)
print(seçenek2)
print(seçenek3)
print(seçenek4)

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

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

if soru == "2":
    sayı3 = int(input("Çıkarma için ilk sayıyı girin: "))
    print(sayı3)
    sayı4 = int(input("Çıkarma için ikinci sayıyı girin: "))
    print(f"{sayı3} - {sayı4} = {sayı3 - sayı4}")
```

```
if soru == "3":
    sayı5 = int(input("Çarpma için ilk sayıyı girin: "))
    print(sayı5)
    sayı6 = int(input("Çarpma için ikinci sayıyı girin: "))
    print(f"{sayı5} x {sayı6} = {sayı5 * sayı6}")

if soru == "4":
    sayı7 = int(input("Bölme için ilk sayıyı girin: "))
    print(sayı7)
    sayı8 = int(input("Bölme için ikinci sayıyı girin: "))
    print(f"{sayı7} / {sayı8} = {sayı7 / sayı8}")
```

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

Dilerseniz bir sonraki bölüme geçmeden önce muhtemelen pek çok kişinin merak ettiği şu sorunun cevabını verelim: *Acaba Python programlarını nasıl .exe hâline getirebiliriz?*

Windows işletim sisteminde Python programlarımızı bir .exe dosyası hâline getirebilmek için **cx_Freeze** adlı hâricî bir programa ihtiyacımız var. Şimdi, bu programı nasıl kuracağımızı ve temel olarak nasıl kullanacağımızı inceleyeceğiz.

cx_Freeze'i kullanabilmek için ilk önce bunu **bilgisayarımıza kurmamız lazım**. Bu programı kurabilmek için öncelikle klavyedeki **Windows+R** tuşlarına birlikte basıp açılan pencerede '**cmd**' yazdıktan sonra **ENTER** düğmesine basarak komut satırını başlatıyoruz. (*Ama Python'ın etkileşimli kabuğunu başlatmıyoruz.*)

Orada şu komutu veriyoruz:

```
pip install cx_Freeze
```

Bu komut, **cx_Freeze** adlı programı bilgisayarımıza kuracak. Kurulum tamamlandıktan sonra **cx_Freeze**'i başarılı bir şekilde kurduğumuzdan emin olmak için yine sistem komut satırında şu komutu çalıştıralım:

```
cxfreeze
```

Bu komut, şuna benzer bir **çıktı** vermeli:

```
Usage: cxfreeze [options] [SCRIPT]

Freeze a Python script and all of its referenced modules to a base
executable which can then be distributed without requiring a Python
installation.

cxfreeze: error: script or a list of modules must be specified
```

Şimdi, bu programı kullanarak kendi yazdığımız bir programı **.exe hâline** getirelim. Mesela, yukarıda yazdığımız hesap makinesi programını '**hesap.py**' adlı bir dosyaya kaydedelim.

Daha sonra, '**hesap.py**' dosyasının bulunduğu klasör içinde '**setup.py**' adlı **başka bir dosya daha oluşturalım** ve **bu dosyayı açıp içine şunları yazalım**:

```
import cx_Freeze

program_dosyası = cx_Freeze.Executable("hesap.py")

cx_Freeze.setup(executables=[program_dosyası])
```

cx_Freeze aslında tıpkı **platform**, **webbrowser** ve **random** gibi bir modüldür. Ancak **cx_Freeze** modülünün öğrendiğimiz öteki modüllerden farkı; bir standart kütüphane modülü değil, bir **hâricî modül** olmasıdır. Hâricî modüller, standart kütüphane modüllerinin aksine Python programlama dilinin bir parçası değildir. O yüzden hâricî modüllerin kullanılabilmesi için bunların

öncelikle **sistemimize kurulması** gerekir. Ama hâricî modüller de kurulduktan sonra aynı standart kütüphane modülleri gibi **`import`** komutuyla içe aktarılır:

```
import cx_Freeze
```

Modülü uygun şekilde içe aktardıktan sonra, modülün içindeki **`Executable`** adlı sınıfı çalıştırarak **programımızın kodlarını barındıran dosyanın adını belirtiyoruz**:

```
program_dosyası = cx_Freeze.Executable("hesap.py")
```

Son olarak **`program_dosyası`** adlı bu değişkeni **`cx_Freeze`** modülünün **`setup()`** adlı fonksiyonunun **`executables`** isimli argümanına **değer** olarak atıyoruz:

```
cx_Freeze.setup(executables=[program_dosyası])
```

Bu dosyada yazan her şeyi şu an tam olarak anlamamış olabilirsiniz. Ama bunun için endişe etmenize gerek yok. İlerleyen derslerde Python hakkında daha çok bilgi edindikten sonra burada gördükleriniz gözünüze çok daha anlaşılır görünecek.

Yukarıda gösterdiğimiz şekilde '**`setup.py`**' dosyamızı hazırladıktan sonra şimdi, **tekrar komut satırını açıp** şu komutu verelim:

```
py setup.py build
```

veya:

```
python setup.py build
```

Bu komut, çalıştırıldığı klasör içinde "**`build`**" adlı bir dizin, onun içinde de "**`exe.win-amd64-3.8`**" adlı bir başka dizin oluşturacak. Bu dizinin içinde '**`hesap.exe`**' adlı bir dosya göreceksiniz. Bu dosyaya **çift tıklayarak** programınızı çalıştırabilirsiniz.

Programınızı çalıştırıp normal bir şekilde kullanırken şunu farkedeceksiniz. Hesap makineniz, hesap işlemlerini yaptıktan sonra işlemin sonucunu göstermeden kapanıyor. Aslında programınız normal çalışıyor. Fakat sonucu gösterir göstermez *çok hızlı bir şekilde kapandığı için bizim işlem sonucunu görmeye fırsatımız olmuyor*.

Bu durumu önlemek için yazdığımız hesap makinesi programının **en son satırına** bir **`input()`** satırı ekleyebiliriz. Böylece programımız tüm işlemleri bitirdikten sonra, kapanmadan önce **ENTER** düğmesine basılmasını bekleyecektir. **Yani programımızın son hâli şöyle görünecek:**

```
seçenek1 = "(1) toplama"
seçenek2 = "(2) çıkarma"
seçenek3 = "(3) çarpma"
seçenek4 = "(4) bölme"

print(seçenek1)
print(seçenek2)
print(seçenek3)
print(seçenek4)

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

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

if soru == "2":
    sayı3 = int(input("Çıkarma için ilk sayıyı girin: "))
    print(sayı3)
    sayı4 = int(input("Çıkarma için ikinci sayıyı girin: "))
    print(f"{sayı3} - {sayı4} = {sayı3 - sayı4}")

if soru == "3":
    sayı5 = int(input("Çarpma için ilk sayıyı girin: "))
    print(sayı5)
    sayı6 = int(input("Çarpma için ikinci sayıyı girin: "))
```

```
    print(f"{sayı5} x {sayı6} = {sayı5 * sayı6}")

if soru == "4":
    sayı7 = int(input("Bölme için ilk sayıyı girin: "))
    print(sayı7)
    sayı8 = int(input("Bölme için ikinci sayıyı girin: "))
    print(f"{sayı7} / {sayı8} = {sayı7 / sayı8}")

input("programdan çıkmak için ENTER'a basın!")
```

'**hesap.py**' dosyasını bu şekilde değiştirdikten sonra **py setup.py build** veya python **setup.py build** komutunu tekrar çalıştırarak **.exe** dosyasını yeniden oluşturabilirsiniz.

13

PYTHON'DA GİRİNTİLEME SİSTEMİ

Geçen bölümde koşullu durumlardan söz ederken Python'daki **girintileme** konusuna da kısaca değinmiştik. Bu bölümde, bu konuyu biraz daha ayrıntılı bir şekilde ele alacağız.

O hâlde, lafı hiç uzatmadan girintileme konusuna hızlı bir giriş yapalım...

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

```
#include <stdio.h>

int main() {
    int a = 1;

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

Eğer istersek yukarıdaki kodları şöyle de yazabiliriz:

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

Bu kodları daha sonra okuyacak kişilerin nefretini kazanmak dışında, kodları bu şekilde yazmamız **herhangi bir soruna neden olmaz**. Yani yukarıda gösterilen her iki kod da derleyiciler ('**compiler**') tarafından aynı şekilde okunup **başarıyla derlenecektir**.

Eğer yazdığınız kodların **okunaklı** olmasını istiyorsanız, yukarıda gösterilen **ilk kodlama biçimini tercih etmeniz gerekir**. Ama dediğim gibi **ikinci kodlama biçimini kullanmanız da programınızın çalışmasını etkilemez**.

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

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

```
a = 1

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

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

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

Bu kodlar çalışma sırasında **hata** verecektir.

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

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

Gedit'i ilk açtığınızda, **Düzen**/**Yeğlenenler**/**Düzenleyici** yolunu takip ederek sekmeler başlığı altındaki ayarları kontrol etmelisiniz. Python programlarımızın girinti yapısının düzgün olabilmesi için orada sekme genişliğini **4** olarak ayarlamanız, '**Sekme yerine boşluk ekle**' seçeneğinin yanındaki kutucuğu da işaretli hâle getirmeniz gerekir. Buna benzer ayarlar bütün iyi metin düzenleyicilerde bulunur. Örneğin **Geany** adlı metin düzenleyiciyi kullanıyorsanız **Düzenle**/**Tercihler**/**Düzenleyici**/**Girinti** yolunu takip ederek **şu ayarlamaları yapabilirsiniz**:

```
genişlik => 4
Tür => Boşluklar
sekme genişliği => 4
```

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

14 DÖNGÜLER

Hatırlarsanız koşullu durumları anlattığımız bölümde, basit bir *hesap makinesi* yapmıştık. Ancak dikkat ettiyseniz o hesap makinesi programında **toplama**, **çıkarma**, **çarpma** veya **bölme** işlemlerinden birini seçip daha sonra o seçtiğimiz işlemi bitirdiğimizde program kapanıyor, başka bir işlem yapmak istediğimizde ise programı yeniden başlatmamız gerekiyordu.

Aynı şekilde kullanıcı adı ve parola soran bir program yazsak şu anki bilgilerimizle her defasında **programı yeniden başlatmak zorunda kalırız**. Yani kullanıcı adı ve parola **yanlış girildiğinde** bu kullanıcı adı ve parolayı tekrar tekrar soramayız; **programı yeniden başlatmamız gerekir**. İşte bu bölümde Python'da yazdığımız kodları **sürekli hâle** getirmeyi, tekrar tekrar döndürmeyi öğreneceğiz.

Kodlarımızı sürekli döndürmemizi için bize yardımcı olacak ögelere Python'da **döngü** ('`loop`') adı veriliyor.

Bu bölümde iki tane döngüden bahsedeceğiz: `while` ve `for` döngüleri. Ayrıca bu bölümde döngüler dışında `break` ve `continue` deyimleri ile `range()` ve `len()` fonksiyonlarına da değineceğiz. Böylece ilerleyen bölümlerde işleyeceğimiz **daha karmaşık konuları biraz daha kolay anlamamızı sağlayacak** temeli edineceğiz. İsterseniz lafı daha fazla uzatmadan yola koyulalım ve ilk olarak `while` döngüsüyle işe başlayalım.

WHILE DÖNGÜSÜ

Yazdığımız kodları tekrar tekrar döndürmemizi sağlayan, programımıza bir süreklilik katan ögelere **döngü** adı verilir demiştik. İşte, `while` döngüsü de bu tanıma tam olarak uyar; yani yazdığımız bir programdaki kodların tamamı işletilince **programın kapanmasına engel olur** ve **kod dizisinin en başa dönmesini sağlar**. Ne demek istediğimizi anlatmanın en iyi yolu bununla ilgili bir **örnek** vermek olacaktır.

O hâlde şu küçücük örneği bir inceleyelim bakalım:

```
a = 0
a = a + 1
print(a)
```

Bu minicik kodun yaptığı iş, birinci satırda **a** adlı bir değişkene **0** değerini verip ikinci satırda bu değere **1** eklemek, üçüncü satırda da bu yeni değeri ekrana yazdırmaktır. Dolayısıyla bu kod parçasının vereceği **çıktı** da **1** olacaktır. Bu çıktıyı verdikten sonra ise **program sona erecektir**.

Peki, ya biz **a** değişkeninin değerini mesela **100**'e kadar teker teker artırmak istersek ne yapacağız? **Acaba şöyle bir şey yazabilir miyiz?**

```
a = 0
a = a + 1
print(a)

a = a + 1
print(a)

a = a + 1
print(a)
```

Böyle böyle 100'e kadar gidilebilir...

Elbette bu tip bir kod yazmaya çalışmak akıl kârı değil. Biz bu tür işlerimiz için **döngülerden** yararlanacağız. Mesela bu iş için `while` döngüsünü kullanabiliriz.

Dikkatlice bakın:

```
a = 0
while a < 100:
    a = a + 1
    print(a)
```

Bu kodu çalıştırdığımızda, 1'den 100'e kadar olan sayıların ekrana yazdırıldığını görürüz.

NOT Bu arada eğer arzu ederseniz yukarıdaki `a = a + 1` satırını `a += 1` şeklinde de yazabilirsiniz.

Dilerseniz yukarıdaki kodları daha iyi anlayabilmek için satırları teker teker inceleyelim:

İlk satırda, `0` değerine sahip bir `a` değişkeni tanımladık:

```
a = 0
```

İkinci ve **üçüncü** satırlarda, "`a` değişkeninin değeri 100 sayısından küçük olduğu müddetçe `a` değişkeninin değerine `1` ekle," cümlesini **Python'caya çevirdik:**

```
while a < 100:
    a += 1
```

Son satırda ise bu yeni `a` değerini ekrana yazdırdık:

```
print(a)
```

Gördüğünüz gibi `while` döngüsü 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 anlıyor,
- `a` değerine `1` ekliyor (*0 + 1 = 1*),

- Bu değeri ekrana yazdırıyor (*1*),
- Başa dönüp tekrar **a** değişkeninin değerini kontrol ediyor,
- **a**'nın şu anda **1** olduğunu görüyor,
- **a** değerinin hâlâ **100**'den küçük olduğunu anlıyor,
- **a** değerine **1** ekliyor (*1 + 1 = 2*),
- Bu değeri ekrana yazdırıyor (*2*),
- Bu işlemi **99** sayısına ulaşana dek tekrarlıyor ve en sonunda bu sayıya da **1** ekleyerek döngüyü sonlandırıyor.

while döngülerinin çalışma mantığını anlamak için bir **örnek** daha verelim:

```
durum = 'devam'

while durum == 'devam':
    soru = input("bir veri girin: ")
    print(soru)
    if soru == 'q':
        durum = 'yeter'
```

Burada da öncelikle değeri '**devam**' olan **durum** adlı bir değişken tanımladık. Bu değişkenin değeri '**devam**' olduğu müddetçe çalışacak bir program yazacağız.

Ardından programımızın **sürekli olarak çalışmasını sağlayacak** olan **while** döngümüzü yazıyoruz:

```
while durum == 'devam':
    soru = input("bir veri girin: ")
    print(soru)
```

Buna göre **durum** değişkeninin değeri '**devam**' olduğu müddetçe **input()** fonksiyonu aracılığıyla kullanıcıdan bir **veri girmesini istiyoruz** ve **her defasında girilen bu değeri ekrana yazdırıyoruz**.

Sıra geldi, **döngüyü durdurup programı sonlandırmamızı sağlayacak koşulu tanımlamaya:**

```
if soru == 'q':
    durum = 'yeter'
```

Buna göre eğer kullanıcı '**q**' değerini girerse **durum** değişkeninin değerini '**yeter**' olarak değiştiriyoruz. **while** döngüsünün devam etmesi, **durum** değişkeninin değerinin '**devam**' olmasına bağlı olduğu için bu değişkenin değeri değişince döngümüz sona eriyor. Bu arada **döngüyü sona erdirmek için** **durum** değişkeninin değerini mutlaka '**yeter**' yapmak **zorunda değilsiniz**. Döngünün sona ermesi için bu değişkenin değerinin '**devam**' dışında herhangi bir şey olması yeterlidir.

Böylece **while** döngüsünün çalışma mantığı hakkında hiç değilse bir fikir sahibi olmuş olduk. Dilerseniz tam bu noktada ilerleyebilmek için ihtiyacımız olacak bazı **yeni işleçlere** ('**operators**') değinelim:

Şimdiye kadar aslında bu işleçlerden birkaç tanesini gördük. Mesela:

- **+ işleci:** Toplama işlemi yapmamızı sağlıyor.
- **- işleci:** Çıkarma işlemi yapmamızı sağlıyor.
- **/ işleci:** Bölme işlemi yapmamızı sağlıyor.
- *** işleci:** Çarpma işlemi yapmamızı sağlıyor.
- **> işleci:** "büyüktür" anlamına geliyor.
- **< işleci:** "küçüktür" anlamına geliyor.
- **== işleci:** "eşittir" anlamına geliyor

Bir de henüz görmediklerimiz, ama bilmemiz gerekenler var:

- **>= işleci:** "büyük eşittir"anlamına geliyor.
- **<= işleci:** "küçük eşittir" anlamına geliyor.
- **!= işleci:** "eşit değildir" anlamına geliyor. (*örn. 2 * 2 != 5*)
- **and işleci:** "ve" anlamına geliyor.
- **or işleci:** "veya" anlamına geliyor.
- **not işleci:** "değil" anlamına geliyor.

Bu işleçlerle **anlam yönünden bağlantılı** olan şu iki değeri de belirtmeden geçmeyelim:

- `True` **değeri:** "doğru" anlamına geliyor.
- `False` **değeri:** "yanlış" anlamına geliyor.

Bu işleçleri ve değerleri 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, daha önce yazdığımız **hesap makinesi programına** değinmiştik. Şimdi, bu programı tekrar görelim:

```
seçenek1 = "(1) toplama"
seçenek2 = "(2) çıkarma"
seçenek3 = "(3) çarpma"
seçenek4 = "(4) bölme"

print(seçenek1)
print(seçenek2)
print(seçenek3)
print(seçenek4)

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

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

if soru == "2":
    sayı3 = int(input("Çıkarma için ilk sayıyı girin: "))
    print(sayı3)
    sayı4 = int(input("Çıkarma için ikinci sayıyı girin: "))
    print(f"{sayı3} - {sayı4} = {sayı3 - sayı4}")

if soru == "3":
```

```
    sayı5 = int(input("Çarpma için ilk sayıyı girin: "))
    print(sayı5)
    sayı6 = int(input("Çarpma için ikinci sayıyı girin: "))
    print(f"{sayı5} x {sayı6} = {sayı5 * sayı6}")

if soru == "4":
    sayı7 = int(input("Bölme için ilk sayıyı girin: "))
    print(sayı7)
    sayı8 = int(input("Bölme için ikinci sayıyı girin: "))
    print(f"{sayı7} / {sayı8} = {sayı7 / sayı8}")
```

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

```
while True:
    seçenek1 = "(1) toplama"
    seçenek2 = "(2) çıkarma"
    seçenek3 = "(3) çarpma"
    seçenek4 = "(4) bölme"

    print(seçenek1)
    print(seçenek2)
    print(seçenek3)
    print(seçenek4)

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

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

    if soru == "2":
        sayı3 = int(input("Çıkarma için ilk sayıyı girin: "))
        print(sayı3)
        sayı4 = int(input("Çıkarma için ikinci sayıyı girin: "))
        print(f"{sayı3} - {sayı4} = {sayı3 - sayı4}")
```

```
if soru == "3":
    sayı5 = int(input("Çarpma için ilk sayıyı girin: "))
    print(sayı5)
    sayı6 = int(input("Çarpma için ikinci sayıyı girin: "))
    print(f"{sayı5} x {sayı6} = {sayı5 * sayı6}")

if soru == "4":
    sayı7 = int(input("Bölme için ilk sayıyı girin: "))
    print(sayı7)
    sayı8 = int(input("Bölme için ikinci sayıyı girin: "))
    print(f"{sayı7} / {sayı8} = {sayı7 / sayı8}")
```

Burada şu değişiklikleri yaptık:

İlk önce programın en başına `while True:` ifadesini ekledik. Bu sayede programımıza şu komutu vermiş olduk: "**Doğru olduğu müddetçe aşağıdaki komutları çalıştırmaya devam et!**"

Zira yukarıda verdiğimiz **işleç** ve **değer** tablosundan da hatırlayacağınız gibi `True`, "**doğru**" anlamına geliyor.

Peki, ne doğru olduğu müddetçe? Neyin doğru olduğunu açıkça belirtmediğimiz için Python burada **her şeyi doğru kabul ediyor**. Yani bir nevi, "**aksi belirtilmediği sürece aşağıdaki komutları çalıştırmaya devam et!**" emrini yerine getiriyor. Dolayısıyla herhangi bir kısıtlama olmaksızın sonsuza kadar devam etmesini istediğiniz döngülerde `while True:` kodunu kullanabilirsiniz.

İkinci değişiklik ise `while True:` ifadesinin altında kalan bütün satırları bir seviye **sağa kaydırmak** oldu.

Bir de şu **örneğe** bakalım:

```
soru = input("Python mı Ruby mi?")
while soru != "Python":
    print("Yanlış cevap!")
```

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

Bu programı çalıştırdığımızda sorulan soruya "**Python**" cevabı vermezsek program, biz müdahale edene kadar ekrana "**Yanlış cevap!**" **çıktısını** vermeye devam edecektir. Çünkü biz bu kodla, **Python'a şu komutu vermiş olduk:** "Soru değişkeninin cevabı Python olmadığı müddetçe ekrana '**Yanlış cevap!**' çıktısını vermeye devam et!"

Eğer bu programı durdurmak istiyorsak **CTRL+C** tuşlarına basmamız gerekir...

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

```
soru = input("Python mı Ruby mi?")

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

Yeni öğrendiğimiz bu **!=** işlecini kullanarak daha önce verdiğimiz şu örneği biraz **sadeleştirebiliriz**:

```
durum = 'devam'

while durum == 'devam':
    soru = input("bir veri girin: ")
    print(soru)
    if soru == 'q':
        durum = 'yeter'
```

Dikkatlice bakın:

```
soru = ''

while soru != 'q':
    soru = input("bir veri: ")
    print(soru)
```

Burada öncelikle **soru** adlı, **içi boş bir değişken** tanımladık. Bu değişkenin değeri '**q**' **olmadığı müddetçe** kullanıcıdan bir veri girmesini istiyoruz. Dolayısıyla eğer kullanıcı **input()** fonksiyonuna '**q**' cevabını verirse **döngümüz sonlanıyor** ve **programımız kapanıyor**. Bu arada **soru** değişkeninin değeri illa boş bir karakter dizisi olmak zorunda değil. Bu değişkeni herhangi bir şekilde tanımlayabiliriz:

```
soru = 'falanfilan'

while soru != 'q':
    soru = input("bir veri: ")
    print(soru)
```

Ama mutlaka soru adlı bir değişken tanımlamamız gerekir. Eğer tanımlamazsak programımız **while soru!= 'q'** satırında **hata** verecektir. Çünkü bu satır, kendisinden önce **soru** adlı bir değişkenin tanımlanmış olmasını bekliyor.

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

for Döngüsü

Bir önceki bölümde verdiğimiz örneklerde **while** döngüsünün belirli bir ölçütün geçerlilik durumuna göre işlem yaptığını görmüştük. Örneğin **while a < 5** dediğimizde, **a** değişkeninin değeri **5**'ten küçük olduğu müddetçe bir işlem yapılacağını anlıyoruz. Aynı şekilde **while True** dediğimizde, aksi belirtilene kadar bir işlem yapılacağını ifade ediyoruz.

Buna göre **while** döngüsü, belirtilen bir ölçüt veya koşul geçerli olmaya devam ettiği müddetçe çalışmaya devam edecek, bu ölçüt veya koşul geçerliliğini yitirdiğinde çalışmayı kesecektir.

for döngüsünün ise **daha farklı bir çalışma mantığı** vardır.

Diyelim ki elimizde şöyle bir değişken var:

```
>>> sayılar = 123456789
```

Biz bu değişken içindeki her bir rakamı **alt alta** yazdırmak istiyoruz. Yani şöyle bir **çıktı** elde etmek istiyoruz:

```
1
2
3
4
5
6
7
8
9
```

Bu işlemi daha önce öğrendiğimiz **yıldızlı ifadeler yardımıyla** yapabileceğimizi biliyorsunuz:

```
>>> print(*str(sayılar), sep="\n")
```

Bildiğiniz gibi yıldızlı ifadeleri, **doğrudan sayılarla birlikte kullanamıyoruz**. O yüzden önce bu sayıları **str()** fonksiyonu yardımıyla **karakter dizisine** dönüştürdük, sonra da yıldız işlecini **sayılar** değişkeni üzerine uyguladık. **sep** argümanına verdiğimiz **\n** değeri sayesinde de her bir sayı **alt alta sıralandı**.

İşte bu işlemi **for** döngüsü aracılığıyla da yapabiliriz. **Dikkatlice bakın:**

```
>>> for sayı in str(sayılar):
...     print(sayı)
```

Bu komut **sayılar** değişkeni içindeki tüm rakamları **tek tek** ve **alt alta** sıralayacaktır. Bunu sağlayan şey ise **for** döngüsünün **sayılar** değişkeni üzerinde bir **döngü** oluşturarak bu değişkenin **her bir ögesi üzerinde tek tek işlem yapabilme kabiliyetine sahip olmasıdır**. Bu kodlarda gördüğünüz **in** işleci "**içinde**" anlamına geliyor. Birazdan bu işleci ayrıntılı bir şekilde inceleyeceğiz.

Bu arada, sayılar üzerinde **doğrudan döngü kurulamadığına** da dikkatinizi çekmek istiyorum. O yüzden öncelikle **sayılar** adlı değişkeni **str()** fonksiyonu yardımıyla karakter dizisine çevirdik.

Biz esasında burada Python'a şu emri vermiş oluyoruz:

```
>>> sayılar değişkeni içindeki her bir sayıya 'sayı' adını ver:
...     sayıyı ekrana bas
```

Bir **örnek** daha verelim:

```
>>> for harf in "istihza":
...     print(harf)
```

Bu kodlar da "**istihza**" karakter dizisi içindeki her bir harfi tek tek ve alt alta ekrana basacaktır. Burada da **in** işlecinin cümleye "**içinde**" anlamı verdiğini gözden kaçırmıyoruz.

Tıpkı bir önceki örnekte olduğu gibi burada da **Python'a şu emri vermiş oluyoruz**:

```
>>> "istihza" karakter dizisi içindeki her bir harfe 'harf' adını
ver:
...     harfi ekrana bas
```

Gelin isterseniz, bu noktada ufacık bir uygulama yapalım. Mesela yukarıdaki **`sayılar`** değişkeni içindeki her bir rakamın **karesini** ve **küpünü** yan yana hesaplayalım. Yani şöyle bir **çıktı** elde edelim:

```
sayı    karesi  küpü
----    ------  ----
1       1       1
2       4       8
3       9       27
4       16      64
5       25      125
6       36      216
7       49      343
8       64      512
9       81      729
```

Önce kodlarımızın **son hâlini** görelim:

```
sayılar = 123456789

print("sayı\tkaresi\tküpü")
print("----\t------\t----")

for sayı in str(sayılar):
    print(f"{sayı}\t{int(sayı)**2}\t{int(sayı)**3}")
```

Burada öncelikle **`sayılar`** adlı değişkenimizi tanımladık:

```
sayılar = 123456789
```

Ardından sayıyı, karesini ve küpünü göstereceğimiz tablonun **başlık bilgilerini oluşturduk**:

```
print("sayı\tkaresi\tküpü")
print("----\t------\t----")
```

Kodların buraya kadar olan kısmını çalıştırırsak şöyle bir **çıktı** alırız:

```
sayı    karesi  küpü
----    ------  ----
```

Şimdi de bu tabloya gerekli bilgileri yerleştirebilmek için **`for`** döngüsünden yararlanıp **`sayılar`** adlı değişkenin üzerinde **döngü** kurarak bu değişkenin her bir ögesi üzerinde **kare** ve **küp hesaplama işlemi yapıyoruz:**

```
for sayı in str(sayılar):
    print(f"{sayı}\t{int(sayı)**2}\t{int(sayı)**3}")
```

Burada **`sayılar`** değişkeni üzerinde döngü kurabilmek için öncelikle bunu **`str()`** fonksiyonuyla bir **karakter dizisine çevirdik**. Ardından bu değişken içindeki her bir ögeye **`sayı`** adını verdik.

İkinci satırda ise **`sayı`** değişkenini, bu değişkenin 2. kuvvetini (*yani karesini*) ve 3. kuvvetini (*yani küpünü*) **`print()`** fonksiyonunu ve **`f`**'li karakter dizilerini kullanarak ekrana **çıktı** olarak verdik.

Bu ikinci satır gözünüze biraz *karmaşık* görünmüş olabilir, ama aslında yaptığımız işlem özünde şudur:

```
for sayı in str(sayılar):
    print(sayı, int(sayı)**2, int(sayı)**3)
```

Biz sadece **`f`**'li karakter dizilerini ve **`\t`** **kaçış dizisini kullanarak çıktıyı** daha **anlaşılır** ve **cazip** hâle getirdik.

Ayrıca burada **`sayı`** adını verdiğimiz her bir değişken üzerinde **aritmetik işlem yapabilmek için** öncelikle bu değişkeni **`int()`** fonksiyonuyla **sayıya dönüştürdüğümüze** de dikkat edin.

Gelin isterseniz önceki bölümde **`while`** döngüsünü anlatırken yazdığımız şu kodu tekrar hatırlayalım:

```
a = 0
while a < 100:
    a += 1
    print(a)
```

Bu kod yardımıyla ekrana 1'den 100'e kadar olan sayıları yazdırabiliyorduk. Aynı işlemi daha basit bir şekilde **`for`** döngüsü ve **`range()`** adlı bir fonksiyon yardımıyla da yapabiliriz (*bu fonksiyonu birazdan ayrıntılı olarak inceleyeceğiz*):

```
for i in range(1, 100):
    print(i)
```

Yukarıdaki Python'ca kod, **Türkçede aşağı yukarı şu anlama gelir:**

```
1- 100 aralığındaki sayıların her birine i adını ver:
    i'nin değerini yaz!
```

Gördüğünüz gibi **`for`** döngüsünü de kullanmak hiç zor değil. Sadece biraz **pratik** gerekiyor.

RANGE() FONKSİYONU

Yukarıda **for** döngülerini anlatırken **range()** adlı bir fonksiyondan söz etmiştik. İşte bu bölümde, **range()** fonksiyonunu inceleyeceğiz.

range() adlı fonksiyon Python'da **sayı aralıklarını belirtmemizi sağlar**. Zaten İngilizcede de bu kelime "**aralık**" anlamına gelir. Mesela:

```
>>> print(range(100))
range(0, 100)
```

Bu **range()** fonksiyonu, yukarıdaki şekilde **0** ila **100** arasındaki sayıları üretir, ancak bunları **hemen listelemez**. Bu fonksiyonun, ürettiği sayı aralığını listelemesini sağlamak için mesela **for** döngüsünden yararlanabiliriz:

```
>>> for i in range(100):
...     print(i)
```

Bu kodlar **0** ile **100** arasındaki sayıları yazdırmamızı sağlar.

Dilerseniz **range()** fonksiyonunun ürettiği sayı aralığını görüntülemek için **print()** fonksiyonundan ve **yıldızlı ifadelerden** de yararlanabilirsiniz:

```
>>> print(*range(100))
```

Gördüğünüz gibi **yıldız işareti**, **range()** fonksiyonun ürettiği sayı aralığını dağıtarak bu aralık içinde yer alan sayıların **ortaya dökülmesini** sağlıyor.

Bu şekilde sayı aralığını dağıttıktan sonra her bir ögenin arasına mesela **birer virgül** yerleştirebileceğinizi de biliyorsunuz:

```
>>> print(*range(100), sep=", ")
```

Başka bir **örnek** daha verelim:

```
>>> for i in range(100, 200):
...     print(i)
```

veya:

```
>>> print(*range(100, 200))
```

Bu komutlar da **100** ile **200** arasındaki sayıları döker.

Bir **örnek** daha:

```
>>> for i in range(1, 100, 2):
...     print(i)
```

veya:

```
>>> print(*range(1, 100, 2))
```

Bu komutlar ise **1** ile **100** arasındaki sayıları **ikişer ikişer** atlayarak yazdırmamızı sağlar.

Gördüğünüz gibi **range()** fonksiyonu da tıpkı daha önce öğrendiğimiz **print()**, **input()** ve **dir()** fonksiyonları gibi Python'daki **hazır fonksiyonlardan** biridir. Tıpkı öteki hazır fonksiyonlar gibi **range()** fonksiyonunu da herhangi bir ek işlem yapmadan istediğimiz her en kodlarımız içinde kullanabiliriz.

Gelin isterseniz bu fonksiyonu kullanarak basit bir program yazalım. Amacımız, öğrenciye sınavdan aldığı notu sorup bu notun **kaç puana denk geldiğini** göstermek. Örneğin eğer öğrenci sınavdan 90 almışsa, bu not '**AA**' puanına karşılık gelecektir.

Bu programda kullanacağımız **not ve puan tablosu** şöyle:

Not	Puan
90-100	AA
85-89	BA
80-84	BB
75-79	CB
70-74	CC
60-69	DC
50-59	DD
49-0	FF

Artık bu bilgiye göre kodlarımızı yazabiliriz...

Öncelikle kodlarımızı topluca görelim:

```
notunuz = input("notunuz: ")

if not notunuz:
    print("Lütfen notunuzu giriniz!")
    quit()

else:
    notunuz = int(notunuz)

if notunuz not in range(101):
    print("Notunuz 0 ile 100 arasında olmalı")
    quit()

elif notunuz in range(90, 101):
    puan = 'AA'

elif notunuz in range(85, 90):
    puan = 'BA'

elif notunuz in range(80, 85):
    puan = 'BB'

elif notunuz in range(75, 80):
    puan = "CB"
```

```
elif notunuz in range(70, 75):
    puan = 'CC'

elif notunuz in range(60, 70):
    puan = 'DC'

elif notunuz in range(50, 60):
    puan = 'DD'

else:
    puan = 'FF'

print('puanınız: ', puan)
```

Burada önce **öğrenciye notunu soruyoruz:**

```
notunuz = input("notunuz: ")
```

Burada basitçe **`input()`** fonksiyonunu kullanarak öğrenciden not bilgisini alıp bunu **`notunuz`** adlı değişkene atadık. Yalnız burada **değişken adını not olarak belirleme hatasına düşmeyin**. Bildiğiniz gibi '**`not`**' kelimesi, Python'da '**değil**' anlamında **işleç** olarak kullanılan bir ögedir. Eğer biz değişkenimize **`not`** adını verirsek işleç olan **`not`** ögesini **gölgelemiş oluruz**.

`notunuz` adlı değişkeni tanımladıktan sonra şöyle bir kod yazdık:

```
if not notunuz:
    print("Lütfen notunuzu giriniz!")
    quit()
```

Bu kodlara göre eğer **`notunuz`** değişkeninin doğruluk değeri **`False`** ise yani öğrenci herhangi bir değer girmeden **ENTER** düğmesine basmışsa dolayısıyla bu değişkenin **içi boşsa** ona bir **uyarı mesajı** gösteriyoruz ve **`quit()`** adlı bir fonksiyonu kullanarak **programdan çıkıyoruz**.

Ama eğer öğrenci bir değer girdikten sonra **ENTER** düğmesine basmışşa **`input()`** fonksiyonundan gelen bu karakter dizisini **`sayı`** değerli bir veriye dönüştürüyoruz:

```
else:
    notunuz = int(notunuz)
```

Öğrencinin boş bir değer göndermesi ihtimaline karşı **önlememizi böylece almış olduk**. Ancak kullanıcı **`0`** ile **`100`** aralığının dışında bir değer de girebilir. Not sistemimize göre bir öğrencinin alabileceği **en düşük not** 0, **en yüksek not** ise **`100`** olacağı için bizim bu aralığın dışındaki bir değerin girilmesi ihtimaline karşı da bir **önlem almamız gerekiyor**.

```
Dikkatlice bakın:
if notunuz not in range(101):
    print("Notunuz 0 ile 100 arasında olmalı")
    quit()
```

Eğer notunuz değeri **`0`** ile **`100`** aralığı içinde değilse yine kullanıcıya bir **uyarı mesajı** gösterip **`quit()`** fonksiyonu yardımıyla **programı durduruyoruz**.

Öğrencinin amacımıza uymayan veriler göndermesi ihtimaline karşı önlemlerimizi aldığımıza göre artık programımızın **asıl kısmını** tasarlamaya geçebiliriz:

```
elif notunuz in range(90, 101):
    puan = 'AA'
```

Eğer öğrencinin girdiği değer **`90`** ile **`101`** arasındaysa **`puan`** değişkeninin değerini '**`AA`**' olarak belirliyoruz.

Daha sonra da **geri kalan puan aralıklarını** uygun şekilde yazıyoruz:

```
elif notunuz in range(85, 90):
    puan = 'BA'

elif notunuz in range(80, 85):
    puan = 'BB'
```

```
elif notunuz in range(75, 80):
    puan = "CB"

elif notunuz in range(70, 75):
    puan = 'CC'

elif notunuz in range(60, 70):
    puan = 'DC'

elif notunuz in range(50, 60):
    puan = 'DD'

else:
    puan = 'FF'
```

Girilen notlara göre **puan** değerlerini belirledikten sonra artık **puan değişkeninin değerini ekrana yazdırabiliriz**:

```
print('puanınız: ', puan)
Gelin şimdi bu kodları bir de topluca görelim:
notunuz = input("notunuz: ")

if not notunuz:
    print("Lütfen notunuzu giriniz!")
    quit()

else:
    notunuz = int(notunuz)

if notunuz not in range(101):
    print("Notunuz 0 ile 100 arasında olmalı")
    quit()

elif notunuz in range(90, 101):
    puan = 'AA'

elif notunuz in range(85, 90):
    puan = 'BA'
```

```
elif notunuz in range(80, 85):
    puan = 'BB'

elif notunuz in range(75, 80):
    puan = "CB"

elif notunuz in range(70, 75):
    puan = 'CC'

elif notunuz in range(60, 70):
    puan = 'DC'

elif notunuz in range(50, 60):
    puan = 'DD'

else:
    puan = 'FF'

print('puanınız: ', puan)
```

Bu noktada size şöyle bir soru sorayım: Acaba **`range()`** fonksiyonu olmasaydı yukarıdaki kodları nasıl yazardık? Siz bu programı, **`range()`** fonksiyonu olmadan nasıl yazacağınızı düşünürken biz de yine Python'ın önemli **hazır fonksiyonlarından** birini daha incemeye geçelim.

LEN() FONKSİYONU

Python'ın son derece kullanışlı hazır fonksiyonlarından birisi de **`len()`** fonksiyonudur. Bu fonksiyon, **karakter dizilerinin uzunluğunu** gösterir. **Mesela:**

```
a = "Afyonkarahisar"
print(len(a))
```

Bu kod, "**`Afyonkarahisar`**" karakter dizisi içindeki harflerin sayısını **ekrana dökecektir**.

Bu fonksiyonu nerelerde kullanabiliriz? Mesela yazdığınız bir programa kullanıcıların giriş yapabilmesi için **parola** belirlemelerini istiyorsunuz. Seçilecek parolaların uzunluğunu sınırlamak istiyorsanız bu fonksiyondan yararlanabilirsiniz.

Hemen **örnek** bir kod yazalım:

```
a = input("Lütfen bir parola belirleyin: ")

if len(a) >= 6:
    print("Parola 5 karakteri geçmemeli!")
else:
    print("Parolanız etkinleştirilmiştir.")
```

`len()` fonksiyonunu yalnızca karakter dizileri ile birlikte kullandığımıza dikkat edin. İleride bu fonksiyonu başka *veri tipleri ile birlikte kullanmayı da* öğreneceğiz. Ancak henüz o veri tiplerini görmedik. Dolayısıyla şimdilik bu fonksiyonun **karakter dizileriyle birlikte kullanılabildiğini**, **ama sayılarla birlikte kullanılamadığını** bilmemiz yeterli olacaktır. Yani şöyle bir **örnek** bizi hüsrana uğratır:

```
>>> sayı = 123456
>>> len(sayı)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'int' has no len()
```

Burada gördüğümüz **hata mesajı**, bize tam sayı veri tipinin **`len()`** fonksiyonu ile birlikte **kullanılamayacağını** söylüyor. **Ama şöyle yaparsak olur:**

```
>>> sayı = 123456
>>> kardiz = str(sayı)
>>> len(kardiz)
6
```

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

BREAK DEYİMİ

break deyimi, **bir döngüyü sona erdirmek gerektiği zaman kullanılır**. Aşağıdaki örnek **break** deyiminin ne işe yaradığını açıkça gösteriyor:

```
kullanıcı_adı = "kullanici"
parola = "parola"

while True:
    soru1 = input("Kullanıcı adı: ")
    soru2 = input("Parola: ")

    if soru1 == kullanıcı_adı and soru2 == parola:
        print("Kullanıcı adı ve parolanız onaylandı.")
        break
    else:
        print("Kullanıcı adınız veya parolanız yanlış.")
        print("Lütfen tekrar deneyiniz!")
```

Bu programda **break** deyimi yardımıyla, kullanıcı adı ve parola doğru girildiğinde **parola sorma işleminin durdurulması** sağlanıyor. Yukarıdaki kodlar arasında, dikkat ederseniz daha önce bahsettiğimiz işleçlerden birini daha kullandık.

Kullandığımız bu işleç, "**ve**" anlamına gelen **and** işleci. Bu işlecin geçtiği satıra tekrar bakalım:

```
if soru1 == kullanıcı_adı and soru2 == parola:
    print("Kullanıcı adı ve parolanız onaylandı.")
```

Burada şu **Türkçe ifadeyi**, **Pythonca'ya çevirmiş olduk:**

"Eğer **soru1** değişkeninin değeri **kullanıcı_adı** değişkeniyle; **soru2** değişkeninin değeri de **parola** değişkeniyle aynı ise ekrana '**Kullanıcı adı ve parolanız onaylandı**,' cümlesini yazdır!"

Burada dikkat edilmesi gereken nokta şu: **and** işlecinin birbirine bağladığı **soru1** ve **soru2** değişkenlerinin ancak ikisi birden doğruysa o bahsedilen cümle *ekrana yazdırılacaktır*. Yani kullanıcı adı ve paroladan biri *yanlışsa* **if** deyiminin

gerektirdiği koşul yerine gelmemiş olacaktır. Okulda mantık dersi almış olanlar, bu **and** işlecini yakından tanıyor olmalılar. **and** işlecinin **karşıtı or** işlecidir. Bu işleç Türkçede **veya** anlamına gelir. Buna göre "**a veya b doğru ise**" dediğiniz zaman, bu **a** veya **b** ifadelerinden **birinin doğru olması** yetecektir. Şayet "**a ve b doğru ise**" derseniz burada hem **a**'nın hem de **b**'nin **doğru olması gerekir**.

continue Deyimi

Bu deyim ise **döngü içinde kendisinden sonra gelen her şeyin es geçilip döngünün en başına dönülmesini sağlar**. Basit bir **örnek** verelim:

```
while True:
    s = input("Bir sayı girin: ")

    if s == "iptal":
        break
    if len(s) <= 3:
        continue

    print("En fazla üç haneli bir sayı girin.")
```

Burada eğer kullanıcı klavyede "**iptal**" yazarsa **programdan çıkılacaktır**. **Bunu:**

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

satırıyla sağlamayı başardık.

Eğer kullanıcı tarafından girilen sayı **üç haneli** veya **daha az haneli** bir sayı ise **continue** deyiminin etkisiyle:

```
print("En fazla üç haneli bir sayı girin.")
```

satırı es geçilecek ve **döngünün en başına dönülecektir**.

Eğer kullanıcının girdiği sayıdaki hane, **üçten fazlaysa** ekrana "**En fazla üç haneli bir sayı girin.**" cümlesi yazdırılacaktır.

Bir **örnek** daha verelim:

```
sayı = "23430234230654602323340"

for i in sayı:
    if i == "0":
        continue
    else:
        print(i)
```

Bu kodlara göre öncelikle bir **`sayı`** değişkeni tanımlıyoruz ve bu değişken üzerinde bir **`for`** döngüsü kuruyoruz:

```
sayı = "23430234230654602323340"

for i in sayı:
```

Eğer **`i`** değişkeninin değeri "**`0`**" ise **`continue`** deyiminin etkisiyle herhangi bir işlem yapmadan **döngünün en başına dönüyoruz**:

```
if i == "0":
    continue
```

Eğer **`i`** değişkeninin değeri "**`0`**" dışında başka bir şey ise **bu değeri ekrana yazdırıyoruz**:

```
else:
    print(i)
```

Böylece **`sayı`** değişkeni içindeki bütün "**`0`**" değerlerini **elemiş oluyoruz**.

IN İŞLECİ

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

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

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

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

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

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

Bir de şuna bakalım:

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

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

Bu işlecin **gerçek hayatta nasıl kullanılabileceğine dair bir örnek** verelim:

```
cevap = input("Sistemden çıkmak istediğinize emin misiniz? [E/H] ")

if 'E' in cevap or 'e' in cevap:
    print("Güle güle!")
else:
    print("Sistemden çıkılmadı!")
```

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

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

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

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

ÖRNEK UYGULAMALAR

15

Buraya gelinceye kadar Python hakkında epey bilgi edindik. Bu esnada bazı basit programlar da yazdık. Ancak öğrendiğimiz bilgileri **pekiştirebilmemiz** için daha fazla **örnek uygulama** yapmamız gerekiyor. İşte bu bölümde, önceki derslerde öğrendiklerimizi somutlaştıran bazı deneme çalışmaları yapacağız.

Birkaç Pratik İpucu

Dilerseniz örnek programlar yazmaya geçmeden önce, komut satırına nasıl **daha kolay ulaşacağımıza** dair bazı ufak tefek **ipuçları** verelim.

Bildiğiniz gibi Windows'ta komut satırına ulaşmak için klavyedeki **Windows logolu tuşa** ve **R harfine** birlikte basarak açılan "**çalıştır**" penceresinde `cmd` komutunu veriyoruz.

Eğer arzu ederseniz Windows'un görev çubuğunda, başlatma düğmesinin yanında yer alan arama çubuğuna `cmd` yazarak da **komut satırına ulaşabilirsiniz**.

Bunun dışında Windows'un komut satırına ulaşmanın bir başka yolu, bilgisayarınızdaki **herhangi bir klasörü açıp** o klasörün adres çubuğunda `cmd` yazdıktan sonra **ENTER** tuşuna basmaktır.

Bu sayede komut satırını **o klasör altında çalıştırmış olursunuz**. Böylece yazdığınız programları içeren klasöre çok daha **kolay** bir şekilde ulaşabilirsiniz.

Eğer Windows'ta standart komut satırı yerine **PowerShell**'de çalışmak istiyorsanız klavyedeki **SHIFT** tuşuna basılı tutarak herhangi bir klasöre **sağ tıklayın**. Açılan menüden "**PowerShell penceresini burada açın**" seçeneğine tıklayarak bu klasör altında bir **PowerShell** oturumu açabilirsiniz.

PowerShell'e ulaşmanın bir başka yolu da Windows görev çubuğundaki **başlat düğmesine sağ tıklayarak** açılan menüden "**Windows PowerShell**" seçeneğini işaretlemektir.

Ubuntu kullanıcıları da herhangi bir klasöre sağ tıkladıklarında açılan menüde "**uçbirimde aç**" seçeneğine tıklayarak o klasör altında bir **komut satırı oturumu başlatabilir**.

Bu pratik ipuçlarını verdikten sonra, artık şu ana kadar öğrendiklerimizi pekiştirmemizi sağlayacak birtakım **örnek programlar** yazmaya başlayabiliriz.

Sayı Tahmin Oyunu

İlk olarak basit bir **sayı tahmin oyunu** yazacağız. Programımız rastgele bir sayı üretecek ve kullanıcıdan bu sayıyı tahmin etmesini isteyecek. Kullanıcının yaptığı tahminlere göre onu "**yukarı**" ve "**aşağı**" şeklinde yönlendirerek doğru sayıyı bulmasını sağlayacak.

Dilerseniz önce programımızın son hâlini **eksiksiz** bir şekilde gösterelim:

```
import random

sayı = random.randint(0, 100)

mesaj = """0 ile 100 arasında bir sayı tuttum.
Acaba hangi sayıyı tuttum?

(Oyundan çıkmak için q tuşuna basın)"""

print(mesaj)

while True:
    tahmin = input("Tahmininiz: ")

    if tahmin == 'q':
```

```
        print("Oyundan çıkılıyor...")
        break

    tahmin = int(tahmin)

    if tahmin == sayı:
        print("Tebrikler! Doğru sayı", sayı)
        break

    elif tahmin < sayı:
        print("Olmadı. Biraz daha yukarı")

    else:
        print("Olmadı. Biraz daha aşağı")
```

Her şeyden önce bu programı çalıştırıp nasıl kullanıldığını bir inceleyin. **Eksik ve kusurlu yanlarını tespit etmeye çalışın**.

Eğer incelemenizi bitirdiyseniz artık bu kodları satır satır ele almaya başlayabiliriz.

Kodlarımızın en başında şu satırı görüyoruz:

```
import random
```

`random` adlı modülden daha önce söz ettiğimizi hatırlıyorsunuz. Python'ın standart kütüphanesinde bulunan bu modül sayesinde **rastgele sayılar** üretebiliyoruz.

Bu modülü kullanabilmek için öncelikle bu modülü "**içe aktarmamız**" gerekiyor. Bunun için **`import random`** komutunu kullandık.

Daha sonra şöyle bir satırla karşılaşıyoruz:

```
sayı = random.randint(0, 100)
```

Esasında siz bu kodlara da yabancı değilsiniz. Yine daha önce **`random`** modülü içinde yer alan **`randint()`** adlı fonksiyondan da söz etmiştik.

Yukarıda sayı **= random.randint(0, 100)** kodunu kullanarak **0** ile **100** arasında rastgele bir sayı ürettik ve ürettiğimiz bu sayıyı **sayı adlı değişkene atadık**. Bu oyunu oynayan kişi işte bu değişkenin tuttuğu sayıyı tahmin etmeye çalışacak. Bu değişkenin değeri, programı her çalıştırdığımızda **farklı bir sayı olacak**.

Sırada şöyle bir kod görüyoruz:

```
mesaj = """0 ile 100 arasında bir sayı tuttum.
Acaba hangi sayıyı tuttum?

(Oyundan çıkmak için q tuşuna basın)"""

print(mesaj)
```

Burada oyunu oynayacak kişilere nasıl bir **mesaj** göstereceğimizi belirliyoruz ve belirlediğimiz **bu mesajı ekrana yazdırıyoruz**. Kullanıcılarımız programı çalıştırdıklarında ilk olarak **bu mesajı karşılarında görecek**.

Daha sonra, **program döngümüzü başlatıyoruz**:

```
while True:
```

Burada teknik olarak programımıza şu komutu vermiş olduk:

Aksi belirtilene kadar aşağıdaki kodları çalıştırmaya devam et!

Dolayısıyla kodlarımız, kullanıcı **sayıyı doğru tahmin edene kadar** veya '**q**' harfine basarak programdan çıkana kadar devam edecek.

Eğer **while True** satırını şöyle yazsaydık:

```
for i in range(10):
```

Bu durumda programımız **10 kez dönüp kapanırdı**. Ama biz **while True** dediğimiz için programımız **aksi belirtilene kadar çalışmaya devam edecek**.

Artık **while** döngüsü içinde çalışmasını istediğimiz kodları yazma aşamasına geçebiliriz...

Döngü içinde ilk olarak şu kodu görüyoruz:

```
tahmin = input("Tahmininiz: ")
```

Burada **`input()`** fonksiyonu yardımıyla kullanıcıdan **bir sayı tahmininde bulunmasını istiyoruz**.

Hatırlarsanız programın en başında kullanıcıya mesajımızı yazarken şöyle bir cümle kurmuştuk:

Oyundan çıkmak için q tuşuna basın.

İşte şimdi **`q`** tuşuna basıldığında programdan çıkılmasını sağlayacak kodları yazacağız:

```
if tahmin == 'q':
    print("Oyundan çıkılıyor...")
    break
```

Bu kodlara göre eğer kullanıcı '**`q`**' harfine basarsa **`break`** deyimi yardımıyla döngüyü kırarak programdan çıkacağız. Nezaket icabı, çıkıştan hemen önce kullanıcıya "**`oyundan çıkılıyor`**" şeklinde bir **mesaj** da gösteriyoruz.

Böylece kullanıcının '**`q`**' harfine bastığında ne olacağını belirlemiş olduk. Artık bu harfe basmadığı, yani **oyunu oynamaya devam etmek istediği durumda** ne yapacağımızı belirleyeceğiz. Bunun için öncelikle şu kodu yazıyoruz:

```
tahmin = int(tahmin)
```

Bildiğiniz gibi **`input()`** fonksiyonu bize bir karakter dizisi veriyor. Ama biz bu programda birtakım **aritmetik işlemler** yapacağımız için karakter dizilerine değil, **sayılara ihtiyaç duyuyoruz**. O yüzden **`input()`** fonksiyonundan gelen karakter dizisini **`int()`** fonksiyonu yardımıyla sayıya dönüştürüyoruz.

Şimdi de kullanıcının **sayıyı doğru tahmin ettiği durumda** ne olacağını belirleyen kodları yazalım:

```
if tahmin == sayı:
    print("Tebrikler! Doğru sayı", sayı)
    break
```

Eğer **`tahmin`** değişkeninin değeri **`sayı`** değişkeninin değerine eşit olursa yani **`input()`** fonksiyonundan gelen veri, **`random.randint()`** fonksiyonundan gelen veri ile **eşleşirse** kullanıcıya **doğru sayıyı tahmin ettiğine dair bir mesaj göstererek** **`break`** deyimi yardımıyla **döngüyü kırıp** programı sonlandırıyoruz.

Şimdi de kullanıcının **sayıyı doğru tahmin edemediği durumlarda** ne yapılacağını belirleyelim:

```
elif tahmin < sayı:
    print("Olmadı. Biraz daha yukarı")

else:
    print("Olmadı. Biraz daha aşağı")
```

Bu kodlara göre eğer kullanıcı, belirlenen sayıdan **daha düşük bir değer tahmin etmişse yukarı çıkmasını**, **aksi hâlde aşağı inmesini istiyoruz**.

Kelime Analizi

İkinci olarak bir **kelime analiz programı** yazacağız. Bu program, kullanıcıdan alınan kelimeleri analiz ederek bu kelime hakkında şu bilgileri verecek:

1. Kelimenin uzunluğu
2. Kelime içinde geçen harfler
3. Kelime içindeki harflerin sıklığı (*yani kelime içinde kaç kez geçtiği*)

Öncelikle kodlarımızı verelim:

```
kelime = input("Analiz edilecek kelimeyi girin: ")

uzunluk = len(kelime)
```

```
harfler = ''

for harf in kelime:
    if harf not in harfler:
        harfler += harf

sıklık = ''

for h in harfler:
    sayaç = 0
    for k in kelime:
        if h == k:
            sayaç += 1

    sıklık += f"\n\t\t{h} harfi {sayaç} kez geçiyor"

print("\n**********************************************************
*****\n")
print(f"\tkelime {uzunluk} harften oluşuyor")

print("\tkelimede ", end="")
print(*harfler, sep=", ", end=" ")
print("harfleri bulunuyor")

print("\tkelimede:")
print(sıklık)
print("\n**********************************************************
*****\n")
```

Birazdan bu kodları ayrıntılı bir şekilde ele alacağız. Ama şimdi, siz de bu programı çalıştırın ve nasıl işlediğini önce kendi kendinize anlamaya çalışın.

Program hakkında en azından bir fikir sahibi olduysanız incelememize geçebiliriz.

İlk olarak kullanıcıdan **analiz etmek üzere** bir kelime istiyoruz:

```
kelime = input("Analiz edilecek kelimeyi girin: ")
```

Daha sonra bu kelimenin uzunluğunu **`len()`** fonksiyonu yardımıyla bulup **`uzunluk`** adlı bir değişkene atıyoruz:

```
uzunluk = len(kelime)
```

Ardından bu kelime içinde **hangi harflerin bulunduğu tespit ediyoruz**. **Burada dikkat etmemiz gereken nokta şu:** Eğer kelime içinde aynı harf birden fazla geçiyorsa o harfi yalnızca *tek bir kez* listeleyeceğiz. Bunun için şu kodları kullanıyoruz:

```
harfler = ''

for harf in kelime:
    if harf not in harfler:
        harfler += harf
```

Burada ilk olarak **harfleri tutacak değişkeni tanımladık**:

```
harfler = ''
```

Bu değişken içinde **aynı harften yalnızca tek bir tane bulunacak**. Bunu sağlamak için kullanıcıdan aldığımız kelime içinde geçen harflere tek tek bakıp; **birden fazla olan harfleri teke indireceğiz**. Bu iş için bir **`for`** döngüsüne ihtiyacımız var:

```
for harf in kelime:
    if harf not in harfler:
        harfler += harf
```

Burada aslında şu Türkçe cümleyi **Pythonca'ya çevirmiş olduk**:

```
kelime içinde geçen her bir harfe tek tek bak:
    eğer bu harf, 'harfler' adlı değişken içinde yoksa:
        bu harfi 'harfler' adlı değişkene ekle
```

Bu kodlar, kelime içinde geçen bir harfin, **harfler** adlı değişkene yalnızca **tek bir kez** eklenmesini sağlıyor. Çünkü biz ilgili harfi, **harfler** değişkenine göndermeden önce bu harfin **harfler** değişkeni içinde bulunup bulunmadığını **if harf not in harfler:** kodu yardımıyla **kontrol ediyoruz**. Eğer **harfler** değişkenine göndermeye çalıştığımız harf, **zaten bu değişken içinde geçiyorsa**; kodlarımız o harfi **harfler** değişkenine **tekrar göndermeyecektir**.

Dilerseniz tekerleme gibi görünen bu karmaşık olayı somut bir örnek üzerinden anlatarak **basitleştirelim**.

Mesela kullanıcıdan gelen kelimenin '**adana**' olduğunu varsayalım. Bu durumda ilk harfimiz '**a**'. Şimdi, bu harfi yukarıdaki **for** döngüsü üzerinde gösterelim:

```
for 'a' in 'adana':
    if 'a' not in harfler:
        harfler += 'a'
```

O anda **harfler** değişkeni boş olduğu, yani bu değişken içinde bir '**a**' harfi **bulunmadığı için for** döngüsünün ilk turunda '**a**' harfi harfler değişkenine gönderilecek.

Döngünün ikinci turunda '**d**' harfi var. Bunu da yukarıdaki gibi gösterelim:

```
for 'd' in 'adana':
    if 'd' not in harfler:
        harfler += 'd'
```

harfler değişkeni içinde '**d**' harfi de olmadığı için bu harf de bu değişkene gönderiliyor. Şu anda **harfler** değişkeninin değeri '**ad**'.

Üçüncü turda ise yine bir '**a**' harfi görüyoruz. Ancak **harfler** değişkeni içinde zaten bir '**a**' harfi olduğu için bu kez bu harf, **harfler** değişkenine **gönderilmeyecek**.

Ardından '**n**' harfi geliyor. **harfler** değişkeninde '**n**' harfi de olmadığı için bu harf de değişkene gönderilecek. Şu anda değişkenin değeri '**adn**'.

Son olarak yine bir '**a**' harfiyle karşılaşıyoruz. Zaten değişken içinde '**a**' harfi olduğu için **bu harf de değişkene gönderilmiyor**. Böylece kullanıcıdan gelen kelime '**adana**' olduğunda, **harfler** değişkeninin değeri '**adn**' olmuş oluyor.

Sırada kelime içindeki bir harfin o kelimede **kaç kez geçtiğini** (*yani harflerin sıklığını*) **hesaplayan** kodları yazmaya geldi. **Dikkatlice bakın:**

```
sıklık = ''

for h in harfler:
    sayaç = 0
    for k in kelime:
        if h == k:
            sayaç += 1

    sıklık += f"\n\t\t{h} harfi {sayaç} kez geçiyor"
```

Bu kodlar, ilk bakışta gözünüze biraz karmaşık görünmüş olabilir. Ama aslında burada yaptığımız şey **kelime içindeki harflere tek tek bakmak** ve bir harfin kelime içindeki her geçişinde **sayaç** adlı **değişkenin değerini bir sayı artırmaktan ibarettir**. Bunun için öncelikle harflerin **sıklık değerlerini tutacak** olan karakter dizisini tanımlıyoruz:

```
sıklık = ''
```

Birazdan kuracağımız **for** döngüsü yardımıyla **kelimeye ilişkin sıklık bilgilerini** bu değişken içinde depolayacağız.

Şimdi, **for** döngümüzü kurmaya başlayabiliriz:

```
for h in harfler:
    sayaç = 0
```

Burada **harfler** değişkeni üzerinde bir **döngü kuruyoruz**. Yani **harfler** değişkeni içindeki **her bir harfe tek tek bakıyoruz**. Bu döngü içinde **sayaç** adlı bir değişken tanımladık. Bu değişken, kelime içindeki harflerin o kelimede **kaç kez geçtiği bilgisini** tutacak.

Şimdi de şu kodlara bakalım:

```
for k in kelime:
    if h == k:
        sayaç += 1
```

Bu defa da **kelime** değişkeni üzerinde **bir döngü kurduk**. Eğer **harfler** değişkeni içindeki bir harf, **kelime** değişkeni içindeki **bir harfle aynıysa** (*yani harfler değişkeni içindeki bir harf kelime değişkeni içinde de geçiyorsa*), **sayaç değişkeninin değerini 1 sayı artırıyoruz**.

Yukarıda anlattıklarımızı somutlaştırmak için bir **örnek** verelim. Mesela kullanıcıdan gelen kelimenin yine "**adana**" olduğunu varsayalım. Bu durumda **harfler** değişkeninin değerinin '**adn**' olacağını biliyorsunuz. **kelime** değişkeni ise kullanıcıdan gelen '**adana**' **karakter dizisi** olacak.

Şimdi "**adana**" karakter dizisini yukarıdaki kodlara uygulayalım:

```
sıklık = ''

for 'a' in 'adn':
    sayaç = 0
    for 'a' in 'adana':
        if 'a' == 'a':
            sayaç += 1
```

Yukarıda kurduğumuz **for** döngüsü, '**adana**' kelimesi içindeki '**a**' harfiyle toplam **3 kez karşılaşacak**. Dolayısıyla **sıklık** değişkeninin görünüşü şöyle olacak:

```
sıklık += f"\n\t\t{a} harfi {3} kez geçiyor"
```

Burada temel olarak '**adn**' kelimesi içindeki her bir harfi '**adana**' kelimesi içindeki harflerle karşılaştırıyoruz. Bunun için **iç içe iki tane for** döngüsü kuruyoruz. Döngü turları sırasında '**a**' harfi, **sayaç** değişkenini 3 kez artırırken; '**d**' ve '**n**' harfleri **birer kez artırıyor**. Dolayısıyla **sıklık** değişkeninin son değeri şöyle oluyor:

```
a harfi 3 kez geçiyor
d harfi 1 kez geçiyor
n harfi 1 kez geçiyor
```

Bu kodları kendi kendinize dikkatli bir şekilde inceleyin. Kodları daha iyi anlamak için **kodların çeşitli yerlerinde değişiklikler yapın**. Mesela bu kodlardaki kaçış dizilerini ne amaçla kullandığımızı kavramaya çalışın.

Not Ortalaması Hesaplayan Program

Son olarak öncekilere kıyasla **daha basit bir program** yazacağız. Yazacağımız program bir sınıftaki **öğrencilerin aldığı notların ortalamasını hesaplayacak**.

Önce kodlarımızın **son hâlini** verelim her zamanki gibi:

```
öğrenci_sayısı = int(input("lütfen öğrenci sayısını giriniz: "))
not_toplamı = 0

for i in range(öğrenci_sayısı):
    not_toplamı += int(input(f"{i+1}. öğrencinin notu: "))

ortalama = not_toplamı / öğrenci_sayısı
print("sınıfın not ortalaması: ", ortalama)
```

Bu notları öncelikle kendi kendinize anlamaya çalışın.

Eğer incelemeniz bittiyse bu kodları **satır satır gözden geçirmeye başlayabiliriz**.

Burada öncelikle sınıfta **toplam kaç öğrenci olduğu bilgisini** kullanıcıdan alıyoruz:

```
öğrenci_sayısı = int(input("lütfen öğrenci sayısını giriniz: "))
```

Bu bilgiyi, **not ortalamasını hesaplarken** kullanacağız. Alınan notların toplamını **öğrenci sayısına bölerek** not ortalamasını bulabileceğiz.

Sırada alınan notların **toplamını tutacak olan değişkenin tanımlanması** var:

```
not_toplamı = 0
```

Kullanıcıdan alacağımız **not bilgilerini** bu değişken içinde **toplayacağız**.

Şimdi de kullanıcıdan **öğrencilerin notlarını tek tek alalım**. Bunun için şöyle bir kod yazdık:

```
for i in range(öğrenci_sayısı):
    not_toplamı += int(input(f"{i+1}. öğrencinin notu: "))
```

Diyelim ki kullanıcı bize öğrenci sayısını **10** olarak verdi. Bu durumda yukarıdaki **for** döngüsü şu şekilde işleyecektir:

```
for i in range(10):
    not_toplamı += int(input(f"{i+1}. öğrencinin notu: "))
```

Burada öncelikle **0** ile **10** aralığındaki sayılar üzerinde bir **for** döngüsü kurduk.

Daha sonra da kullanıcıdan aldığımız not bilgilerini **art arda birbiriyle toplayıp** **not_toplamı** adlı değişkene yolladık.

Ayrıca **i** değişkeninin değerini kullanarak girilen notun **kaçıncı öğrencinin notu olduğu** bilgisini de gösterdiğimize dikkat edin:

```
f"{i+1}. öğrencinin notu: ")
```

Burada kodumuzu **i+1** şeklinde yazmamızın nedeni tabii ki Python'ın saymaya 0'dan başlıyor olmasıdır. İlk döngüde '**0. öğrencinin notu**' gibi garip bir cümle ortaya çıkmaması için **i değişkeninin değerini bir sayı artırdık**.

Gelin, isterseniz yukarıdaki kodları biraz somutlaştıralım. İlk döngü şöyle gerçekleşir:

```
for i in range(10):
    not_toplamı += int(input(f"{0+1}. öğrencinin notu: "))
```

Diyelim ki kullanıcı ilk öğrencinin notunu **80** olarak girdi. Bu durumda **not_toplamı** değişkenine giden değer **80** olacaktır.

Dolayısıyla ilk döngü sonunda **not_toplamı** değişkeninin değeri **80** oldu. **İkinci döngü** ise şöyle gerçekleşir:

```
for i in range(10):
    not_toplamı += int(input(f"{1+1}. öğrencinin notu: "))
```

İkinci öğrencinin notu da **70** olsun. **not_toplamı** değişkeninin **80** olan değerine **70** de eklenince bu değişkenin değeri **150** olur.

Onuncu öğrencinin notu da girildikten sonra artık **toplam notu öğrenci sayısına bölerek ortalamayı bulabiliriz**:

```
ortalama = not_toplamı / öğrenci_sayısı
print("sınıfın not ortalaması: ", ortalama)
```

Bu kodları da yazıp anladığımıza göre artık bir sonraki konumuza geçebiliriz.

Veri Tipleri

Şu ana kadar öğrendiklerimizi örneklendiren programlar yazdıktan sonra, gelin isterseniz biraz da bazı **teknik konulardan** söz edelim. Mesela Python programlama dilindeki "**veri tipi**" kavramını inceleyelim.

Python programlama dilinde her verinin bir tipi bulunur. Bir verinin tipi, o veriyle hangi işlemleri yapıp hangi işlemleri yapamayacağımızı belirler. Dolayısıyla o anda **elimizde bulunan bir verinin tipini bilmek son derece önemlidir**.

Python'da verilerin tipini **`type()`** adlı bir fonksiyon yardımıyla **sorgulayabiliriz**. **Örneğin**:

```
>>> n = 5
>>> type(n)
<class 'int'>
```

Demek ki **`n`** değişkeninin tuttuğu verinin tipi bir '**`int`**', yani **tam sayı** imiş...

Bir de şuna bakalım:

```
>>> isim = 'istihza'
>>> type(isim)
<class 'str'>
```

Buradan da **`isim`** değişkeninin tuttuğu verinin bir '**`str`**', yani **karakter dizisi** olduğunu anlıyoruz.

Peki ya şu?

```
>>> f = 5.9
>>> type(f)
<class 'float'>
```

Bu da bize, **`f`** değerinin bir '**`float`**', yani **kayan noktalı sayı** olduğunu söylüyor.

Yukarıdakiler bizim **kendi ürettiğimiz veriler**di. Ayrıca bu veriler bize başka kaynaklardan da geliyor olabilir.

Mesela **`input()`** fonksiyonundan gelen verinin **bir karakter dizisi** olduğunu biliyoruz. **Örneğin**:

```
veri = input("veri: ")
print(type(veri))
```

Bu kodları çalıştırdığımızda şu **çıktıyı** alıyoruz:

```
<class 'str'>
```

Dolayısıyla eğer bu veriyle bir **aritmetik işlem** yapmak istiyorsak öncelikle **`int()`** fonksiyonunu kullanarak bu veriyi bir **sayıya çevirmemiz gerekiyor**.

Aynı şekilde **`random`** adlı modülün **`randint()`** fonksiyonunun da bizden parametre olarak sayı tipinde **iki adet veri beklediğini**, dolayısıyla bu fonksiyon içinde kullanacağımız farklı tipteki verileri, öncelikle **sayıya dönüştürmemiz gerektiğini** biliyoruz:

```
import random

a = int(input("ilk sayı: "))
b = int(input("ikinci sayı: "))

if a > b:
    print("ilk sayı ikinci sayıdan küçük olmalı")
else:
    print(f"{a} ile {b} arasında üretilen rastgele sayı: ", random.
randint(a, b))
```

Burada öncelikle kullanıcıdan **iki adet sayı** istedik. Bu sayıları **`random`** modülünün **`randint()`** fonksiyonuna argüman olarak verip bu iki sayı aralığında **rastgele bir sayı** üreteceğiz. Ancak **`randint()`** fonksiyonunun çalışabilmesi için hem argümanların **sayı tipinde** olması hem de verilen ilk sayının **ikinci sayıdan küçük olması** gerekiyor. Yani mesela bu fonksiyonu kullanarak 20 ile 60 arasında rastgele bir sayı üretebilirsiniz, ama 60 ile 20

arasında rastgele bir sayı üretemezsiniz. Bu yüzden kullanıcının, **yanlışlıkla ilk sayıyı ikinci sayıdan büyük girmesi ihtimaline karşı**, önce `if` ve `else` deyimleri yardımıyla kullanıcıdan gelen ilk verinin, **ikinci veriden küçük olup olmadığını kontrol ediyoruz**:

```
if a > b:
    print("ilk sayı ikinci sayıdan küçük olmalı")
```

Burada eğer `a` değeri `b` değerinden **büyükse** kullanıcıyı uyarıyoruz.

Ayrıca yukarıdaki kodlarda `a` ve `b` değerlerini `input()` fonksiyonu yardımıyla kullanıcıdan alırken `int()` fonksiyonuyla da **dönüştürme** işlemi yaptığımıza dikkat edin.

Bir başka örnek ise `len()` fonksiyonudur. Bu fonksiyonun önemli bir özelliğinin **sayılar üzerinde kullanılamaması** olduğunu daha önce söylemiştik. Dolayısıyla bir sayının uzunluğunu `len()` fonksiyonu yardımıyla öğrenebilmek için öncelikle o sayıyı bir **karakter dizisine** dönüştürmemiz gerekiyor:

```
>>> n = 123456
>>> len(n)
```

Bu komutlar **hata** verecektir. Burada öncelikle `n` değerini **karakter dizisine çevirmemiz gerekir**:

```
>>> len(str(n))
6
```

Python'daki her veri tipinin **kendine göre bir kullanım alanı vardır**. Dolayısıyla yazdığımız programlarda, **doğru iş için doğru veri tipini kullanmaya dikkat etmemiz gerekir**. Zira bazı işlemleri yapmak, bazı veri tipleri ile çok kolayken başka veri tipleri ile çok zor veya imkânsız olabilir.

Biz şimdiye kadar veri tipleri olarak yalnızca karakter dizileri, tam sayılar ve kayan noktalı sayılardan söz ettik. Ancak Python'da bunların dışında **başka veri tipleri de** bulunur. İlerleyen derslerde bu veri tiplerinden de söz edeceğiz.

HELP() FONKSİYONU

Bu bölüme gelinceye kadar Python'ın standart kütüphanesinde yer alan bazı modüllerden söz ettik. Bu modüller şunlardı: **`platform`**, **`webbrowser`**, **`datetime`** ve **`random`**.

Ayrıca standart kütüphanede, bunların dışında başka pek çok modülün de bulunduğunu söylemiştik.

Burada şöyle bir soru aklımıza geliyor: *Acaba biz bu modüllerin ne işe yaradığını, bu modüllerin hangi fonksiyon ve niteliklere sahip olduklarını nereden öğreneceğiz?*

Bir **modül** (*veya herhangi bir fonksiyon*) hakkında bilgi edinmek istediğimizde, elbette ilk başvuracağımız kaynak *İnternet* olacak. İnternet üzerinde hangi iş için hangi modülü veya fonksiyonu kullanmanız gerektiği, bu modül veya fonksiyonların nasıl kullanılacağı veya bunların hangi ögeleri barındırdığı hakkında oldukça ayrıntılı bilgilere ulaşabilirsiniz.

Python programlama dilinde, İnternet kaynakları dışında bize bir modül veya fonksiyon hakkında hızlı bir şekilde bilgi sağlayan **`help()`** adlı bir **hazır fonksiyon da bulunur**. Bu fonksiyonu şöyle kullanıyoruz:

Diyelim ki **`print()`** fonksiyonu hakkında **bilgi sahibi olmak istiyoruz**. Bunun için şu komutu verebiliriz:

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

Bu komut bize şöyle bir **çıktı** veriyor:

```
Help on built-in function print in module builtins:

print(...)
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file:  a file-like object (stream); defaults to the current sys.
stdout.
```

```
sep:   string inserted between values, default a space.
end:   string appended after the last value, default a newline.
flush: whether to forcibly flush the stream.
```

Bu kısa bilgilendirme yazısı bize **`print()`** fonksiyonunun **ne işe yaradığını ve hangi argümanları aldığını sade bir şekilde gösteriyor**. Elbette bu yazılanları anlayabilmek için *belirli bir düzeyde İngilizce bilgisine sahip olmak gerekiyor*. Ancak İnternet üzerinde bu konuyla ilgili **Türkçe kaynaklara da ulaşma imkânınız var**.

Aynı şekilde **`help()`** fonksiyonunu kullanarak bir modül hakkında da bilgi edinebilirsiniz. Mesela **`random`** modülü hakkında yardım alalım:

```
>>> import random
```

Elbette bir modül üzerinde herhangi bir işlem yapabilmek için o **modülü içe aktarmamız gerekiyor**. O yüzden öncelikle **`random`** modülünü içe aktardık.

Artık bu modül hakkında yardım alabiliriz:

```
>>> help(random)
```

Bu komut da bize modül hakkında epey ayrıntılı sayılabilecek teknik bilgiler sunacaktır.

`help()` fonksiyonu dışında, daha önce öğrendiğimiz **`dir()`** fonksiyonunun da Python'daki herhangi bir öge hakkında **fikir sahibi** olmamıza yardımcı olabileceğini biliyoruz. Bu fonksiyonu kullanarak herhangi bir ögenin içeriğini sorgulayabiliyoruz. Böylece ilgili ögenin bize **neler sunduğunu** görebiliyoruz:

```
>>> import datetime
>>> dir(datetime)
```

Ayrıca **`dir()`** ile sorgulayıp bulduğumuz ögeler üzerinde yine **`help()`** fonksiyonunu uygulayarak o ögeler hakkındaki **yardım bilgilerine** de ulaşabiliriz. **Mesela**:

```
>>> import webbrowser
>>> dir(webbrowser)
```

Bu komutla **`webbrowser`** modülü içinde hangi ögelerin olduğunu sorguladık. Diyelim ki bu listede görünen **`open`** ögesi hakkında bilgi edinmek istiyoruz. O zaman da şu komutu kullanabiliriz:

```
>>> help(webbrowser.open)
```

Daha önce de söylediğimiz gibi **`help()`** fonksiyonundan yararlanabilmeniz için **belli düzeyde İngilizce biliyor olmanız şart**. Ancak yine dediğimiz gibi **Python ve Python'ın ögeleri** hakkında İnternet üzerinde **Türkçe kaynaklara da ulaşmanız mümkün**.

LİSTELER

16

Bu bölümde '**liste**' adlı bir veri tipinden bahsedeceğiz. Esasında biz önceki derslerimizde de birkaç örnekte listelerden üstünkörü de olsa söz etmiştik. Dolayısıyla siz en azından **liste kavramına** tamamen yabancı sayılmazsınız.

Mesela önceki bölümlerde **dir()** adlı bir fonksiyondan söz etmiştik. Daha önce bu fonksiyonu, Python'daki modüllerin içeriğini sorgulamak için kullandığımızı hatırlıyorsunuz. Bu fonksiyonla ilgili basit bir **örnek** verelim:

```
>>> import datetime
>>> dir(datetime)
['MAXYEAR', 'MINYEAR', '__builtins__', '__cached__', '__doc__', '__
file__',
'__loader__', '__name__', '__package__', '__spec__', 'date',
'datetime',
'datetime_CAPI', 'sys', 'time', 'timedelta', 'timezone', 'tzinfo']
```

Gördüğünüz gibi **dir()** fonksiyonunun verdiği **çıktı**, köşeli parantezler içinde birbirlerinden **virgül** ile ayrılmış birtakım ögelerden oluşuyor. İşte bu tür ögelerden oluşan veri tipine Python'da **liste** adını veriyoruz.

Biz de köşeli parantezler içinde birbirlerinden **virgül** ile ayrılmış birtakım ögeler tanımlayarak kendi listelerimizi oluşturabiliriz. Mesela şöyle bir **liste tanımlayalım**:

```
>>> urller = [“google”, “yahoo”, “bing”, “yandex”]
```

Gördüğünüz gibi bir liste tanımlamak son derece kolay bir iş.

Yukarıdaki şekilde listemizi tanımladıktan sonra şimdi **etkileşimli kabukta:**

```
>>> urller
```

komutunu verdiğimizde tanımladığımız **urller** adlı listenin ögeleri ekrana yazdırılacaktır.

Yukarıdaki örnekte **dört tane karakter** dizisinden oluşan bir **liste** tanımladık. Eğer arzu ederseniz listeler içinde birbirinden farklı tiplerde verileri de aynı anda depolayabilirsiniz. **Örneğin**:

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

Burada üç tane **karakter dizisi**, iki tane de **tam sayıdan** oluşan bir liste tanımladık.

Hatta eğer isterseniz bir **liste** içine başka bir veya **birden fazla liste** bile yerleştirebilirsiniz.

Mesela aşağıdaki gibi **3x3’lük bir matris** oluşturabilirsiniz:

```
>>> matris = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Daha önce de söylediğimiz gibi burada dikkat etmemiz gereken nokta, liste ögelerini tanımlarken **köşeli parantezler** kullanıyor olmamız. Ayrıca liste içindeki ögeleri birbirinden **virgül** ile ayırdığımıza da dikkat ediyoruz.

Tanımladığımız listelerin öge sayısını, daha önce öğrendiğimiz **len()** fonksiyonu yardımıyla sorgulayabiliriz:

```
>>> len(liste)
5

>>> len(matris)
3
```

Gelin, isterseniz tam bu noktada hem daha önce öğrendiklerimizi **pekiştirmek** hem de yeni öğrendiklerimizi özümseyebilmek için *ufak bir uygulama yapalım*.

Şimdi **boş bir metin düzenleyici** açarak şu komutları yazalım:

```
import webbrowser

urller = [“google”, “yahoo”, “bing”, “yandex”]

for url in urller:
    webbrowser.open(f”www.{url}.com”)
```

Bu kodları çalıştırdığınızda *google.com*, *yahoo.com*, *bing.com* ve *yandex.com* adresleri bir İnternet tarayıcısı içinde **aynı anda açılacaktır**.

Burada **webbrowser** modülünü programımızda kullanabilmek için öncelikle bu modülü **içe aktardık**:

```
import webbrowser
```

Daha sonra **url** adlarını içeren listemizi tanımladık:

```
urller = [“google”, “yahoo”, “bing”, “yandex”]
```

Amacımız bu listedeki her bir ögeyi kullanarak *www.google.com*, *www.yahoo.com*, *www.bing.com* ve *www.yandex.com* gibi İnternet adresleri elde etmek. Bu adresleri bu şekilde elde ettikten sonra bunları **webbrowser** modülünün **open()** fonksiyonuna **parametre** olarak göndereceğiz. Böylece bu İnternet adresleri tarayıcı üzerinde **tek tek açılacak**.

Bu iş için **urller** adlı liste üzerinde bir **for** döngüsü kurmamız gerekiyor:

```
for url in urller:
    webbrowser.open(f"www.{url}.com")
```

Bu **for** döngüsü içinde **f**'li karakter dizilerinden yararlanarak istediğimiz İnternet adreslerini bir çırpıda oluşturduk:

```
f"www.{url}.com"
```

Bu noktada size şöyle bir soru sormama izin verin: Acaba **liste** adlı veri tipini hiç bilmiyor olsaydık böyle bir kodu nasıl yazardık.

Dikkatlice bakın:

```
import webbrowser

url1 = "google"
url2 = "yahoo"
url3 = "bing"
url4 = "yandex"

for url in url1, url2, url3, url4:
    webbrowser.open(f"www.{url}.com")
```

Gördüğünüz gibi bu kodları hiç **liste** veri tipi kullanmadan da yazabiliyoruz. Ancak bu kodlar listeli kodla aynı çıktıyı verse de liste ile yazılan kodlar kadar esnek değildir. Diyelim ki biz bu programa başka bir web sitesi daha eklemek istiyoruz. Mesela bu kodlara **DuckDuckGo** web sitesini de ekleyelim. **Önce listeli kodlarımızı görelim**:

```
import webbrowser

urller = ["google", "yahoo", "bing", "yandex", "duckduckgo"]

for url in urller:
    webbrowser.open(f"www.{url}.com")
```

Burada urller listesinin en sonuna "**duckduckgo**" karakter dizisini eklememiz yeterli oldu. Aynı işlemi **listesiz kodlarla** yapmak istersek şöyle bir şey yazmamız gerekir:

```
import webbrowser

url1 = "google"
url2 = "yahoo"
url3 = "bing"
url4 = "yandex"
url5 = "duckduckgo"

for url in url1, url2, url3, url4, url5:
    webbrowser.open(f"www.{url}.com")
```

Gördüğünüz gibi **listesiz kodlara ekleme yaparken iki yerde değişiklik yapmamız gerekiyor.** Önce beşinci url'yi **url5 = "duckduckgo"** koduyla tanımlıyoruz. Ardından da **for** döngüsü içinde bu **url5** adlı değişkeni belirtiyoruz.

Peki, acaba **for** döngülerini de bilmiyor olsaydık ne yapardık? Böyle bir durumda kodları şöyle yazabilirdik:

```
import webbrowser

url1 = "google"
url2 = "yahoo"
url3 = "bing"
url4 = "yandex"
url5 = "duckduckgo"

webbrowser.open(f"www.{url1}.com")
webbrowser.open(f"www.{url2}.com")
webbrowser.open(f"www.{url3}.com")
webbrowser.open(f"www.{url4}.com")
webbrowser.open(f"www.{url5}.com")
```

veya daha basitçe:

```
import webbrowser

webbrowser.open("www.google.com")
webbrowser.open("www.yahoo.com")
webbrowser.open("www.bing.com")
webbrowser.open("www.yandex.com")
webbrowser.open("www.duckduckgo.com")
```

Belki bu kadar küçük bir programda çok fazla dikkati çekmiyordur, ancak aslında **son derece verimsiz** ve **acemice** bir kod yazma biçimidir bu. Üstelik bu şekilde, kodlarımız tüm esnekliğini de yitiriyor. Ayrıca bu kodlarda gereğinden fazla **tekrar** var. Bir program yazarken her zaman, yaptığımız iş için **en uygun** ve **en pratik** olan araçları seçmeye dikkat etmeliyiz.

Listeler hakkında genel birtakım bilgiler verdiğimize göre şimdi; *listeleri nasıl yöneteceğimizi, yani listeye öge ekleme, listeden öge çıkarma, listenin ögelerine ulaşma, listenin ögelerini değiştirme gibi işlemleri nasıl yapacağımızı* incelemeye geçebiliriz.

Listelerin Metotları

Python programlama dilinde her veri tipinin **birtakım metotları** bulunur. Metotlar, bir **veri tipi üzerinde çeşitli işlemler yapabilmemizi sağlayan ögeler**dir.

Bir veri tipinin hangi metotları olduğunu görmek için her zamanki gibi **`dir()`** fonksiyonundan yararlanabilirsiniz. Mesela listelerin hangi metotlara sahip olduğunu görmek için şöyle bir kod yazabiliriz:

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

Python'daki **`liste`** veri tipi '**`list`**' kelimesi ile gösterilir. Dolayısıyla **`dir()`** fonksiyonuna '**`list`**' kelimesini **argüman** olarak verirseniz **`liste`** veri tipinin metotlarını inceleyebilirsiniz.

Aynı şekilde eğer isterseniz herhangi bir listeyi de **dir()** fonksiyonuna **argüman** olarak verebilir, böylece o **listenin metotlarını görebilirsiniz**:

```
>>> liste = [2, 4, 6, 8, 10]

>>> dir(liste)
```

Hatta bu iş için listelerin ayırt edici özelliği olan **köşeli parantezleri** bile kullanabilirsiniz:

```
>>> dir([])
```

Yukarıdaki komutlardan herhangi birini verdiğimizde şöyle bir **çıktı** elde ediyoruz:

```
['__add__', '__class__', '__contains__', '__delattr__', '__
delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__
getattribute__',
'__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__
init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__
mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__reversed__', '__rmul__', '__setattr__', '__setitem__', '__
sizeof__',
'__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count',
'extend',
'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

Burada bizi ilgilendiren metotlar, başında ve sonunda '__' işareti taşımayanlar. Bizi ilgilendiren metotları **programatik** olarak nasıl ayıklayabileceğinizi biliyorsunuz:

```
>>> for metot in dir(list):
...     if "__" not in metot:
...         print(metot)
...
```

```
append
clear
copy
count
extend
index
insert
pop
remove
reverse
sort
```

Burada **if** deyimini nasıl kullandığımıza dikkat edin. Yukarıdaki kodlarda aslında şu Türkçe cümleyi **Pythonca'ya çevirmiş olduk:**

```
>>> dir(list) içindeki her bir metoda 'metot' adını ver:
...     eğer 'metot' adı içinde '__' işaretleri yoksa:
...         'metot' adını ekrana yazdır
```

İşte bu bölümde, yukarıdaki çıktıda görünen metotları tek tek ele alacağız.

APPEND()

İlk metodumuz **append()**. Bu kelime Türkçede "**eklemek**", "**iliştirmek**" gibi anlamlara geliyor.

Bir listeye yeni bir öge eklemek için **append()** metodundan faydalanıyoruz.

Örnek olarak şöyle bir liste oluşturalım:

```
>>> kripto = ["bitcoin", "ethereum", "litecoin"]
```

Şimdi bu listeye '**ripple**' adlı yeni bir öge ekleyelim:

```
>>> kripto.append("ripple")
>>> kripto
['bitcoin', 'ethereum', 'litecoin', 'ripple']
```

Gördüğünüz gibi **`append()`** metodu, kendisine argüman olarak verilen '**`ripple`**' ögesini listenin **en sonuna** ekledi.

Ancak **`append()`** metodu yalnızca tek bir argüman alır. Eğer bu metodu kullanarak bir listeye birden fazla yeni öge eklemek istiyorsanız elbette **`for`** **döngülerinden yararlanabilirsiniz**:

```
>>> yeni_ögeler = [“cardano”, “stellar”, “chainlink”]
>>> for öge in yeni_ögeler:
...     kripto.append(öge)
...
>>> kripto
[‘bitcoin’, ‘ethereum’, ‘litecoin’, ‘ripple’, ‘cardano’, ‘stellar’,
‘chainlink’]
```

Eğer siz acemi bir programcı olsaydınız '**`cardano`**', '**`stellar`**' ve '**`chainlink`**' ögelerini listeye şöyle eklerdiniz:

```
yeni_ögeler.append(‘cardano’)
yeni_ögeler.append(‘stellar’)
yeni_ögeler.append(‘chainlink’)
```

Ama artık **acemi bir programcı değilsiniz** ve ögeleri böyle tek tek eklemek yerine **listelerden** ve **`for`** döngülerinden yararlanmanın çok daha akıllıca bir iş olduğunu gayet iyi biliyorsunuz.

`append()` metodu, listelerin son derece **pratik** ve **faydalı** bir metodudur. Hatta listelerin belki de en sık kullanılan metodu **`append()`** metodudur desek hiç de abartmış olmayız.

Gelin, isterseniz listelerin bu önemli metodu ile ilgili ufak bir **uygulama çalışması** yapalım.

Amacımız bir sınıftaki öğrencilerin **ad** ve **soyad bilgilerinden oluşan bir liste meydana getirmek**. Bunun için şöyle bir program yazabiliriz:

```
print(“Listeye eklenecek öğrencinin adı ve soyadı:”)
print(“(listeyi tamamlamak için ‘q’ tuşuna basın)”)
```

```
liste = []
sayaç = 0

while True:
    sayaç += 1
    öğrenci = input(f"öğrenci {sayaç}: ")

    if öğrenci == 'q' and not liste:
        break

    elif öğrenci == 'q' and liste:
        print("\n")
        print("----------------------------")
        print("ÖĞRENCİ LİSTESİ OLUŞTURULDU:")
        print("----------------------------")

        for öğrenci in liste:
            print(öğrenci)

        print("\nToplam öğrenci sayısı", len(liste))

        break

    else:
        liste.append(öğrenci)
```

İsterseniz şimdi de bu programı **satır satır inceleyerek** her kodun ne iş yaptığını tek tek anlamaya çalışalım.

Burada öncelikle kullanıcıdan ne beklediğimizi belirten mesajlarımızı ekrana yazdırdık:

```
print("Listeye eklenecek öğrencinin adı ve soyadı:")
print("(listeyi tamamlamak için 'q' tuşuna basın)")
```

Buna göre kullanıcı bize bir **öğrenci adı ve soyadı bilgisi** verecek. Tüm öğrencilerin ad ve soyad bilgilerini girdikten sonra da '**q**' tuşuna basarak listeyi tamamlayacak ve programdan çıkacak.

Daha sonra **iki tane değişken** oluşturuyoruz:

```
liste = []
sayaç = 0
```

liste adlı değişken, **öğrencilerin ad ve soyad bilgilerini tutacak**. **sayaç** adlı değişkeni ise o anda **kaçıncı öğrencinin bilgilerinin girildiğini kullanıcıya gösterebilmek için kullanacağız**. Bu değişkenin başlangıç değerini **0** olarak belirledik.

Şimdi, **döngümüzü başlatıyoruz:**

```
while True:
    sayaç += 1
    öğrenci = input(f"öğrenci {sayaç}: ")
```

Burada her döngüde **sayaç** değişkeninin değerini **1** sayı artırıyoruz:

```
sayaç += 1
```

Bu değeri, sonraki satırda **kullanıcıdan veri alırken input()** fonksiyonu içinde kullanıyoruz:

```
öğrenci = input(f"öğrenci {sayaç}: ")
```

İsterseniz buraya kadar olan kodları bir çalıştırın.

Bu kodları çalıştırdığınızda şöyle bir **çıktı** alacaksınız:

```
Listeye eklenecek öğrencinin adı ve soyadı:
(listeyi tamamlamak için 'q' tuşuna basın)
öğrenci 1:
```

Burada kullanıcı, listeye eklenmek üzere öğrencinin adını ve soyadını girecek. Kullanıcının, herhangi bir şekilde **programdan çıkabilmesi için** '**q**' tuşuna **basması gerekiyor**.

Artık kullanıcının gireceği **verileri toplama aşamasına** geçebiliriz.

Şimdi, öncelikle programımızın kullanıcı tarafından **herhangi bir anda sonlandırılabilmesi için** bir koşul tanımlayalım:

```
if öğrenci == 'q' and not liste:
    break
```

Buna göre eğer öğrenci değeri '**q**' ise ve eğer liste boş ise yani kullanıcı, öğrenci listesine herhangi bir veri girmeden '**q**' tuşuna basarak **programdan çıkmak istemişse** `break` deyimi yardımıyla döngüyü sonlandırıp programdan çıkıyoruz.

Bir de kullanıcının listeye **öğrenci ekledikten sonra** '**q**' tuşuna bastığı ve **liste oluşturma işlemini sonlandırarak** programdan çıkmak istediği durum için bir koşul oluşturalım:

```
elif öğrenci == 'q' and liste:
    print("\n")
    print("----------------------------")
    print("ÖĞRENCİ LİSTESİ OLUŞTURULDU:")
    print("----------------------------")

    for öğrenci in liste:
        print(öğrenci)

    print("\nToplam öğrenci sayısı", len(liste))

    break
```

Buna göre eğer kullanıcıdan gelen veri '**q**' ise ve liste de boş değilse yani eğer **kullanıcı listeye birtakım ögeler eklemiş** ve **artık programdan çıkmak istiyorsa;** bu durumda, kullanıcıya öğrenci listesinin oluşturulduğu bilgisini verip bir `for` döngüsü içinde listedeki tüm öğrencileri ekrana yazdırıyoruz. Ayrıca **bilgi olarak listedeki öğrenci sayısını da gösteriyoruz**.

Tüm bu işlemleri tamamladıktan sonra da `break` deyimini kullanarak döngüyü sonlandırıyoruz.

Eğer kullanıcı '**q**' tuşuna basıp programdan çıkmak yerine listeye eklenmek üzere ad ve soyad bilgisi vermişse bu durumda da **`append()`** metodunu kullanarak **kullanıcıdan gelen verileri listeye ekliyoruz**:

```
else:
    liste.append(öğrenci)
```

Programımızı yazdık, ancak bu programda önemli bir **eksiklik** var. Eğer kullanıcı herhangi bir veri girişi yapmadan doğrudan **ENTER** düğmesine basarsa bu boş veri de **öğrenci listesine ekleniyor** ve **program ilerlemeye devam ediyor**. Böyle bir durumda listeye boş veri girilemeyeceği konusunda **kullanıcının uyarılması** ve '**`öğrenci 1`**', '**`öğrenci 2`**' şeklinde ilerleyen satırların **ilerlemesinin durdurulması daha doğru olacaktır**. Programda bu sorunu çözecek bir değişikliğin nasıl yapılabileceği üzerinde düşünün.

Böylece listelerin **`append()`** metodunu öğrenmiş olduk. Birazdan bir başka metodu incelemeye geçeceğiz. Ama ondan önce isterseniz Python'da, **bir listeye öge eklemenin bir başka yolunu** daha gösterelim.

Python'da bir listeye öge eklemek için **`append()`** metodunun dışında **`artı (+)`** işlecinden de yararlanabiliriz.

Bunun nasıl yapılacağını göstermek için daha önce tanımladığımız şu listeyi kullanalım:

```
>>> kripto = [“bitcoin”, “ethereum”, “litecoin”]
```

Bu listeye **+** işlecini kullanarak yeni bir öge eklemek için şu komutu yazıyoruz:

```
>>> kripto + [“ripple”]

[‘bitcoin’, ‘ethereum’, ‘litecoin’, ‘ripple’]
```

Burada, **bir listeye yine bir başka liste eklediğimize dikkat edin**. **`kripto`** adlı listeye yeni bir öge eklemek için **+** işlecini kullanarak şöyle bir kod **YAZAMAYIZ**:

```
>>> kripto + “ripple”
```

Bu kod **hata** verecektir. Çünkü bu yöntemi kullanarak bir listeye ancak başka bir liste ekleyebilirsiniz. Dolayısıyla ekleyeceğimiz ögeyi önce listeye çevirmemiz gerekiyor.

Ancak bir listeye **+** işleci ile yeni bir öge eklediğimizde bu ekleme işlemi **asıl liste üzerinde değişiklik yapmayacaktır.** Yani:

```
>>> kripto

[“bitcoin”, “ethereum”, “litecoin”]
```

Gördüğünüz gibi ‘**ripple**’ ögesi aslında **kripto** adlı listeye kalıcı olarak eklenmemiş. Ekleme işleminin **kalıcı olmasını sağlamak için** şöyle bir kod yazabilirsiniz:

```
>>> kripto = kripto + [“ripple”]
>>> kripto
[‘bitcoin’, ‘ethereum’, ‘litecoin’, ‘ripple’]
```

Burada ‘**ripple**’ ögesini **kripto** adlı listeye ekledikten sonra bu değişikliğin kaybolmaması için bu eklemeyi içeren **kripto** adlı yeni bir liste oluşturduk.

Yukarıdaki kodları şöyle de yazabilirdiniz:

```
>>> kripto += [“ripple”]
```

Bu kod şununla **aynı anlama gelir**:

```
>>> kripto = kripto + [“ripple”]
```

+ işleci bir listeye **birden fazla öge eklememize de izin verir**:

```
>>> kripto += [“tron”, “iota”]
>>> kripto
[‘bitcoin’, ‘ethereum’, ‘litecoin’, ‘ripple’, ‘tron’, ‘iota’]
```

Gördüğünüz gibi Python’da bir listeye öge eklemenin birden fazla yolu var. Siz, hangi yöntem kolayınıza geliyorsa onu tercih edebilirsiniz.

Şimdi, dilerseniz bu konuyu burada keselim ve **`append()`** dışındaki metotları da inceleyelim.

EXTEND()

'**`Extend`**' kelimesi Türkçede '**genişletmek**', '**uzatmak**' gibi anlamlara gelir. İşte listelerin **`extend()`** metodu da oluşturduğumuz listeleri genişletmemizi veya uzatmamızı sağlayacak.

Bu metodun işlevini anlatabilmenin en iyi yolu tabii ki **örnekler üzerinde çalışmak**.

Öncelikle **yeni bir liste** oluşturalım:

```
>>> liste1 = ["kapı", "pencere", "ev", "bahçe"]
```

Bunun yanında **ikinci bir liste daha** oluşturalım:

```
>>> liste2 = ["masa", "sandalye", "koltuk", "televizyon"]
```

Amacımız bu iki listeyi birbiriyle **birleştirmek**. Bunun için **`for`** döngülerinden ve **`append()`** metodundan yararlanabileceğimizi biliyorsunuz:

```
>>> for öge in liste1:
...     liste2.append(öge)
...
>>> liste2
['masa', 'sandalye', 'koltuk', 'televizyon', 'kapı', 'pencere',
'ev', 'bahçe']
```

Burada **`liste1`** adlı listenin tüm ögeleri üzerinden tek tek geçip bu listenin ögelerini **`append()`** metodu yardımıyla **`liste2`** adlı listeye iliştirdik.

Ancak bu işlemi yapmanın daha **pratik** bir yöntemi var. Python'da iki listeyi birbiriyle birleştirmek için **`extend()`** adlı bir metottan da yararlanabiliriz:

```
>>> liste1 = ["kapı", "pencere", "ev", "bahçe"]
>>> liste2 = ["masa", "sandalye", "koltuk", "televizyon"]
>>> liste1.extend(liste2)
```

```
>>> liste1
['kapı', 'pencere', 'ev', 'bahçe', 'masa', 'sandalye', 'koltuk',
'televizyon']
```

Gördüğünüz gibi **extend()** metodu iki listenin ögelerini **tek bir liste içinde birleştirmeye** yarıyor. Ya da başka bir ifadeyle, bir listeyi **genişletiyor** ve **uzatıyor**.

append() metodunu anlatırken de sözünü ettiğimiz gibi eğer arzu ederseniz **iki listeyi birleştirmek için +** işlecinden de yararlanabilirsiniz:

```
>>> liste1 = ["kapı", "pencere", "ev", "bahçe"]
>>> liste2 = ["masa", "sandalye", "koltuk", "televizyon"]
>>> liste1 + liste2
```

Elbette bu işlemin **kalıcı olabilmesi için** yaptığımız değişikliği bir değişken içinde **sabitlememiz** gerekiyor:

```
>>> liste3 = liste1 + liste2
```

Yine burada da hangi yöntemi seçeceğiniz konusunda tamamen özgürsünüz.

INSERT()

İngilizcede '**insert**' kelimesi '**yerleştirmek**', '**sokmak**' gibi anlamlara geliyor. İşte bu **insert()** metodu yardımıyla listenin herhangi bir noktasına öge yerleştirebiliyor veya sokabiliyoruz.

Elimizde şöyle bir liste olsun:

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

insert() metodu yardımıyla listenin ilk sırasına "**Ayşe**" ögesini yerleştirebiliriz:

```
>>> liste.insert(0, "Ayşe")
```

Gördüğünüz gibi **insert()** metodu öteki metotların aksine tek değil **iki argüman alıyor**. Burada parantez içindeki ilk argüman, "**Ayşe**" ögesinin liste içinde **yerleştirileceği sırayı gösteriyor**.

Listemizin **son durumunu** kontrol edelim:

```
>>> liste
```

Bu komutun **çıktısı** şöyle olur:

```
["Ayşe", "Hale", "Ahmet", "Jale", "Lale", "Mehmet"]
```

Gördüğünüz gibi tıpkı daha önce öğrendiğimiz dilimleme tekniğinde olduğu gibi Python burada da saymaya **sıfırdan** başlıyor. Yani tıpkı bir karakter dizisinin ilk ögesinin sıra numarası **0** olduğu gibi bir listenin de ilk ögesinin sıra numarası **0**'dır.

Bu metotun da tıpkı **`append()`** metodunda olduğu gibi listeye **yalnızca bir adet** öge eklememize izin verdiğini gözden kaçırmıyoruz.

REMOVE()

`append()`, **`extend()`** ve **`insert()`** metotları yardımıyla l**isteye ögeler eklemeyi** öğrendik. *Peki, ya listemizden öge çıkarmak istersek ne yapacağız?*

Python'da bu işi yapmamızı sağlayan **iki tane metot** var. Biz önce bunlardan ilki olan **`remove()`** metoduna bakacağız. Bu kelime Türkçede '**çıkarmak**', '**kaldırmak**', '**silmek**' gibi anlamlara geliyor.

Diyelim ki yukarıda tanımladığımız **`kripto`** adlı listeden "**`ethereum`**" ögesini **çıkarmak**/**kaldırmak** istiyoruz. O zaman şu komutu vermemiz gerekir:

```
>>> kripto.remove("ethereum")
```

Eğer listede "**`ethereum`**" adlı birden fazla öge varsa Python **listede bulduğu ilk ögeyi çıkaracaktır**.

POP()

Eğer listeden çıkaracağınız bir ögenin **ekrana yazılmasını da istiyorsanız** **`pop()`** metodunu kullanabilirsiniz.

Şu komutu deneyelim:

```
>>> kripto.pop()
```

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

```
>>> kripto.pop(0)
```

Bu komut ise listedeki ilk, yani 0. ögeyi çıkarır ve çıkardığı ögeyi ekrana yazdırır.

Anladığınız gibi **`pop()`** ile **`remove()`** arasındaki en temel fark; **`pop()`** metodunun **silinen ögeyi ekrana yazdırması**, **`remove()`** metodunun ise **yazdırmamasıdır**. Ayrıca **`pop()`** metodunda isim belirterek **listeden silme işlemi yapamazsınız**. Mutlaka silinecek ögenin liste içindeki **sırasını** vermelisiniz. **`remove()`** metodunda da bu durumun tam tersi söz konusudur. Yani **`remove()`** metodunda da sıra belirtemezsiniz; **isim vermeniz gerekir**.

Şimdiye kadar:

- bir listenin en sonuna nasıl öge ekleyeceğimizi (**`append()`**),
- listeyi başka bir listeyle nasıl genişletebileceğimizi (**`extend()`**)
- listenin herhangi bir yerine nasıl öge ekleyeceğimizi (**`insert()`**),
- listeden isim vererek nasıl öge çıkaracağımızı (**`remove()`**),
- listeden sayı vererek nasıl öge çıkaracağımızı (**`pop()`**)

öğrendik.

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

INDEX()

Diyelim ki elimizde şöyle bir liste var:

```
liste = [“Simovic”, “Prekazi”, “Jardel”, “Nouma”]
```

Listedeki “**Jardel**” ögesinin listenin kaçıncı sırasında olduğunu merak ediyorsunuz. İşte bu **index()** metodu, sizin aradığınız şey! Bunu şöyle kullanıyoruz:

```
>>> liste.index(“Jardel”)
```

Bu komut, “**Jardel**” ögesinin liste içinde **kaçıncı sırada olduğunu** gösterecektir bize.

Burada da liste ögelerini saymaya 0'dan başladığımızı gözden kaçırmıyoruz...

SORT()

Bazen listemizdeki ögeleri **alfabe sırasına dizmemiz** gerekebilir. İşte böyle bir durumda kullanacağımız metodun adı **sort()**.

Basit bir **örnek** verelim bu metot ile ilgili.

Diyelim ki elimizde şöyle bir liste var:

```
>>> üyeler = [‘Ahmet’, ‘Mehmet’, ‘Ceylan’, ‘Seyhan’, ‘Mahmut’,
‘Zeynep’,
... ‘Abdullah’, ‘Kadir’, ‘Kemal’, ‘Kamil’, ‘Selin’, ‘Senem’,
... ‘Sinem’, ‘Tayfun’, ‘Tuna’, ‘Tolga’]
```

Bu listedeki isimleri mesela **alfabe sırasına dizmek için sort()** metodunu kullanabiliriz:

```
>>> üyeler.sort()
>>> print(üyeler)
[‘Abdullah’, ‘Ahmet’, ‘Ceylan’, ‘Kadir’, ‘Kamil’, ‘Kemal’, ‘Mahmut’,
 ‘Mehmet’, ‘Selin’, ‘Senem’, ‘Seyhan’, ‘Sinem’, ‘Tayfun’, ‘Tolga’,
 ‘Tuna’, ‘Zeynep’]
```

Bu metot, elbette yalnızca harfleri alfabe sırasına dizmek için değil **sayıları sıralamak için de kullanılabilir**:

```
>>> sayılar = [1, 0, -1, 4, 10, 3, 6]
>>> sayılar.sort()
>>> print(sayılar)
[-1, 0, 1, 3, 4, 6, 10]
```

Gördüğünüz gibi **`sort()`** metodu, ögeleri **artan sıralamaya** tabi tutuyor. Yani ögeler '**a**, **b**, **c**' veya 1, 2, 3 şeklinde sıralanıyor. Bunun tersini yapmak da mümkündür. Yani istersek Python'ın sıralama işlemini 'c, b, a' şeklinde yapmasını da sağlayabiliriz. Bunun için **`sort()`** metodunun **`reverse`** adlı isimli argümanını kullanacağız.

Dikkatlice bakın:

```
>>> üyeler = ['Ahmet', 'Mehmet', 'Ceylan', 'Seyhan', 'Mahmut',
'Zeynep',
... 'Abdullah', 'Kadir', 'Kemal', 'Kamil', 'Selin', 'Senem',
... 'Sinem', 'Tayfun', 'Tuna', 'Tolga']
>>> üyeler.sort(reverse=True)
```

Gördüğünüz gibi **`sort()`** metodunun **`reverse`** adlı isimli argümanına verdiğimiz **`True`** değeri sayesinde liste ögelerini **ters sıraladık**. Bu parametrenin öntanımlı değeri **`False`**'tur. Yani **`sort()`** metodu öntanımlı olarak ögeleri **artıra artıra** sıralar. Ögeleri azalta azalta sıralamak için **`reverse`** parametresinin **`False`** olan öntanımlı değerini **`True`** yapmamız yeterli olacaktır.

Gelin isterseniz **`sort()`** metodunu kullanarak bir **örnek** daha verelim. Elimizde şöyle bir liste olsun:

```
>>> isimler = ["ahmet", "ışık", "ismail", "çiğdem", "can", "şule"]
```

Bu listedeki isimleri **alfabe sırasına dizelim**:

```
>>> isimler.sort()
>>> isimler
['ahmet', 'can', 'ismail', 'çiğdem', 'ışık', 'şule']
```

Gördüğünüz gibi **çıktı** pek beklediğimiz gibi değil. Çünkü listelerin **`sort()`** metodu **Türkçe karakterleri düzgün sıralayamaz**. Elbette bu metodun, Türkçe karakterleri **düzgün bir şekilde sıralamasını** sağlamanın çeşitli yolları var. Biz ilerleyen derslerimizde bu işin nasıl yapılacağını inceleyeceğiz. Ama şimdilik Türkçe karakterlerin nasıl düzgün sıralanacağını bir kenara bırakıp listelerin başka bir metodunu anlatmaya geçelim.

REVERSE()

Bu metot, listedeki **ögeleri ters yüz eder**. Şöyle ki:

```
>>> li = [“a”, “b”, “c”, “ç”, “d”]
>>> li.reverse()
>>> li
[‘d’, ‘ç’, ‘c’, ‘b’, ‘a’]
```

Burada herhangi bir sıralama işlemi yapılmadığına dikkat edin. **`reverse()`** metodu listedeki her bir ögeyi **sağdan sola dizmekle** yetinecektir:

```
>>> li = [1, 2, “a”, “c”, “ç”, “İ”, 0]
>>> li.reverse()
>>> li
[0, ‘İ’, ‘ç’, ‘c’, ‘a’, 2, 1]
```

Yani burada **`sort()`** benzeri bir işlem yapılmıyor.

COUNT()

Listelerle birlikte kullanabileceğimiz başka bir metot **`count()`** metodudur. Görevi ise liste içinde bir ögenin, **kaç kez geçtiğini** söylemektir:

```
>>> liste = [“a”, “b”, “s”, “a”]
>>> liste.count(“a”)
2
```

Böylece listelerin metotlarını incelemiş olduk. Şimdi, listelerin başka bir özelliğini incelemeye geçebiliriz.

Listelerin Dilimlenmesi

Dilimleme kavramını, karakter dizilerinden hatırlıyorsunuz. Bir karakter dizisinin **herhangi bir ögesine erişebilmek için** o karakter dizisini şu şekilde dilimlediğimizi biliyorsunuz:

```
>>> kardiz = "istihza"
>>> kardiz[0]
'i'
```

Aynı yöntemi kullanarak **listeleri de rahatlıkla dilimleyebiliriz**:

```
>>> liste = ["ahmet", "mehmet", "veli", "ayşe", "zeynep"]
>>> liste[0]
'ahmet'
```

Bu komut listenin ilk, yani **0.** ögesini ekrana yazdıracaktır.

Dilimleme yöntemini kullanarak bir listenin **herhangi bir ögesine erişebilirsiniz**. Yapmanız gereken tek şey, o ögenin liste içindeki sırasını belirtmek. Tabii bunu yaparken saymaya **sıfırdan başlamayı** unutmayacaksınız:

```
>>> liste[1]
'mehmet'

>>> liste[2]
'veli'
```

Eğer bir listenin son ögesini almak istiyorsanız birkaç seçeneğiniz var. Üzerinde çalıştığınız listenin kaç ögeli olduğunu biliyorsanız doğrudan **son ögenin sırasını** belirtebilirsiniz. Mesela listeniz 5 ögeli ise belirtmeniz gereken sayı **4** olacaktır. Çünkü bildiğiniz gibi **Python saymaya 0'dan başlıyor:**

```
>>> liste[4]
'zeynep'
```

Eğer listenin uzunluğunu bilmiyorsanız **len()** fonksiyonundan yararlanabilirsiniz. Zira bu komut size ilgili **listenin uzunluğunu** verecektir.

Dolayısıyla bu fonksiyonun verdiği çıktıdan **1** sayı çıkararak **son ögeyi** bulabilirsiniz:

```
>>> liste[len(liste)-1]
'zeynep'
```

Ancak bir listenin, **en son ögesine ulaşmanın en kolay yolu** şudur:

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

Sondan bir önceki ögeye ise şöyle ulaşıyoruz:

```
>>> liste[-2]
'ayşe'
```

Bunlar, dilimleme tekniğini kullanarak yapabileceğimiz **temel işlemler**. Ama bu teknik bize **daha karmaşık işlemler** yapma imkânı da sağlar.

Mesela dilimleme tekniğini kullanarak 100 ögeli bir listenin ögelerini, **ikişer ikişer** atlayarak yazdırabiliriz.

Öncelikle **100 ögeli** bir liste oluşturalım:

```
>>> liste = range(100)
```

Bu liste 0'dan 100'e kadar olan sayıları içeren bir **aralık nesnesi** oluşturuyor.

Şimdi, bu listenin ögeleri üzerinde **ikişer ikişer atlayarak** yeni bir liste oluşturalım:

```
>>> yeni = liste[0:100:2]
```

Bu listenin ögelerini nasıl görüntüleyeceğinizi biliyorsunuz:

```
>>> print(*yeni)
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46
48 50 52
54 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94 96 98
```

Elbette o anda elinizde olan bir listenin öge sayısını bilmiyor olabilirsiniz. **Bu durumda yukarıdaki kodu şöyle de yazabilirsiniz**:

```
>>> yeni = liste[0:len(liste):2]
```

Yukarıda da belirttiğimiz gibi **`len(liste)`** komutu zaten bir **listenin uzunluğunu** veriyor. Dolayısıyla listenin uzunluğunu kendimiz el yordamıyla bulup yazmak yerine bu işi **Python**'a bırakabiliriz.

Hatta yukarıdaki işlemi çok daha kolay bir şekilde şöyle de yapabiliriz:

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

Karakter dizilerinden de hatırladığınız gibi dilimlediğimiz bir veri tipinin **ilk** ve **en son ögesini dilimleme işlemi sırasında belirtmek zorunda değiliz**.

Peki, acaba bir listenin ögelerini nasıl ters yüz edebiliriz? Esasında bu da çok kolay. **Dikkatlice bakın**:

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

Bu şekilde bir listenin ögelerini sondan başa doğru, **birer birer sıralamış oluyoruz**. Yani aslında **`reverse()`** metodunun yaptığı işi yapıyoruz:

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

Dilimleme yöntemleriyle yapabileceklerimiz bunlarla da sınırlı değildir. Mesela eğer tanımladığımız bir listedeki ögelerin tamamını değil de yalnızca 2. ve 3. ögeleri görmek istiyorsanız, şu komuttan faydalanabilirsiniz:

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

Yukarıdaki komutta **birinci sayı dâhil**, **ikinci sayı hariç** olacak şekilde bu ikisi arasındaki ögeler listelenecektir. Yani **`liste[2:4]`** komutu listedeki 2. ve 3. ögeleri yazdıracaktır.

Dilimleme yöntemleriyle listeye yeni *öge yerleştirmek, listeden öge silmek vb.* de mümkündür.

Önce **temiz bir liste** oluşturalım:

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

Bu listenin herhangi bir yerindeki **bir ögeyi silip onun yerine başka bir öge yerleştirmek için** şu komutu kullanabiliriz:

```
>>> liste[0] = “çilek”
```

Bu komut listenin en başındaki ögeyi silip onun yerine “**`çilek`**” ögesini getirecektir.

Eğer en baştaki ögeyi silmeden bu ögeyi bir **sağa kaydırmak istersek** yukarıdaki komutu şöyle yazabiliriz:

```
>>> liste[:0] = [“çilek”]
```

Ancak bunun yerine **`insert()`** metodunu kullanmak çok daha kolaydır:

```
>>> liste.insert(0, “çilek”)
```

Bir listenin en sonuna **bir veya birden fazla öge eklemek için** ise dilimleme yönteminden şu şekilde yararlanabiliriz:

```
>>> liste[len(liste):] = [“domates”, “salata”]
```

Ancak bu yöntem yerine **`extend()`** metodunu veya **+** işlecini kullanmak çok daha akıllıca olacaktır:

```
>>> liste.extend([“domates”, “salata”])
```

veya:

```
>>> liste += [“domates”, “salata”]
```

Bu listenin mesela 3. sırasına **bir veya birden fazla öge yerleştirmek için** şu komutu kullanabiliriz:

```
>>> liste[3:3] = [“kebap”, “lahmacun”]
```

Bu listenin **2. sırasındaki ögeyi silmek için** ise şu komutu kullanabiliriz:

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

Bir listenin **herhangi bir yerindeki bir ögeyi silmenin daha mantıklı bir yolu** ise **`del`** adlı bir deyimden yararlanmaktır:

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

Bir listenin **2. sırasındaki ögeyi silip yerine bir veya birden fazla öge eklemek için** şu komutu kullanabiliriz:

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

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

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

Böylece hem **metotlar** hem de **dilimleme** yöntemi yardımıyla, Python'daki `liste` veri tipini nasıl yöneteceğimizi öğrenmiş olduk.

Bu arada, yukarıda anlattıklarımızın hepsi bir anda zihninize yerleşmeyebilir. Bu gayet normal. Burada anlattıklarımızı kendi yazdığınız programlarda uyguladıkça sindirecek ve belleğinize kazıyacaksınız. Ayrıca hatırlayamadığınız konuları ihtiyacınız olduğunda **referans kaynaklarından** da her an kontrol etme imkânına sahipsiniz.

İç İçe Geçmiş Listeler

Bu noktaya gelinceye kadar öğrendiklerimiz sayesinde artık bir listenin **nasıl tanımlanacağını** biliyoruz. Mesela:

```
>>> diller = ["ingilizce", "almanca", "fransızca"]
```

Bu listenin **tamamına** nasıl ulaşacağımızı da biliyoruz:

```
>>> diller
['ingilizce', 'almanca', 'fransızca']
```

Hatta dilimleme tekniği sayesinde, bu listenin ögelerine **tek tek** de erişebiliyoruz:

```
>>> diller[0]
'ingilizce'

>>> diller[-1]
'fransızca'
```

Peki, bir liste içinde başka listelerin olduğu, yani iç içe listelerin söz konusu olduğu bir durumda, bu tür listelerin ögelerine nasıl erişeceğiz?

Mesela yukarıda tanımladığımız **matris** listesini ele alalım:

```
>>> matris = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Burada **iç içe geçmiş üç tane liste** görüyoruz.

Acaba bu matrisin içindeki ilk listenin ilk elemanına nasıl ulaşırız?

Dikkatlice bakın:

```
>>> matris[0][0]
1
```

Burada, **sıfırıncı listenin sıfırıncı ögesini** almış olduk.

Şu da ikinci listenin **son elemanı**:

```
>>> matris[1][-1]
6
```

Burada kendi kendinize bazı denemeler yaparak bu sisteme kolayca aşinalık kazanabilirsiniz.

LIST() FONKSİYONU

Listeler konusunu kapatmadan önce listelere ilişkin önemli bir fonksiyondan da söz etmeden geçmeyelim. Bu önemli fonksiyonun adı **list()**.

Bu fonksiyon, tıpkı **int()**, **str()** ve **float()** fonksiyonları gibi veri tipleri arasında **dönüştürme işlemi** yapmamızı sağlar. **list()** fonksiyonunu kullanarak **liste olmayan bir veri tipini listeye** dönüştürebiliriz.

Örneğin bu fonksiyonu kullanarak **range()** fonksiyonunun ürettiği **sayı aralığını** bir **liste** hâlinde alabiliriz:

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

Gördüğünüz gibi bu komut **0** ile **10** arasında bir **sayı aralığı üretti**. İşte bu aralıktaki sayılara, **list()** fonksiyonuyla ulaşabiliriz:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

list() fonksiyonu karakter dizilerini de bir **listeye** dönüştürebilir:

```
>>> kardiz = "istihza"
>>> list(kardiz)
['i', 's', 't', 'i', 'h', 'z', 'a']
```

Liste veri tipi konusunda epey söz söyledik. Artık listeler konusunu burada noktalayıp **demetler** (*tuples*) konusuna geçebiliriz...

17 DEMETLER

Bu bölümde inceleyeceğimiz yeni veri tipinin adı '**demet**'. Aslında siz bu veri tipine de tamamen yabancı değilsiniz. Geçen bölümlerde, sadece isim olarak olsa bile demetlerden söz etmiştik. Örneğin **platform** modülünü incelediğimiz bölümde, bu modül içindeki **python_version_tuple()** adlı fonksiyonun bize bir **demet verdiğini** öğrenmiştik:

```
>>> import platform
>>> platform.python_version_tuple()
('3', '8', '3')
```

Burada Python'ın *büyük*, *küçük* ve *minik* sürüm numaralarını **parantez içinde** birbirinden **virgül** ile ayrılmış ögeler hâlinde görüyoruz. İşte birbirinden virgülle ayrılmış, parantez içinde gösterilen bu ögelere '**demet**' (*tuple*) adı veriyoruz.

Demetler, **listelere benzeyen** bir veri tipidir. Ama listeler ile aralarında çok temel bir fark bulunur. Listeler üzerinde **oynamalar yapabiliriz**. Yani öge *ekleyebilir*, öge *çıkarabiliriz*. **Demetlerde ise böyle bir şey yoktur**. Birazdan bu konuya daha ayrıntılı bir şekilde değineceğiz. Ama isterseniz şimdi, kendi demetlerimizi nasıl tanımlayacağımızı görelim.

Demetleri tanımlamanın **iki yolu** bulunur. Birinci yöntemde, demet tanımlamak için **parantez işaretlerini** kullanıyoruz:

```
>>> demet1 = (“Ali”, “Veli”, 49, 50)
```

İkinci yöntemde ise **parantezleri hiç kullanmaksızın**, ögeleri **yalnızca birbirinden virgül** ile ayırıyoruz:

```
>>> demet2 = “Ali”, “Veli”, 49, 50
```

Gördüğünüz gibi ikinci yöntem *değişken tanımlamaya çok benziyor*.

Parantezli de olsa parantezsiz de olsa yukarıda tanımladıklarımızın ikisi de **demet sınıfına girer**. İsterseniz bu durumu teyit edelim:

```
>>> type(demet1)
<class ‘tuple’>

>>> type(demet2)
<class ‘tuple’>
```

Peki, boş bir demet nasıl oluşturulur? **Çok basit:**

```
>>> demet = ()
```

Peki, tek ögeli bir demet nasıl oluşturulur? **O kadar basit değil**. Aslında basit, ama biraz tuhaf:

```
>>> demet = (“su”,)
```

Gördüğünüz gibi tek ögeli bir demet oluşturabilmek için ögenin yanına bir **virgül koyuyoruz**! Hemen teyit edelim:

```
>>> type(demet)
<class ‘tuple’>
```

O virgülü koymazsak ne olur?

```
>>> demet2 = (“su”)
```

demet2'nin tipini kontrol edelim:

```
>>> type(demet2)
<class ‘str’>
```

Demek ki virgülü koymazsak **demet** değil, alelade bir karakter dizisi oluşturmuş oluyoruz.

"*Peki, listeler varken bu demetler ne işe yarar*?" diye sorduğunuzu duyar gibiyim.

Bir defa, demetler listelerin aksine **değişiklik yapmaya müsait olmadıklarından** listelere göre daha **güvenlidir**. Yani yanlışlıkla, değiştirmek istemediğiniz veriler içeren bir liste hazırlamak istiyorsanız **demetleri kullanabilirsiniz**.

Ayrıca demetler, listelere göre **daha hızlı** çalışır. Dolayısıyla bir program içinde sonradan değiştirmeniz gerekmeyecek verileri **gruplamak** için **liste yerine demet kullanmak daha mantıklı olabilir**.

Dediğimiz gibi **demetler listelere çok benzer**. Dolayısıyla listelerle yaptığınız pek çok şeyi demetlerle de yapabilirsiniz. Örneğin **len()** fonksiyonunu kullanarak demetlerin uzunluğunu sorgulayabilirsiniz:

```
>>> d = (‘elma’, ‘armut’, ‘çilek’)
>>> len(d)
3
```

Aynı şekilde tıpkı karakter dizileri ve listelerde olduğu gibi **dilimleme yöntemlerini** demetler üzerinde de uygulayabilirsiniz:

```
>>> d[0]
>>> d[1]
>>> d[-1]
>>> d[:2]
```

Hatta tıpkı iç içe listeler tanımladığınız gibi **iç içe demetler** de tanımlayabilirsiniz:

```
>>> d = (1, 2, 3, (4, 5, 6), 7, 8, 9)
```

Listelerde olduğu gibi farlı **veri tiplerinden oluşan demetler de** oluşturabilirsiniz:

```
>>> d = ('a', 1, [0, 1, 2], (3, 4, 5), 3.2, -4)
```

...ve bu ögelere de **tek tek** erişebilirsiniz:

```
>>> d[0]
'a'

>>> d[2]
[0, 1, 2]
```

Gördüğünüz gibi bu demetin 2. ögesi bir **liste**. Bu listenin ögelerine de tek tek nasıl ulaşabileceğinizi biliyorsunuz:

```
>>> d[2][0]
0
>>> d[2][1]
1
>>> d[2][2]
2
```

Demetlerin Metotları

Tıpkı listelerde olduğu gibi demetlerin de **kendine ait birtakım metotları bulunur**. Bu metotların neler olduğunu görüntülemek için daha önce öğrendiğimiz yöntemleri kullanabilirsiniz.

Python'da demet veri tipini '**tuple**' kelimesi ile gösteriyoruz. Dolayısıyla demetlerin metotlarını şöyle listeyebiliriz:

```
>>> dir(tuple)
```

Bu iş için demetlerin ayırt edici işareti olan **parantezleri** de kullanabilirsiniz:

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

Hatta **metotları görüntülemek için** herhangi bir demetten de yararlanabilirsiniz:

```
>>> aile = "Anne", "Baba", "Kardeşler"
>>> dir(aile)
```

Yukarıdaki komutların herhangi birini verdiğimizde şu **çıktıyı** elde ediyoruz:

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__
mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__
rmul__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'count',
'index']
```

Burada bizi ilgilendiren metotlar şunlar:

```
>>> for i in dir(tuple):
...     if "__" not in i:
...         print(i)
...
count
index
```

Gördüğünüz gibi demetlerin yalnızca **iki tane metodu** var. Bu iki metot, tıpkı listelerin **`count()`** ve **`index()`** metotları gibi çalışır.

Demetler **değişiklik kabul etmeyen** bir veri tipi olduğu için bu veri tipi doğal olarak **değişiklik yapmaya yönelik herhangi bir metot da içermiyor**.

COUNT()

Dediğimiz gibi listelerin **`count()`** metodu nasıl çalışıyorsa demetlerin **`count()`** metodu da öyle çalışır. Bu metodu, bir demet içindeki bir ögenin **o demet içinde kaç kez geçtiğini öğrenmek için** kullanıyoruz:

```
>>> d = (1, 2, 3, 2, 5, 3, 4, 5, 10)
>>> d.count(3)
2
```

INDEX()

`index()` metodunu da tıpkı listelerde olduğu gibi bir ögenin **demet içindeki sırasını** öğrenmek için kullanıyoruz:

```
>>> d.index(10)
8
```

tuple() Fonksiyonu

Daha önce öğrendiğimiz **str()**, **int()**, **float()** ve **list()** fonksiyonlarına benzer şekilde, **tuple()** fonksiyonu da **demet olmayan veri tiplerini demet veri tipine dönüştürmemize olanak tanır**. **Örneğin**:

```
>>> aralık = range(0, 100, 2)
```

Burada **0** ile **100** arasındaki sayıları **ikişer ikişer atlayarak** gösteren bir **aralık** nesnesi oluşturduk.

Bu aralık içinde yer alan sayıları görüntüleyebilmek için **list()** fonksiyonundan yararlanabiliyoruz:

```
>>> l = list(aralık)
>>> print(l)
```

Bu şekilde **aralık** nesnesini bir **listeye** dönüştürmüş olduk. Aynı şekilde bu **aralık** nesnesini bir **demete** de dönüştürebiliriz:

```
>>> d = tuple(aralık)
>>> print(d)
(0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34,
36, 38,
40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72,
74, 76,
78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98)
```

Böylece **demet** veri tipini de anlatmış olduk. Gördüğünüz gibi bu veri tipi **listelere çok benziyor**. Bu ikisi arasındaki tek fark, listeler üzerinde değişiklik yapılabilirken **demetler üzerinde değişiklik yapılamamasıdır**.

18 SÖZLÜKLER

Bu bölümde **sözlük** adlı **yeni bir veri tipinden** söz edeceğiz. Sözlüğün ne demek olduğunu tanımlamadan önce, gelin isterseniz işe bir **örnekle** başlayalım:

```
>>> sözlük = {"elma": "meyve", "domates": "sebze", 1: "sayı"}
```

Burada teknik olarak "**elma**" ögesi bir "**anahtar**", "**meyve**" ise bu anahtarın "**değeri**" olarak adlandırılır. Aynı şekilde "**sebze**" değerinin anahtarı "**domates**"tir. Dolayısıyla Python'da *sözlük*, *anahtar* ve *değer* çiftlerinden oluşan ve bunlar arasında **bağ kuran** bir veri tipidir.

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

```
>>> telefon_defteri = {"Ahmet": "0533 123 45 67",
... "Salih": "0532 321 54 76",
... "Selin": "0533 333 33 33"}
```

Gördüğünüz gibi sözlükleri tanımlamak da son derece **kolay** bir iş. Burada sözlükleri tanımlamak için **süslü parantezlerden** yararlandığımıza ve anahtar ile değeri birbirinden ayırmak için **iki nokta üst üste işaretini** kullandığımıza dikkat ediyoruz. Ayrıca anahtar-değer çiftlerini de birbirinden **virgül** ile ayırdığımızı da gözden kaçırmıyoruz.

Şimdi, gelelim sözlüklerle neler yapabileceğimize...

Şu komuta bir bakalım:

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

veya:

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

Bu komutlar "**`Ahmet`**" ve "**`Salih`**" adlı anahtarların karşısında hangi değer varsa onu ekrana yazdıracaktır. Dikkat edin; sözlükten öge çağırırken küme parantezlerini değil, **köşeli parantezleri** kullanıyoruz. Bu arada aklınızda bulunsun, sözlük içindeki ögeleri **anahtara göre çağırıyoruz**, değere göre değil. Yani ögeleri çağırırken iki nokta üst üste işaretinin **solundaki ifadeleri** kullanıyoruz, 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["Salih"] = "0555 555 55 55"
```

Buradan anladığımız şu: Bir sözlüğe yeni bir öge eklerken de var olan 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** daha önce öğrendiğimiz **`del`** deyiminden yararlanıyoruz:

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

Bu tür işlemleri yapabilmek için sözlüklerin **metotlarından** da yararlanabiliriz. *Peki acaba sözlüklerin metotları nelerdir?*

Python'da **sözlük** veri tipi '**dict**' kelimesi ile gösterilir. Dolayısıyla **sözlük** veri tipinin metotlarını **sorgulamak** için şöyle bir komut kullanabiliriz:

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

Bu iş için sözlüklerin **ayırt edici işareti olan süslü parantezlerden** de yararlanabilirsiniz:

```
>>> dir({})
```

Hatta sözlüklerin metotlarını sorgulamak için **herhangi bir sözlüğü** de kullanabilirsiniz:

```
>>> dir(telefon_defteri)
```

Bu komutların herhangi birini verdiğimizde şöyle bir **çıktı** elde ediyoruz:

```
['__class__', '__contains__', '__delattr__', '__delitem__', '__
dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__gt__', '__hash__', '__init__', '__init_
subclass__',
'__iter__', '__le__', '__len__', '__lt__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__
setattr__',
'__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'clear',
'copy',
'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault',
'update', 'values']
```

Burada **bizi ilgilendiren metotlar** şunlar:

```
>>> for metot in dir(dict):
...     if "__" not in metot:
...         print(metot)
...
clear
copy
```

```
fromkeys
get
items
keys
pop
popitem
setdefault
update
values
```

İsterseniz şimdi de bu metotların **en sık** kullanılanlarını inceleyelim.

Sözlüklerin Metotları

Yukarıdaki metotları incelemeye **keys()** metodundan başlayalım.

keys()

keys() metodunu kullanarak bir **sözlük içindeki tüm anahtarları** elde edebiliriz:

```
>>> telefon_defteri.keys()
['Ahmet', 'Salih', 'Selin']
```

values()

values() metodu ise bize bir **sözlükteki değerleri** verir:

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

ITEMS()

Bu metot ise bize aynı anda hem **anahtarları** hem de **değerleri** verir. Mesela elimizde şöyle bir sözlük olsun:

```
sözlük = {'a': 0, 'b': 1, 'c': 2, 'ç': 3, 'd': 4,
          'e': 5, 'f': 6, 'g': 7, 'ğ': 8, 'h': 9,
          'ı': 10, 'i': 11, 'j': 12, 'k': 13,
          'l': 14, 'm': 15, 'n': 16, 'o': 17,
          'ö': 18, 'p': 19, 'r': 20, 's': 21,
          'ş': 22, 't': 23, 'u': 24, 'ü': 25,
          'v': 26, 'y': 27, 'z': 28}
```

Bu sözlüğün anahtarlarına şu şekilde ulaşabileceğimizi biliyoruz:

```
print(sözlük.keys())
```

Aynı şekilde bu **sözlüğün değerlerine** ise şöyle ulaşıyoruz:

```
print(sözlük.values())
```

Eğer bu sözlüğün aynı anda hem anahtarlarına hem de değerlerine ulaşmak isterseniz **`items()`** metodunu kullanacaksınız. Dikkatlice bakın:

```
print(sözlük.items())
```

Bu komut bize şöyle bir **çıktı** verir:

```
dict_items([('a', 0), ('b', 1), ('c', 2), ('ç', 3), ('d', 4), ('e',
5),
('f', 6), ('g', 7), ('ğ', 8), ('h', 9), ('ı', 10), ('i', 11), ('j',
12),
('k', 13), ('l', 14), ('m', 15), ('n', 16), ('o', 17), ('ö', 18),
('p', 19),
('r', 20), ('s', 21), ('ş', 22), ('t', 23), ('u', 24), ('ü', 25),
('v', 26),
('y', 27), ('z', 28)])
```

Bu çıktıya dikkatlice bakarsanız bunların aslında bir liste içinde yer alan **iki ögeli demetler** olduğunu göreceksiniz.

Eğer **`sözlük.items()`** üzerinde bir **`for`** döngüsü kurarsanız durumu daha net görebilirsiniz:

```
for i in sözlük.items():
    print(i)
```

Bu kodlar bize şöyle bir **çıktı** verir (*aşağıdaki çıktının sadece bir kısmı gösterilmiştir*):

```
('a', 0)
('b', 1)
('c', 2)
('ç', 3)
('d', 4)
('e', 5)
('f', 6)
('g', 7)
('ğ', 8)
('h', 9)
```

Mesela **`sözlük`** adlı sözlüğün **anahtar** ve **değerlerini** kullanarak şöyle bir **çıktı** vermek istiyorsanız:

```
a = 0
b = 1
c = 2
ç = 3
d = 4
e = 5
f = 6
g = 7
ğ = 8
h = 9
```

Şu kodları yazabilirsiniz:

```
for k, v in sözlük.items():
    print(f"{k} = {v}")
```

Burada iki ögeli demetlerin ilk ögesini **k**, ikinci ögesini ise **v** adlı bir değişkene atadık. Daha sonra **f**'li karakter dizileri içinde istediğimiz görüntüyü elde edip **çıktı** olarak verdik.

CLEAR()

Eğer biz sözlükteki **bütün ögeleri silmek istersek** `clear()` metodundan yararlanıyoruz:

```
>>> telefon_defteri.clear()
```

GET()

Dilerseniz bu metodun **ne işe yaradığını** bir **örnek** üzerinden gösterelim. Diyelim ki bir hava durumu programı yazmak istiyoruz. Tasarımıza göre kullanıcı bir *şehir adı* girecek. Program da girilen şehre özgü *hava durumu bilgilerini* ekrana yazdıracak. Bunu yapabilmek için daha önceki bilgilerimizi de kullanarak şöyle bir şey yazabiliriz:

```
soru = input("Şehrinizin adını tamamı küçük harf olacak şekilde
yazınız: ")

if soru == "istanbul":
    print("gök gürültülü ve sağanak yağışlı")

elif soru == "ankara":
    print("açık ve güneşli")

elif soru == "izmir":
    print("bulutlu")

else:
    print("Bu şehre ilişkin havadurumu \
    bilgisi bulunmamaktadır.")
```

Ama yukarıdaki yöntemin biraz meşakkatli olacağı açık. Sadece üç şehir için hava durumu bilgilerini sorgulayacak olsak mesele değil, ancak **onlarca şehri kapsayacak bir program** üretmekse amacımız, yukarıdaki yöntem yerine daha **pratik** bir yöntem uygulamak gayet yerinde bir tercih olacaktır. İşte bu noktada programcının imdadına Python'daki **`sözlük`** veri tipi yetişecektir.

Yukarıdaki kodların yerine getirdiği işlevi, şu kodlarla da gerçekleştirebiliriz:

```
soru = input("Şehrinizin adını tamamı küçük harf olacak şekilde
yazınız: ")

cevap = {"istanbul" : "gök gürültülü ve sağanak yağışlı",
         "ankara"   : "açık ve güneşli",
         "izmir"    : "bulutlu"}

print(cevap.get(soru, "Bu şehre ilişkin havadurumu \
bilgisi bulunmamaktadır."))
```

Gördüğünüz gibi ilk önce, **normal biçimde** kullanıcıya sorumuzu soruyoruz. Ardından da **anahtar-değer** çiftleri şeklinde şehir adlarını ve bunlara karşılık gelen hava durumu bilgilerini bir **sözlük içinde depoluyoruz**. Daha sonra, **`sözlük`** metotlarından biri olan **`get()`** metodunu seçiyoruz. Bu metot bize sözlük içinde bir değerin **var olup olmadığını** denetleme imkânının yanı sıra, adı geçen değerin **sözlük içinde var olmaması** durumunda kullanıcıya gösterilecek bir **mesaj** belirleme olanağı da sunar.

Python sözlüklerinde bulunan bu **`get()`** metodu bizi bir **`else`** veya sonraki derslerimizde işleyeceğimiz **`try-except`** bloğu kullanarak **hata yakalamaya uğraşma zahmetinden de kurtarır**.

Burada **`print(cevap.get(soru,"Bu şehre ilişkin hava durumu bilgisi bulunmamaktadır."))`** satırı yardımıyla, **soru** adlı değişkenin değerinin **sözlük içinde var olup olmadığını** sorguluyoruz. Eğer kullanıcının girdiği şehir adı, sözlüğümüz içinde **bir anahtar olarak tanımlanmışsa bu anahtarın değeri ekrana yazdırılacaktır**. Eğer kullanıcının girdiği şehir adı sözlüğümüz içinde bulunmuyorsa bu defa kullanıcıya "**Bu şehre ilişkin hava durumu bilgisi bulunmamaktadır.**" biçiminde bir mesaj gösterilecektir.

`if` deyimleri yerine sözlüklerden yararlanmanın, yukarıda bahsedilen faydaların dışında bir de şu yararları vardır:

1. Öncelikle sözü geçen senaryo için sözlükleri kullanmak programcıya daha az kodla **daha çok iş yapma** olanağı sağlar.
2. Sözlük programcının elle oluşturacağı `if-elif-else` bloklarından daha performanslıdır ve bize çok hızlı bir şekilde **veri sorgulama imkânı** sağlar.
3. Kodların daha az yer kaplaması sayesinde **programın bakımı da kolaylaşacaktır**.
4. Tek tek `if-elif-else` blokları içinde şehir adı ve buna ilişkin hava durumu bilgileri tanımlamaya kıyasla sözlük içinde **yeni anahtar-değer çiftleri oluşturmak daha pratiktir**.

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

KÜMELER

19

Bu bölümde '**küme**' adlı yeni bir veri tipinden bahsedeceğiz.

Kümeler Python programlama dilinin **temel veri tiplerinden biridir**. Her ne kadar bu veri tipi karakter dizileri veya listeler kadar sık bir şekilde karşınıza çıkmayacak olsa da kimi durumlarda *işlerinizi epey kolaylaştırabilir*.

Basit bir **örnek** verelim. Diyelim ki bir kelime içinde geçen harfleri **listelemek** istiyorsunuz. Ama bunu yaparken **her harften bir tane** olmasını amaçlıyorsunuz. Bunun için şu ana kadarki bilgilerinizi kullanarak **şöyle bir program yazabilirsiniz**:

```
kelime = input("kelime: ")

li = []

for harf in kelime:
    if harf not in li:
        li.append(harf)

print(li)
```

Ama bu işlemi kümeler ile çok daha **pratik** bir şekilde hâlledebilirsiniz. **Dikkatlice bakın:**

```
kelime = input("kelime: ")

li = set(kelime)

print(li)
```

Burada dikkatimizi çeken iki nokta var: Birincisi, **`input()`** fonksiyonundan gelen karakter dizisini küme veri tipine dönüştürmek için **`set()`** adlı bir fonksiyondan yararlandık.

İkincisi, karakter dizisini kümeye dönüştürdüğümüzde ögelerin **sırasının bozulduğunu** görüyoruz. **Yani:**

```
>>> s = "istihza"
>>> set(s)
{'h', 'z', 'a', 's', 'i', 't'}
```

Gördüğünüz gibi karakter dizisini **`set()`** fonksiyonu ile bir kümeye dönüştürdüğümüzde tekrar eden harfler **teke iniyor**, ancak karakter dizisinin içindeki ögelerin sırası bozuluyor. Bu durumun sebebi kümelerin '**sırasız**' bir veri tipi olmasıdır. Yani bir kümeye dönüştürdüğünüz ögeleri aynı sırayla **geri alacağınızın garantisi yok**tur.

Kümeler sırasız bir veri tipi olduğu için doğal olarak bu veri tipi için öge sıralaması diye bir şey de söz konusu değildir. Yani başka veri tiplerinde kullandığımız **dilimleme yöntemlerini** kümelerde kullanamıyoruz:

```
>>> s = set("istihza")
>>> s[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object is not subscriptable
```

Yukarıda gösterdiğimiz **set()** fonksiyonunu ayrıca hâlihazırda tanımlanmış olan bir liste içindeki **tekrar eden ögeleri teke indirmek için** de kullanabilirsiniz:

```
>>> l = [“ahmet”, “veli”, “ayşe”, “selin”,
... “mehmet”, “ahmet”, “ayşe”, “veli”, “ali”]
```

Amacımız **bu liste içinde her ögeden yalnızca bir tane olmasını sağlamak**. İşte bunun için **küme veri tipinden** yararlanabiliriz:

```
>>> küme = set(l)
>>> küme
{‘ahmet’, ‘mehmet’, ‘ayşe’, ‘veli’, ‘selin’, ‘ali’}
```

Eğer verilerinizi bir küme içinde değil de **yine bir liste içinde** almak isterseniz elbette **list()** fonksiyonundan yararlanabilirsiniz:

```
>>> list(küme)
[‘ahmet’, ‘ayşe’, ‘mehmet’, ‘veli’, ‘selin’, ‘ali’]
```

Burada önce **set()** fonksiyonuyla **l** kümesini bir kümeye dönüştürüp çift ögeleri **teke indirdikten sonra** bu kümeyi **list()** fonksiyonuyla tekrar listeye dönüştürdük.

Dediğimiz gibi bir küme tanımlamak için **set()** fonksiyonundan yararlanıyoruz. Mesela bu fonksiyonu kullanarak şu şekilde **boş bir küme** oluşturabiliriz:

```
>>> küme = set()
```

Gelin, isterseniz bu verinin tipini teyit edelim:

```
>>> print(type(küme))
<class ‘set’>
```

Karakter dizilerini nasıl ‘**str**’ kelimesi ile gösteriyorsak kümeleri de ‘**set**’ kelimesiyle gösterdiğimize dikkat edin.

Python'da bir küme oluşturmanın ikinci yöntemi, **süslü parantezlerden yararlanmaktır**. Mesela şu şekilde bir küme oluşturabiliriz:

```
>>> küme = {"a", "b", "c", "d", "e", "f"}
>>> küme
{'e', 'b', 'c', 'd', 'a', 'f'}
```

Ancak süslü parantezleri kullanarak **boş bir küme** oluşturamayız:

```
>>> küme = {}
```

Bu şekilde oluşturduğunuz şey küme değil, **sözlük** olacaktır:

```
>>> type(küme)
<class 'dict'>
```

Peki, kümelerin hangi metotlara sahip olduğunu nasıl kontrol edeceğiz? Bunun için yine '**set**' kelimesinden yararlanabiliriz:

```
>>> dir(set)
```

Bu listede bizi ilgilendiren metotlar şunlar:

```
'add', 'clear', 'copy', 'difference', 'difference_update',
'discard',
'intersection', 'intersection_update', 'isdisjoint', 'issubset',
'issuperset', 'pop', 'remove', 'symmetric_difference',
'symmetric_difference_update', 'union', 'update'
```

İşte bu bölümde, yukarıda gördüğümüz metotları inceleyeceğiz.

Kümelerin Metotları

Yukarıdaki listeden de gördüğünüz gibi kümelerin **epey metodu** var. Biz bu metotların hepsini tek tek inceleyeceğiz. Ama aslında *bu metotların tamamı her zaman karşınıza çıkmayacaktır*. Tüm bu metotlar arasında, özellikle **şu üç metodu aklınızda tutmaya çalışın**:

```
add()
remove()
update()
```

Python'da '**küme**' kavramının matematikten bildiğimiz **küme kavramıyla da sıkı bir bağlantısı vardır**. Dolayısıyla *kesişim*, *birleşim* ve *fark* gibi küme işlemleri gerçekleştireceksiniz şu metotları da bilmeniz gerekir:

```
union()
intersection()
difference()
```

Geri kalan metotları ise ihtiyaç duymanız hâlinde **referans kaynaklarına** bakarak ne işe yaradıklarını hatırlayabilirsiniz.

O hâlde lafı daha fazla uzatmadan kümelerin metotlarını incelemeye geçebiliriz.

add()

Python'daki küme veri tipinin en önemli metotlarından biri **`add`** metodudur. **Bu metodu bir kümeye öge eklemek için** kullanıyoruz:

```
>>> k = set()
>>> k.add("ahmet")
>>> k
{'ahmet'}
```

Bu metot, kümeye **her defasında tek bir öge** eklememize izin verir. Ama elbette **for** döngüsünden yararlanarak birden fazla ögeyi de kümeye ekleyebileceğinizi biliyorsunuz:

```
>>> k = set()
>>> l = [“a”, “b”, “c”, “d”]
>>> for öğe in l:
...     k.add(öğe)
```

REMOVE()

Eğer bir kümeden öge çıkarmak istiyorsanız **remove()** metodunu kullanabilirsiniz.

Öncelikle bir **küme** tanımlayalım:

```
>>> k = set([“ahmet”, “mehmet”, “veli”])
>>> k
{‘ahmet’, ‘mehmet’, ‘veli’}
```

Bu arada **k** kümesini tanımlamak için **set()** fonksiyonunu **nasıl kullandığımıza dikkat edin**:

```
>>> k = set([“ahmet”, “mehmet”, “veli”])
```

Burada “**ahmet**”, “**mehmet**” ve “**veli**” ögelerinden oluşan bir listeyi doğruca **set()** kümesine bir **argüman** olarak verip **k** adlı kümemizi tanımladık.

Dilerseniz liste yerine **demetlerden** de yararlanabilirsiniz:

```
k = set((“ahmet”, “mehmet”, “veli”))
```

Hatta bu iş için **sözlükleri** bile kullanabilirsiniz:

```
k = set({“ahmet”: “adana”, “mehmet”: “istanbul”, “veli”: “mersin”})
```

Bu durumda, sözlüğün anahtarları kümenin ögelerini oluşturacaktır. Ama eğer siz bir sözlüğün anahtarlarından değil de **değerlerinden** bir küme oluşturmak isterseniz şöyle bir kod da yazabilirsiniz:

```
>>> sözlük = {“ahmet”: “adana”, “mehmet”: “istanbul”, “veli”:
“mersin”}
>>> k = set(sözlük.values())
>>> k
{‘istanbul’, ‘mersin’, ‘adana’}
```

Eğer arzu ederseniz bir sözlüğün **anahtarlarından bir küme**, **değerlerinden de ayrı bir küme** oluşturabilirsiniz:

```
>>> k1, k2 = set(sözlük.keys()), set(sözlük.values())
>>> k1
{‘ahmet’, ‘mehmet’, ‘veli’}
>>> k2
{‘istanbul’, ‘mersin’, ‘adana’}
```

Neyse... Biz şimdi konuyu daha fazla dağıtmadan kaldığımız yerden kümelere devam edelim.

Yukarıda şöyle bir **küme** tanımlamıştık:

```
>>> k = set([“ahmet”, “mehmet”, “veli”])
```

Bu arada bu kümeyi doğrudan **süslü parantezleri** kullanarak da tanımlayabileceğinizi biliyorsunuz:

```
>>> k = {“ahmet”, “mehmet”, “veli”}
```

Şimdi, bu kümeden "**mehmet**" ögesini çıkaralım:

```
>>> k.remove("mehmet")
>>> k
{'ahmet', 'veli'}
```

Eğer çıkarmak istediğiniz öge kümede yoksa bu metot **hata** verecektir.

DISCARD()

discard() metodu da bir kümeden **öge çıkarmak için** kullanılır:

```
>>> k.remove("ahmet")
```

Eğer kümede böyle bir öge yoksa bu metot herhangi bir **hata vermez**. Zaten bu metodun **remove()** metodundan **tek farkı da budur**.

CLEAR()

Bir **kümenin içini tamamen boşaltmak için** **clear()** metodunu kullanıyoruz:

```
>>> k.clear()
>>> k
>>> set()
```

Böylece **k** adında **boş bir küme** elde etmiş olduk.

Eğer amacınız kümeyi **tamamen ortadan kaldırmaksa** **del()** fonksiyonunu kullanabilirsiniz:

```
>>> del k
```

Bu durumda **k** kümesi tüm ögeleriyle birlikte tamamen ortadan kalkacaktır:

```
>>> k

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

POP()

Bu metodu, **listelerden** hatırlayacaksınız. Aynı, listelerde olduğu gibi bir **küme içinden bir ögeyi silmek ve silinen ögeyi ekrana yazdırmak için** `pop()` metodunu kullanıyoruz:

```
>>> k.pop()
```

Ancak listelerin aksine kümeler **sırasız bir veri tipi** olduğu için bu komutun **hangi ögeyi sileceğini bilemezsiniz**.

Etkileşimli kabuk üzerinde çalışırken bu metot sanki ögeleri baştan sona doğru siliyormuş gibi görünebilir. Fakat gerçek programlarda bu silme işleminin **rastgele** olduğunu görebilirsiniz. Bunun için şu kodları, **bir program dosyasına yazıp çalıştırın**:

```
küme = {"ahmet", "mehmet", "ali", "veli", "hale", "jale", "lale"}

küme.pop()
küme.pop()
küme.pop()

print(küme)
```

Burada art arda üç kez **`pop()`** işlemi gerçekleştirdik. Bu üç işlem, **her defasında farklı üç ögeyi kümeden attı**. Dolayısıyla bu kodları her çalıştırdığınızda kümeden farklı ögelerin silindiğini göreceksiniz.

UPDATE()

`update()` metodu da tıpkı **`add()`** metodu gibi **bir kümeye öge eklemek için kullanılır**. **`update()`** metodunun **`add()`** metodundan **farkı**, **`update()`** metodu ile **bir kümeye birden fazla öge ekleyebiliyor olmamızdır**:

```
>>> küme1 = {"A", "B", "C"}
>>> küme2 = {"a", "b", "c"}
>>> küme1.update(küme2)
```

Burada **`küme1`** adlı kümeyi, **`küme2`** adlı kümenin ögeleri ile güncelledik. Yani **`küme2'`**nin ögelerini **`küme1`**'e ekledik.

`update()` metodunu kullanarak **liste**, **demet** ve **sözlük** gibi veri tipleri içindeki ögeleri bir kümeye ekleyebilirsiniz:

```
>>> l = [1, 2, 3]
>>> k = set()
>>> k.update(l)
```

Burada **`l`** adlı listenin ögelerini **`k`** adlı kümeye ekledik.

COPY()

Eğer bir kümeyi **kopyalamak** isterseniz **`copy()`** metodunu kullanabilirsiniz:

```
>>> k.copy()
```

Daha önce de söylediğimiz gibi **`küme`** veri tipinin, **matematikten bildiğimiz kümelerle sıkı bir bağlantısı vardır**. Dolayısıyla matematikteki kümelere ait *birleşim*, *kesişim* ve *fark* gibi işlemleri bu **`küme`** veri tipiyle rahatlıkla gerçekleştirebilirsiniz.

Şimdi, dilerseniz *matematiksel kümelere ilişkin metotları* inceleyelim.

UNION()

İki kümenin birleşim kümesini bulmak için `union` metodunu kullanıyoruz:

```
>>> küme1 = {"A", "B", "C"}
>>> küme2 = {"a", "b", "c"}
>>> küme3 = küme1.union(küme2)
>>> küme3
{'B', 'c', 'C', 'b', 'A', 'a'}
```

DIFFERENCE()

Eğer iki kümenin **fark kümesini** bulmak isterseniz **`difference()`** metodunu kullanabilirsiniz:

```
>>> küme1 = {"a", "b", "c"}
>>> küme2 = {"b", "c", "d"}
>>> küme1.difference(küme2)
{'a'}
```

SYMMETRIC_DIFFERENCE()

`difference()` metodunu kullanarak **iki küme arasındaki farklı ögeleri bulmayı öğrenmiştik**. Örneğin elimizde şöyle iki küme var diyelim:

```
>>> a = set([1, 2, 5])
>>> b = set([1, 4, 5])
```

Eğer **a** kümesinin **b** kümesinden farkını bulmak istersek şu komutu veriyoruz:

```
>>> a.difference(b)
{2}
```

Demek ki **a** kümesinde bulunup **b** kümesinde bulunmayan öge **2** imiş.

Bir de **b** kümesinde bulunup **a** kümesinde bulunmayan ögelere bakalım:

```
>>> b.difference(a)
{4}
```

Bu da bize **4** çıktısını verdi. Demek ki bu öge, **b** kümesinde bulunuyor ama **a** kümesinde bulunmuyormuş. *Peki, ya kümelerin ikisinde de bulunmayan ögeleri aynı anda nasıl alacağız?* İşte bu noktada yardımımıza **symmetric_difference()** adlı metot yetişecek:

```
>>> a.symmetric_difference(b)
{2, 4}
```

Böylece **iki kümede de bulunmayan ögeleri aynı anda almış olduk**. Yani **a** ve **b** kümelerinin *simetrik farkını* elde ettik.

INTERSECTION()

intersection() metodu bize iki kümenin **kesişim kümesini** verecektir:

```
>>> k1 = set([1, 2, 3, 4])
>>> k2 = set([1, 3, 5, 7])
>>> k1.intersection(k2)
{1, 3}
```

Bu metot yardımıyla iki küme arasındaki **ortak elemanları** bulmuş oluyoruz.

DIFFERENCE_UPDATE()

Bu metot, **`difference()`** metodundan **elde edilen sonuca göre bir kümenin güncellenmesini** sağlar.

Yani? Hemen bir **örnek** verelim:

```
>>> k1 = set([1, 2, 3])
>>> k2 = set([1, 3, 5])
>>> k1.difference_update(k2)
>>> print(k1)
{2}
```

Gördüğünüz gibi bu metot **`k1`**'in **`k2`**'den farkını aldı ve bu farkı kullanarak **`k1`**'i yeniden oluşturdu. **`k1`** ile **`k2`** arasındaki **tek fark `2`** adlı öge idi. Dolayısıyla **`difference_update()`** metodunu uyguladığımızda **`k1`**'in ögelerinin silinip yerlerine **`2`** adlı ögenin geldiğini görüyoruz.

SYMMETRIC_DIFFERENCE_UPDATE()

Bu metodun **ne işe yaradığını** bir örnekle anlatalım:

```
>>> a = set([1,2, 5])
>>> b = set([1,4, 5])
>>> a.symmetric_difference_update(b)
>>> print(a)
{2, 4}
```

Gördüğünüz gibi **`a`** kümesinin **eski ögeleri gitti**, yerlerine **`a`** kümesi ile **`b`** kümesinin **simetrik farkı** (yani **`symmetric_difference()`** metoduyla elde edilen çıktı) **geldi**. Yani **`a`** kümesi, **`symmetric_difference()`** metodunun sonucuna göre **güncellenmiş** oldu.

INTERSECTION_UPDATE()

Bu metodun görevi, **`intersection()`** metodundan **elde edilen sonuca göre bir kümenin güncellenmesini sağlamaktır**:

```
>>> k1 = set([1, 2, 3])
>>> k2 = set([1, 3, 5])
>>> k1.intersection_update(k2)
>>> print(k1)
{1, 3}
>>> print(k2)
{1, 3, 5}
```

Gördüğünüz gibi **`intersection_update()`** metodu **`k1`**'in bütün ögelerini sildi ve yerlerine **`k1`** ve **`k2`**'nin **kesişim kümesinin elemanlarını** koydu.

ISDISJOINT()

Bu metodun çok basit bir görevi vardır. Bu metodu kullanarak **iki kümenin kesişim kümesinin boş olup olmadığı sorgulayabilirsiniz**. Hatırlarsanız aynı işi bir önceki bölümde gördüğümüz **`intersection()`** metodunu kullanarak da yapabiliyorduk. Ama eğer hayattan tek beklentiniz iki kümenin kesişim kümesinin boş olup olmadığını, yani **bu iki kümenin ortak eleman içerip içermediğini** öğrenmekse basitçe **`isdisjoint()`** metodundan yararlanabilirsiniz:

```
>>> a = set([1, 2, 3])
>>> b = set([2, 4, 6])
>>> a.isdisjoint(b)
False
```

Gördüğünüz gibi **`a`** ve **`b`** kümesinin kesişimi boş olmadığı için yani, bu **iki küme ortak en az bir öge barındırdığı için** **`isdisjoint()`** metodu **`False`** çıktısı veriyor. Burada temel olarak şu soruyu sormuş oluyoruz: "*a ve b ayrık kümeler mi?*"

Python da bize cevap olarak "**Hayır değil**," anlamına gelen **`False`** çıktısını veriyor... Çünkü **`a`** ve **`b`** kümelerinin **ortak bir elemanı var** (2).

Bir de şuna bakalım:

```
>>> a = set([1, 3, 5])
>>> b = set([2, 4, 6])
>>> a.isdisjoint(b)
True
```

Burada **a** ve **b** kümeleri **ortak hiç bir elemana sahip olmadığı için** "**Doğru**" anlamına gelen **True** çıktısını elde ediyoruz.

ISSUBSET()

Bu metot yardımıyla, **bir kümenin bütün elemanlarının başka bir küme içinde yer alıp yer almadığını sorgulayabiliriz**. Yani bir kümenin, başka bir kümenin **alt kümesi olup olmadığını** bu metot yardımıyla öğrenebiliriz. Eğer bir küme, başka bir kümenin **alt kümesi ise** bu metot bize **True** değerini verecek, eğer **değilse False** çıktısını verecektir:

```
>>> a = set([1, 2, 3])
>>> b = set([0, 1, 2, 3, 4, 5])
>>> a.issubset(b)
True
```

Bu örnekte **True** çıktısını aldık, çünkü **a** kümesinin bütün ögeleri **b** kümesi içinde yer alıyor. Yani **a**, **b**'nin **alt kümesi**dir.

ISSUPERSET()

Bu metot, bir önceki bölümde gördüğümüz **`issubset()`** metoduna benzer. Matematik derslerinde gördüğümüz "**kümeler**" konusunda hatırladığınız "**b kümesi a kümesini kapsar**" ifadesini bu metotla gösteriyoruz. Önce bir önceki derste gördüğümüz örneğe bakalım:

```
>>> a = set([1, 2, 3])
>>> b = set([0, 1, 2, 3, 4, 5])
>>> a.issubset(b)
True
```

Buradan, "**a kümesi b kümesinin alt kümesidir**" sonucuna ulaşıyoruz.

Bir de şuna bakalım:

```
>>> a = set([1, 2, 3])
>>> b = set([0, 1, 2, 3, 4, 5])
>>> b.issuperset(a)
True
```

Burada ise "**b kümesi a kümesini kapsar**" sonucunu elde ediyoruz. Yani **b** kümesi **a** kümesinin bütün elemanlarını içinde barındırıyor.

KÜME İŞLEÇLERİ

Yukarıda, metotlar yardımıyla yaptığımız bazı işlemleri ayrıca **çeşitli işleçler** yardımıyla da hâlledebiliriz. Gelin, isterseniz bu işleçlerin neler olduğunu kısaca gözden geçirelim.

BİRLEŞİM KÜMESİ İŞLECİ

İki kümenin **birleşim kümesini bulmak için `union()`** metodu yerine **|** işlecini kullanabilirsiniz:

```
>>> küme1 | küme2
{'B', 'b', 'a', 'c', 'C', 'A'}
```

Fark Kümesi İşleci

İki kümenin fark kümesini bulmak için **`difference()`** metodu yerine **-** işlecini de kullanabilirsiniz:

```
>>> küme1 - küme2
{'a'}
```

Simetrik Fark Kümesi İşleci

İki kümenin simetrik fark kümesini bulmak için **`symmetric_difference`** metodu yerine **^** işlecini de kullanabilirsiniz:

```
>>> a ^ b
{2, 4}
```

Kesişim Kümesi İşleci

İki kümenin kesişim kümesini bulmak için **`intersection()`** metodu yerine **&** işlecini de kullanabilirsiniz:

```
>>> k1 & k2
{1, 3}
```

Alt küme Sorgulama İşleci

Bir kümenin, **bir başka kümenin alt kümesi olup olmadığını** sorgulamak için **`issubset()`** metodu yerine **<=** işlecini kullanabilirsiniz:

```
>>> a = set([1, 2, 3])
>>> b = set([0, 1, 2, 3, 4, 5])
>>> a <= b
True
```

Üst Küme Sorgulama İşleci

Bir kümenin, **bir başka kümenin üst kümesi olup olmadığını** sorgulamak için **`issuperset()`** metodu yerine **>=** işlecini kullanabilirsiniz:

```
>>> a = set([1, 2, 3])
>>> b = set([0, 1, 2, 3, 4, 5])
>>> b >= a
True
```

Öz Alt Küme Sorgulama İşleci



Bir kümenin kendisiyle aynı olmayan tüm alt kümelerine, o kümenin **öz alt kümeleri** denir.

Bir kümenin, bir başka kümenin **öz alt kümesi olup olmadığını** sorgulamak için **<** işlecini kullanabilirsiniz:

```
>>> a = set([1, 2, 3])
>>> b = set([0, 1, 2, 3, 4, 5])
>>> a < b
True
```

Bu öz alt küme sorgulama işleminin, bir metot karşılığı yoktur.

Öz Üst Küme Sorgulama İşleci



Bir kümenin kendisiyle aynı olmayan tüm üst kümelerine, o kümenin **öz üst kümeleri** denir.

Bir kümenin, bir başka kümenin **öz üst kümesi olup olmadığını** sorgulamak için **>** işlecini kullanabilirsiniz:

```
>>> a = set([1, 2, 3])
>>> b = set([0, 1, 2, 3, 4, 5])
>>> b > a
True
```

Bu öz üst küme sorgulama işleminin bir metot karşılığı yoktur.

FONKSİYONLAR 20

Daha önce de ifade ettiğimiz gibi fonksiyonlar, **belirli işlevleri yerine getirmemizi sağlayan birtakım araçlardır**. Biz bu bölüme gelinceye kadar; `print()`, `input()`, `len()`, `range()` ve `dir()` gibi pek çok fonksiyon öğrendik. Bu fonksiyonların özelliği bunların Python programlama dilinin bir parçası olmalarıdır. Bunlara teknik olarak "**hazır fonksiyonlar**" veya "**gömülü fonksiyonlar**" adını veriyoruz. Bu bölümün ilerleyen sayfalarında gömülü fonksiyonlardan söz edeceğiz. Ama biz bu bölümde asıl olarak **kendi fonksiyonlarımızı nasıl tanımlayacağımızı** öğreneceğiz.

Şimdi şöyle bir düşünün: Diyelim ki çalıştığınız iş yerinde her gün bir yerlere bir dilekçe gönderiyorsunuz. Aslında farklı yerlere gönderdiğiniz bu dilekçeler temel olarak aynı içeriğe sahip. Yani mesela dilekçeyi Mehmet Bey'e göndereceğiniz zaman yazıya Mehmet Bey'in adını, Ahmet Bey'e göndereceğiniz zaman ise Ahmet Bey'in adını yazıyorsunuz. Ayrıca tabii dilekçe üzerindeki tarih bilgilerini de güne göre düzenliyorsunuz.

Mantıklı bir insan, yukarıdaki gibi bir durumda elinde bir dilekçe taslağı bulundurur ve sadece değiştirmesi gereken kısımları değiştirip dilekçeyi hazır hâle getirir. Her defasında aynı bilgileri en baştan yazmaz.

Dilerseniz anlattığımız bu durumu **Python programlama dili** ile temsil etmeye çalışalım:

```
print('''Sayın Mehmet Bey,
19.06.2020 tariginde yaptığımız başvurunun sonuçlandırılması
hususunda yardımlarınızı rica ederiz.
Saygılarımızla,
Orçun Kunek''')
```

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

```
print('''Sayın Ahmet Bey,
15.01.2010 tariginde yaptığımız başvurunun sonuçlandırılması
hususunda yardımlarınızı rica ederiz.
Saygılarımızla,
Orçun Kunek''')
```

Burada dikkat ederseniz iki dilekçenin **metin içeriği aslında aynı**dır. Sadece bir iki yer değişiyor. Bu kodları bu şekilde yazmak oldukça **verimsiz** bir yoldur. Çünkü bu sistem hem programcıya **çok vakit kaybettirir** hem fazlasıyla **kopyala-yapıştır işlemi gerektirir** hem de bu kodların **bakımını yapmak çok zordur**. Eğer dilekçe metni üzerinde bir değişiklik yapmak isterseniz program içindeki ilgili kodları **tek tek bulup düzeltmeniz gerekir**. Yani mesela yukarıdaki iki dilekçenin aynı program içinde olduğunu varsayarsak dilekçe metnindeki bir hatayı düzeltmek istediğimizde aynı düzeltmeyi **birkaç farklı yerde** yapmamız gerekir.

Örneğin yukarıdaki dilekçe metninde bir **yazım hatası** var. İlk satırda "`tarihinde`" yazacağımıza "`tariginde`" yazmışız. Tabii biz Mehmet Bey'e yazdığımız dilekçenin metnini Ahmet Bey'e yazacağımız dilekçeyi hazırlarken **kopyala-yapıştır** yaptığımız için aynı hata Mehmet Bey'e gidecek dilekçede de var. Şimdi, biz bu yazım hatasını düzeltmek istediğimizde, bütün dilekçe metinlerini **tek tek gözden geçirmemiz** gerekecektir. Hele bir de dilekçe sayısı çok fazlaysa bu işlemin ne kadar zor olacağını bir düşünün. Tabii bu basit durum için her metin düzenleyicide bulunan **bul-değiştir** özelliğini kullanabiliriz. **Ama işler her zaman bu kadar kolay olmayabilir**. Bu bakım ve taslaklama işini kolaylaştıracak bir çözüm olsa ve bizi aynı şeyleri tekrar tekrar yazmaktan kurtarsa ne kadar güzel olurdu değil mi? İşte, Python buna benzer durumlar için bize **fonksiyon** denen bir araç sunar. Biz de bu bölümde, bu faydalı aracı olabildiğince ayrıntılı bir şekilde incelemeye çalışacağız.

Fonksiyonları Tanımlamak

Python'da fonksiyonları kullanabilmek için öncelikle **fonksiyonları tanımlamamız** gerekiyor. **Fonksiyon tanımlamak için `def`** adlı bir deyimden yararlanacağız. Python'da fonksiyonları şöyle tanımlıyoruz:

```
def fonksiyon_adı():
```

Gördüğünüz gibi önce **`def`** deyimini, ardından da fonksiyonumuzun adını yazıyoruz. **Fonksiyon adı olarak istediğiniz her şeyi yazabilirsiniz**. Ancak fonksiyon adı belirlerken fonksiyonun ne işe yaradığını anlatan kısa bir isim belirlemeniz hem sizin hem de kodlarınızı okuyan kişilerin işini bir hayli kolaylaştıracaktır. Ayrıca fonksiyonları tanımlarken **en sona parantez** ve **iki nokta üst üste** işaretlerini de koymayı **unutmuyoruz**.

Böylece ilk fonksiyonumuzu tanımlamış olduk. Ama henüz işimiz bitmedi. Bu fonksiyonun bir işe yarayabilmesi için bunun **altını doldurmamız gerekiyor**. Fonksiyon tanımının iki nokta üst üste işareti ile bitmesinden, sonraki satırın girintili olması gerektiğini tahmin etmişsinizdir. Gelin, isterseniz biz bu **fonksiyonun altını çok basit bir şekilde dolduralım**:

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

Böylece eksiksiz bir fonksiyon tanımlamış olduk. Burada dikkat etmemiz gereken en önemli şey, **`def fonksiyon_adı():`** satırından sonra gelen kısmın **girintili** yazılmasıdır.

Şimdi, isterseniz bu fonksiyonu **nasıl kullanabileceğimizi görelim**:

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

Bu kodları bir metin düzenleyiciye kaydedip çalıştırdığımızda hiçbir çıktı elde edemeyiz. Çünkü biz burada fonksiyonumuzu **sadece tanımlamakla yetindik**. *Henüz bu fonksiyonun işletilmesini sağlayacak kodu yazmadık*. Şimdi ,bu fonksiyonun hayat kazanmasını sağlayacak kodları girebiliriz:

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

fonksiyon_adı()
```

Burada fonksiyonun **çalışmasını sağlayan şey**, en son satırdaki **`fonksiyon_adı()`** kodudur. Bu satırı ekleyerek daha önce tanımladığımız fonksiyonu çağırmış oluyoruz. Bu satıra teknik olarak **fonksiyon çağrısı** (*function call*) adı verilir. Asla unutmayın, bir fonksiyon tanımlandıktan sonra ancak herhangi bir şekilde "**çağrılırsa**" çalışır.

Yukarıdaki kodları içeren programı çalıştırdığımızda ekrana "**`merhaba dünya!`**" çıktısının verildiğini göreceğiz.

Tebrikler! Böylece ilk eksiksiz fonksiyonunuzu hem **tanımlamış** hem de **çağırmış** oldunuz.

Dilerseniz, bu konunun en başında verdiğimiz dilekçe örneğini de bir **fonksiyon hâline getirelim**:

```
def dilekce_gönder():
    print('''\
    Sayın Mehmet Bey,
    19.06.2020 tarihinde yaptığımız başvurunun sonuçlandırılması
    hususunda yardımlarınızı rica ederiz.
```

```
    Saygılarımızla,
    Orçun Kunek''')

dilekce_gönder()
```

Elbette bu örnek Python'daki fonksiyonların bütün yeteneklerini ortaya koymaktan aciz. Üstelik bu fonksiyon, en başta bahsettiğimiz sorunları da çözemiyor henüz. Ama ne yapalım... Şu ana kadar öğrendiklerimiz ancak bu kadarını yapmamıza müsaade ediyor. Biraz sonra, öğreneceklerimiz sayesinde fonksiyonlarla **çok daha faydalı** ve **manalı** işler yapabileceğiz.

Yeni bir bölüme geçmeden önce, isterseniz fonksiyonlarla ilgili olarak buraya kadar öğrendiğimiz kısmı biraz irdeleyelim.

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

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

Python'da bir fonksiyon tanımı şöyle bir yapıya sahiptir:

```
def fonksiyon_adı():
```

Fonksiyonlarımıza vereceğimiz adların, **olabildiğince betimleyici olması** herkesin hayrınadır.

Elbette bir fonksiyonun **işlevli** olabilmesi için **sadece tanımlanması yetmez**. Ayrıca tanımladığımız fonksiyonun bir de **gövdesi**nin olması gerekir. Fonksiyon gövdesini **girintili** olarak yazıyoruz. Dolayısıyla Python'da bir **fonksiyon temel olarak iki kısımdan oluşur**. İlk kısım fonksiyonun tanımlandığı **başlık kısmı**, ikinci kısım ise fonksiyonun içeriğini oluşturan **gövde** kısmıdır. *Başlık ve gövde dışında kalan her şey fonksiyonun da dışındadır*. **Bir fonksiyon, gövdedeki girintili kısmın bittiği yerde biter**. Örneğin şu bir fonksiyondur:

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

Bu fonksiyon **`def selamla():`** satırıyla başlar, **`print("nasılsın?")`** satırıyla biter. **Fonksiyonların çalışabilmesi için bu fonksiyonlar tanımlandıktan sonra çağrılmalıdır**. Örneğin yukarıdaki fonksiyonu şöyle çağırıyoruz:

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

selamla()
```

Dediğimiz gibi bu fonksiyon, **`def selamla():`** **satırıyla başlar**, **`print("nasılsın?")`** **satırıyla biter**. Fonksiyon çağrısı dediğimiz **`selamla()`** satırı **bu fonksiyonun dışındadır**. Çünkü bu satır, **`selamla()`** fonksiyonunun gövdesini oluşturan girintili kısmın **dışında** yer alıyor.

Eğer bu söylediklerimiz size biraz kafa karıştırıcı gelmişse hiç endişe etmenize gerek yok. Tam olarak ne demek istediğimizi biraz sonra gayet net bir şekilde anlamanızı sağlayacak örnekler vereceğiz.

Dilerseniz bu bölümü kapatmadan önce fonksiyonlarla ilgili **birkaç basit örnek daha yaparak** bu konuya ısınmanızı sağlayalım:

```
def tek():
    print("Girdiğiniz sayı bir tek sayıdır!")

def çift():
```

```
    print("Girdiğiniz sayı bir çift sayıdır!")

sayı = input("Lütfen bir sayı giriniz: ")

if int(sayı) % 2 == 0:
    çift()
else:
    tek()
```

Burada **`tek()`** ve **`çift()`** adlı iki fonksiyon tanımladık. **`tek()`** adlı fonksiyonun görevi, ekrana "**`Girdiğiniz sayı bir tek sayıdır!`**" çıktısı vermek. **`çift()`** adlı fonksiyonun görevi ise ekrana "**`Girdiğiniz sayı bir çift sayıdır!`**" çıktısı vermek.

Daha sonra Python'ın **`input()`** fonksiyonunu kullanarak kullanıcıdan bir sayı girmesini istiyoruz. Ardından da kullanıcı tarafından girilen bu sayının **tek mi yoksa çift mi** olduğunu denetliyoruz. Eğer sayı, **2'ye tam olarak bölünüyorsa çifttir**. **Aksi hâlde bu sayı tek**tir.

Burada **`çift()`** ve **`tek()`** adlı fonksiyonları nasıl çağırdığımıza dikkat edin. Eğer kullanıcının girdiği sayı 2'ye **tam olarak bölünüyorsa** yani, bu sayı **çiftse** daha önce tanımladığımız **`çift()`** adlı fonksiyon devreye girecektir.

Yok, eğer kullanıcının verdiği sayı 2'ye **tam olarak bölünmüyorsa** o zaman da **`tek()`** adlı fonksiyon **devreye girer**.

Bu kodlarda özellikle, fonksiyonların nerede başlayıp nerede bittiğine dikkat edin. Daha önce de dediğimiz gibi bir fonksiyon **`def`** parçacığıyla başlar, gövdesindeki girintili alanın sonuna kadar devam eder. **Girintili alanın dışında kalan bütün kodlar o fonksiyonun dışındadır**. Mesela yukarıdaki örnekte **`tek()`** ve **`çift()`** birbirinden **bağımsız** iki fonksiyondur. Bu fonksiyonlardan sonra gelen **`sayı`** değişkeni de **fonksiyon alanının dışında yer alır**.

Fonksiyonlarda Parametre Kullanımı

Yukarıdaki örneklerde gördüğünüz gibi bir fonksiyon tanımlarken fonksiyon **adını belirledikten sonra bir parantez işareti kullanıyoruz**. Bu parantez işaretleri, örneklerde de gördüğümüz gibi hiçbir bilgi içermeyebilir. Yani bu parantezler **boş** olabilir. Ancak yapacağımız işin niteliğine göre bu parantezlerin içine birtakım bilgiler yerleştirmemiz gerekebilir. İşte bu bilgilere **parametre** adı verilir. **Zaten Python fonksiyonları da asıl gücünü bu parametrelerden alır**. Şimdi, şöyle bir **örnek** verdiğimizi düşünelim:

```
def selamla():
    print("merhaba, benim adım istihza!")

selamla()
```

Burada fonksiyonumuzu tanımladıktan sonra en sondaki **`selamla()`** satırı yardımıyla fonksiyonumuzu çağırdık. Yani tanımladığımız fonksiyona hayat öpücüğü verdik.

Ancak yukarıda tanımladığımız fonksiyon **oldukça kısıtlı bir kullanım alanına sahiptir**. Bu fonksiyon, ekrana yalnızca "**`merhaba, benim adım istihza!`**" çıktısını verebilir. Eğer fonksiyonların yapabildiği şey bundan ibaret olsaydı, emin olun fonksiyonlar hiçbir işimize yaramazdı. Ama şimdi öğreneceğimiz **parametre kavramı** sayesinde fonksiyonlarımıza takla atmayı öğreteceğiz. Gelin, isterseniz çok basit bir örnekle başlayalım:

```
def selamla(isim):
    print(f"merhaba, benim adım {isim}!")
```

Belki fark ettiniz belki de fark etmediniz ama burada aslında çok önemli bir şey yaptık. Fonksiyonumuza bir **parametre** verdik! Şimdiye kadar tanımladığımız fonksiyonlarda, fonksiyon tanımı hep **boş bir parantezden** oluşuyordu. Ancak bu defa parantezimizin içinde bir **değişken adı** görüyoruz. Bu değişkenin adı **isim**. Fonksiyonlar söz konusu olduğunda, parantez içindeki bu değişkenlere **parametre** adı verilir.

Fonksiyonumuzu tanımlarken bir parametre belirttikten sonra bu parametreyi fonksiyonun gövdesinde de kullandık. İsterseniz şimdi, tanımladığımız bu **fonksiyonun bir işe yarayabilmesi için fonksiyonumuzu çağıralım**:

```
selamla("istihza")
```

Kodları bir arada görelim:

```
def selamla(isim):
    print(f"merhaba, benim adım {isim}!")

selamla("istihza")
```

Bu kodları çalıştırdığımızda şöyle bir **çıktı** alırız:

```
merhaba, benim adım istihza!
```

Burada **`selamla()`** adlı fonksiyonu **`istihza`** argümanı ile birlikte çağırdık. Böylece çıktıda "**`istihza`**" değerini aldık. **Burada terminolojiyle ilgili ufak bir not düşelim**: Fonksiyonlar tanımlanırken parantez içinde belirtilen değerlere **parametre** adı verilir. Aynı fonksiyon çağrılırken parantez içinde belirtilen değerlere ise **argüman** adı verilir. Ama her iki durum için de **parametre adının kullanıldığını** bazı yerlerde görebilirsiniz.

Neyse... Biz bu terminoloji işini bir kenara bırakıp yolumuza devam edelim.

Eğer **`selamla()`** adlı fonksiyonu farklı bir argüman ile çağırırsak elbette alacağımız çıktı da **farklı** olacaktır:

```
def selamla(isim):
    print(f"merhaba, benim adım {isim}!")

selamla("Ahmet Efendi")
```

Bu defa çıktımız şöyle:

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

Burada önemli olan nokta, **`selamla()`** adlı fonksiyonun **bir adet parametreye** sahip olmasıdır. Dolayısıyla bu fonksiyonu **argümansız olarak veya birden fazla argümanla çağıramayız**. Yani fonksiyonu şu şekillerde çağırmak **hata** almamıza yol açacaktır:

```
selamla("Ahmet", "Mehmet")
```

veya:

```
selamla()
```

Sanırım bu örnekler fonksiyonlardaki parametre kavramının ne işe yaradığını net bir biçimde ortaya koyuyor. Daha sonraki bölümlerde bu parametre kavramından bolca yararlanacağız. İleride göreceğimiz örnekler ne kadar karmaşık olursa olsun **işin temeli** aynen yukarıda anlattığımız gibidir. Eğer **bu temeli iyi kavrarsanız başka yerlerde göreceğiniz daha karmaşık örnekleri anlamakta zorlanmazsınız**.

Gelin, isterseniz bu noktada **basit bir örnek** verelim.

Hatırlarsanız yukarıda **`tek()`** ve **`çift()`** adlı iki fonksiyon tanımlamıştık. Şimdi, bu fonksiyonları **daha işlevli** hâle getirelim.

Dikkatlice bakın:

```
def tek_mi_çift_mi(sayı):
    if sayı % 2 == 0:
        print(f"{sayı} sayısı bir çift sayıdır")
    else:
        print("f{sayı} sayısı bir tek sayıdır")

tek_mi_çift_mi(12)
```

Tanımladığımız bu **`tek_mi_çift_mi()`** adlı fonksiyon **`sayı`** adlı bir argüman alıyor. Daha sonra bu argümanın 2'ye **tam olarak bölünüp bölünmediğini kontrol ederek girilen sayının tek mi yoksa çift mi olduğunu söylüyor**. Eğer sayı **`2`**'ye tam bölünüyorsa yani, bölme işleminden kalan değer **`0`** ise o hâlde bu sayı bir **çift** sayıdır. Aksi hâlde bu sayı bir **tek** sayıdır.

Bu fonksiyonu şöyle kullandık:

```
tek_mi_çift_mi(12)
```

Burada **`tek_mi_çift_mi()`** adlı fonksiyona **argüman** olarak **`12`** sayısını verdik. Fonksiyonumuz bu sayının **`2`**'ye tam bölünüp bölünmediğini kontrol etti. Bu sayı **`2`**'ye bölündüğünde **bölme** işleminden kalan değer **`0`** olduğu için dolayısıyla **`12`** sayısı **`2`**'ye **tam bölündüğü için bu sayının bir çift sayı** olduğunu anladı.

Dilerseniz bu anlattıklarımızla ilgili ufak bir **örnek** daha yapalım.

Python'da, verilen sayıları toplayan **`sum()`** adlı bir fonksiyon bulunur. Bu fonksiyonu şöyle kullanıyoruz:

```
>>> sayılar = [45, 90, 43]
>>> sum(sayılar)
178
```

`sum()` fonksiyonu, kendisine **argüman** olarak verilen bir **sayı** listesinin ögelerini birbiriyle **toplayıp sonucu bize bildiriyor**.

Ancak Python'da bu **`sum()`** fonksiyonuna benzer bir şekilde bir sayı listesini alıp ögelerini **birbiriyle çarpan hazır bir fonksiyon bulunmuyor**.

Python, bize bu işlem için herhangi bir hazır fonksiyon sunmadığından böyle bir fonksiyona ihtiyaç duyduğumuzda **kendi yöntemimizi kendimiz icat etmek zorundayız**.

O hâlde hemen başlayalım. Diyelim ki elimizde şöyle bir **sayı listesi** var:

```
>>> sayılar = [45, 90, 43]
```

Soru şu: *Acaba bu listedeki sayıları birbiriyle nasıl çarparız?* Bunu yapmanın en kolay yolu, listedeki bütün sayıları 1'le çarpıp, bütün değerleri tek bir değişken içinde toplamaktır.

Öncelikle değeri **1** olan bir **başlangıç değişkeni** belirlememiz gerekiyor:

```
>>> a = 1
```

Daha sonra listedeki bütün sayıları **a** değişkeninin değeriyle **çarpıp** bu değeri yine **a** değişkenine atayacağız:

```
>>> for i in sayılar:
...     a = a * i
```

Yukarıdaki kodu şöyle de yazabilirsiniz:

```
>>> for i in sayılar:
...     a *= i
```

Böylece listedeki bütün sayıların çarpımını gösteren değer **a** değişkenine atanmış oldu. İsterseniz bu **a** değerini yazdırıp sonucu kendi gözlerinizle görebilirsiniz:

```
>>> print(a)
174150
```

Gördüğünüz gibi listedeki bütün sayıların çarpımı **a** değişkeninde tutuluyor. **Kodları topluca görelim**:

```
>>> sayılar = [45, 90, 43]
>>> a = 1
>>> for i in sayılar:
...     a *= i
...
>>> print(a)
174150
```

Şimdi, şöyle bir düşünün. Diyelim ki bir program yazıyorsunuz ve bu programın değişik yerlerinde, bir liste içindeki sayıları birbiriyle çarpmanız gerekiyor. **Bunun için şöyle bir yol takip edebilirsiniz:**

- Önce bir **sayı listesi** tanımlarsınız,
- Daha sonra, değeri `1` olan bir **değişken tanımlarsınız**,
- Son olarak da listedeki **bütün sayıları bu değişkenin değeriyle çarpıp** elde ettiğiniz değeri tekrar bu değişkene atarsınız,
- Program içinde **gereken her yerde bu işlemleri tekrar edersiniz**.

Bu yöntem, sizi aynı şeyleri sürekli tekrar etmek zorunda bıraktığı için oldukça **verimsiz** bir yoldur. İşte Python'daki fonksiyonlar böyle bir durumda hemen **devreye girer**. Mantıklı bir programcı, yukarıdaki gibi her defasında tekerleği yeniden icat etmek yerine, tekerleği bir kez icat eder sonra gereken yerlerde icat ettiği bu tekerleği kullanır. Biz de yukarıdaki işlemleri içeren bir fonksiyonu **tek bir kez** tanımlayacağız ve program içinde gereken yerlerde bu fonksiyonu çağıracağız. Şimdi gelin, **yukarıdaki işlemleri içeren fonksiyonumuzu tanımlayalım**:

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

Böylece taslağımızı oluşturmuş olduk. Artık bu fonksiyonu kullanarak istediğimiz bir **sayı grubunu birbiriyle rahatlıkla çarpabiliriz**. Bunun için yapmamız gereken tek şey **`çarp()`** adlı fonksiyonu **çağırmak**:

```
çarp([3, 5, 7])
```

Burada dikkat ederseniz **`çarp()`** fonksiyonuna verdiğimiz sayıları bir liste içine aldık. Çünkü **`çarp()`** fonksiyonu **tek bir parametre alıyor**. Eğer bu fonksiyonu şu şekilde çağırırsanız **hata** alırsınız:

```
çarp(3, 5, 7)
```

Çünkü burada **`çarp()`** fonksiyonuna **birden fazla argüman** verdik. Hâlbuki fonksiyonumuz sadece **tek bir argüman** alıyor. Elbette dilerseniz önce bir sayı listesi tanımlayabilir, ardından da bu listeyi fonksiyona argüman olarak verebilirsiniz:

```
sayılar = [3, 5, 7]
çarp(sayılar)
```

Şimdi, kodları **topluca** görelim:

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

sayılar = [3, 5, 7]
çarp(sayılar)
```

Bu kodları çalıştırdığınızda **`105`** sonucunu alırsınız.

Bu arada, yukarıdaki kodlarda **`çarp()`** fonksiyonuna ait **`liste`** adlı parametrenin yalnızca **temsili** bir isimlendirme olduğuna dikkat edin. Program içinde daha sonra fonksiyonu çağırırken argüman olarak kullanacağınız değerin **liste adını taşıma zorunluluğu yoktur**. Mesela bizim örneğimizde **`çarp()`** fonksiyonunu **sayılar adlı bir liste ile çağırdık**.

Fonksiyonların, işimizi ne kadar kolaylaştırdığını görüyorsunuz. Yapmak istediğimiz işlemleri bir **fonksiyon** olarak tanımlıyoruz ve gerektiği yerde bu fonksiyonu çağırarak işimizi hâllediyoruz. Eğer işlemlerde bir değişiklik yapmak gerekirse tanımladığımız fonksiyonu yeniden düzenlememiz yeterli olacaktır.

Eğer fonksiyonlar olmasaydı, fonksiyon içinde tek bir kez tanımladığımız işlemi programın farklı yerlerinde **defalarca** tekrar etmemiz gerekecekti. Üstelik işlemlerde bir değişiklik yapmak istediğimizde de bütün programı baştan sona tarayıp değişiklikleri tek tek elle uygulamak zorunda kalacaktık.

Dilerseniz en başta verdiğimiz dilekçe örneğine tekrar dönelim ve o durumu **fonksiyonlara uyarlayalım**:

```
def dilekçe_gönder(kime, tarih, kimden):
    print(f'''\
        Sayın {kime},

        {tarih} tarihinde yaptığımız başvurunun sonuçlandırılması
        hususunda yardımlarınızı rica ederiz.

        Saygılarımızla,
        {kimden}
    ''')

dilekçe_gönder("Mehmet Bey", "19.06.2020", "Orçun Kunek")
```

Gördüğünüz gibi yukarıdaki fonksiyon işimizi bir hayli kolaylaştırıyor. Burada fonksiyonumuzu sadece **bir kez** oluşturuyoruz. Ardından dilekçeyi kime göndereceksek **uygun bilgileri kullanarak yeni bir fonksiyon çağrısı yapabiliriz**. Mesela dilekçeyi Ahmet Bey'e göndereceksek şöyle bir satır yazmamız yeterli olacaktır:

```
dilekçe_gönder("Ahmet Bey", "21.06.2020", "Erdener Topçu")
```

Ayrıca dilekçe metninde bir **değişiklik** yapmak istediğimizde sadece fonksiyon gövdesini düzenlememiz yeterli olacaktır.

Sıralı Parametreler

Python'da fonksiyonlarınızı **istediğiniz sayıda parametre** ile tanımlayabilirsiniz. Mesela şöyle bir fonksiyon tanımlayabilirsiniz:

```
def kayıt_ekle(isim, soyisim, şehir, meslek, tel, adres):
    kayıt = {}

    kayıt[f"{isim} {soyisim}"] = [şehir, meslek, tel, adres]
    print("Bağlantı bilgileri kayıtlara eklendi!\n")

    for k, v in kayıt.items():
        print(k)
        print("-"*len(k))
        for i in v:
            print(i)

kayıt_ekle("Orçun", "Kunek", "Adana", "Şarkıcı",
"0322 123 45 67", "Baraj Yolu")
```

Yine burada, kodlarımızın çirkin bir görüntü oluşturmaması için ögeleri nasıl **alt satıra** geçirdiğimize dikkat edin. Eğer **Python kodlarına duyarlı bir metin düzenleyici kullanıyorsanız** (*mesela IDLE*) **virgül işaretlerinden sonra ENTER** tuşuna bastığınızda **düzgün girinti yapısı otomatik olarak oluşturulacaktır**.

Bu kodlarda, öncelikle **`kayıt_ekle()`** adlı bir fonksiyon tanımladık. Bu fonksiyon toplam **altı adet parametre alıyor**. Bunlar; **`isim`**, **`soyisim`**, **`şehir`**, **`meslek`**, **`tel`** ve **`adres`**.

Tanımladığımız bu fonksiyonun gövdesinde ilk olarak **`kayıt`** adlı bir sözlük oluşturduk. Bu sözlük, ekleyeceğimiz bağlantıya dair **bilgileri tutacak**. Daha sonra, oluşturduğumuz bu sözlüğe öge ekliyoruz. Buna göre fonksiyon parametrelerinden olan **`isim`** ve **`soyisim`**; **`kayıt`** adlı sözlükteki **anahtar kısmını** oluşturacak. **`şehir`**, **`meslek`**, **`tel`** ve **`adres`** değerleri ise **`kayıt`** adlı sözlükteki değer kısmını meydana getirecek.

Sözlüğü oluşturduktan sonra ekrana "**`Bağlantı bilgileri kayıtlara eklendi!`**" biçiminde bir **mesaj** yazdırıyoruz.

Daha sonra da **kayıt** adlı sözlüğün ögelerini belli bir düzen çerçevesinde ekrana yazdırıyoruz.

Bu işlemi nasıl yaptığımıza dikkat edin. Python sözlüklerinde **items()** adlı bir metot bulunduğunu biliyorsunuz. Bu metot yardımıyla bir sözlük içinde bulunan **bütün anahtar** ve **değer çiftlerini** elde edebiliyoruz. **Sözlükler** konusunu anlatırken bu metottan bahsetmiştik ama dilerseniz buna bu bölümde de bir **örnek** verelim.

Diyelim ki elimizde şöyle bir **sözlük** var:

```
>>> sözlük = {"programlama dili": "Python",
... "metin düzenleyici": "Kate"}
```

Şimdi, **items()** metodunu bu sözlük üzerine uygulayalım:

```
>>> print(sözlük.items())
[('programlama dili', 'Python'),
('metin düzenleyici', 'Kate')]
```

Gördüğünüz gibi **sözlük** adlı sözlüğe ait bütün anahtar ve değerler bir liste içinde **iki ögeli demetler** hâlinde toplandı. Şimdi şu kodu yazalım:

```
>>> for k, v in sözlük.items():
...     print(k, v)
```

Buradan şöyle bir **çıktı** alırız:

```
programlama dili Python
metin düzenleyici Kate
```

items() metodunun ne işe yaradığını tekrar anlattığımıza göre biz tekrar **kayıt_ekle()** fonksiyonumuzu incelemeye devam edebiliriz. Biraz önce anlattığımız gibi **kayıt_ekle()** fonksiyonu içindeki **for k, v in kayıt.items():** satırında **k** değişkeni **kayıt** adlı sözlükteki anahtarları, **v** değişkeni ise aynı sözlükteki değerleri temsil ediyor. Böylece sözlükteki **anahtar** ve **değer çiftlerini birbirinden ayırmış olduk**.

Bu işlemi yaptıktan sonra, öncelikle **k** değişkenini ekrana yazdırıyoruz. Yani **kayıt** adlı sözlükteki anahtar kısmını almış oluyoruz. Bu kısım, fonksiyondaki **isim** ve **soyisim** parametrelerinin değerini gösteriyor.

print("-"*len(k)) satırı ise bir önceki satırda ekrana yazdırdığımız isim ve soyismin altına, **isim ve soyisim değerlerinin uzunluğu kadar çizgi çekmemizi sağlıyor**. Böylece **isim** ve **soyismi**, fonksiyondaki öteki bilgilerden görsel olarak ayırmış oluyoruz.

En son olarak da **kayıt** adlı sözlüğün **değer** kısmındaki ögeleri tek tek ekrana yazdırıyoruz.

Fonksiyonumuzu başarıyla tanımladıktan sonra, sıra geldi bu **fonksiyonu çağırmaya**.

Fonksiyonumuzu sırasıyla, "**Orçun**", "**Kunek**", "**Adana**", "**Şarkıcı**", "**0322 123 45 67**" ve "**Baraj Yolu**" argümanlarıyla birlikte çağırıyoruz. Böylece bu kodları çalıştırdığımızda şöyle bir **çıktı** elde ediyoruz:

```
Bağlantı bilgileri kayıtlara eklendi!

Orçun Kunek
-----------
Adana
Şarkıcı
0322 123 45 67
Baraj Yolu
```

İsterseniz, **kayıt_ekle()** fonksiyonundaki *parametreleri kendiniz yazmak yerine kullanıcıdan almayı da tercih edebilirsiniz*. **Mesela şöyle bir şey yazabilirsiniz**:

```
def kayıt_ekle(isim, soyisim, şehir, meslek, tel, adres):
    kayıt = {}
    kayıt[f"{isim} {soyisim}"] = [şehir, meslek, tel, adres]
    print("\nBağlantı bilgileri kayıtlara eklendi!\n")
    for k, v in kayıt.items():
        print(k)
        print("-"*len(k))
        for i in v:
            print(i)
```

```
isi = input("isim: ")
soy = input("soyisim: ")
seh = input("şehir: ")
mes = input("meslek: ")
tel = input("telefon: ")
adr = input("adres: ")

kayıt_ekle(isi, soy, seh, mes, tel, adr)
```

Yukarıdaki fonksiyonları kullanırken dikkat etmemiz gereken çok önemli bir nokta var. **`kayıt_ekle()`** adlı fonksiyonu kullanırken argüman olarak vereceğimiz değerlerin sırası **büyük önem taşıyor**. Yani bu değerleri, fonksiyon tanımındaki sıraya göre yazmamız gerek. Buna göre **`kayıt_ekle()`** fonksiyonunu çağırırken ilk argümanımızın **isim**, ikincisinin **soyisim**, üçüncüsünün **şehir**, dördüncüsünün **meslek**, beşincisinin **telefon**, altıncısının ise **adres** olması gerekiyor. Aksi hâlde, bizim yukarıda verdiğimiz örnekte çok belli olmasa da, fonksiyondan alacağımız **çıktı** hiç de beklediğimiz gibi olmayabilir. Ancak takdir edersiniz ki bu kadar fazla sayıda parametrenin *sırasını akılda tutmak hiç de kolay bir iş değil*. İşte bu noktada Python'daki **isimli argümanlar devreye girer** ve bizi büyük bir dertten kurtarır.

İsimli Parametreler

Şimdi, isimli parametrelerin nasıl kullanıldığını inceleyelim. İsterseniz **yukarıda verdiğimiz örnekten yararlanalım**:

```
def kayıt_ekle(isim, soyisim, şehir, meslek, tel, adres):
    kayıt = {}
    kayıt[f"{isim} {soyisim}"] = [şehir, meslek, tel, adres]
    print("\nBağlantı bilgileri kayıtlara eklendi!\n")
    for k, v in kayıt.items():
        print(k)
        print("-"*len(k))
        for i in v:
            print(i)

kayıt_ekle(isim = "Abdurrahman", soyisim = "Çelebi",
           meslek = "Öğretmen", tel = "0212 123 45 67",
           şehir = "İstanbul", adres = "Çeliktepe")
```

Gördüğünüz gibi **`kayıt_ekle()`** adlı fonksiyonumuzun parametrelerini **isimleriyle birlikte** kullanıyoruz. Böylece parametreleri **sıra gözetmeden** kullanma imkânımız oluyor. Bizim örneğimizde bütün parametreler karakter dizilerinden oluştuğu için **isimli parametre** kullanmanın faydası, ilk bakışta pek belli olmuyor. Ama özellikle sayılar ve karakter dizilerini **karışık** olarak içeren fonksiyonlarda yukarıdaki yöntemin faydası daha belirgindir. Mesela şu örneğe bakalım:

```
def terfi_ettir(kişi, e_poz, y_poz, e_maaş, z_oranı):
    print(f"{kişi}, {e_poz} pozisyonundan {y_poz} "
          "pozisyonuna terfi etmiştir!")
    print(f"Bu kişinin {e_maaş} TL olan maaşı "
          f"{e_maaş + (e_maaş * z_oranı/100)} TL'ye
yükseltilmiştir!")

terfi_ettir("Ahmet Öncel", "İş Geliştirme Uzmanı",
            "İş Geliştirme Müdürü", 7500, 25)
```

İşte bu örnekte **argüman sıralamasının önemi** ortaya çıkar.

Eğer burada, mesela "**Ahmet Öncel**" argümanıyla **7500** argümanının yerini değiştirirseniz programınız **hata** verecektir. Çünkü bu fonksiyonda biz **7500** sayısını kullanarak **aritmetik bir işlem** yapıyoruz. Eğer **7500**'ün olması gereken yerde bir sayı yerine **karakter dizisi** olursa **aritmetik işlem yapılamaz**.

Bu arada yukarıdaki fonksiyonun sağa doğru çok fazla yayılarak çirkin bir kod görüntüsü vermemesi için **satırları nasıl alta kaydırdığımıza** dikkat edin.

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

```
def terfi_ettir(kişi, e_poz, y_poz, e_maaş, z_oranı):
    print(f"{kişi}, {e_poz} pozisyonundan {y_poz} "
          "pozisyonuna terfi etmiştir!")
    print(f"Bu kişinin {e_maaş} TL olan maaşı "
          f"{e_maaş + (e_maaş * z_oranı/100)} TL'ye
yükseltilmiştir!")

terfi_ettir(e_maaş = 7500, e_poz = "İş Geliştirme Uzmanı",
            kişi = "Ahmet Öncel", y_poz = "İş Geliştirme Müdürü",
            z_oranı = 25)
```

Teknik olarak söylemek gerekirse Python fonksiyonlarında sırasına göre çağrılan argümanlara "**sıralı argüman**" (*positional argument*), ismine gören çağrılan argümanlara ise "**isimli argüman**" (*keyword argument*) adı verilir. Python fonksiyonlarında, sıralı argümanlardan bolca yararlanılır. Ancak parametre sayısının çok fazla olduğu durumlarda, isimli argümanları kullanmak da işinizi bir hayli kolaylaştırabilir.

Zorunlu İsimli Parametreler

Yukarıdaki örneklerde gördüğünüz gibi bir fonksiyon tanımında geçen parametreleri, fonksiyon çağrısı sırasında hem **konumlarına** göre hem de **isimlerine** göre kullanabiliyoruz. Ancak bazı durumlarda, fonksiyonunuz içinde tanımladığınız parametrelerin **yalnızca ismiyle** çağrılmasını isteyebilirsiniz.

Mesela bu tür parametreleri, **`print()`** fonksiyonundan tanıyoruz. Bildiğiniz gibi bu fonksiyonun **`sep`** ve **`end`** gibi isimli parametrelerini mutlaka **isimleriyle birlikte** kullanmamız gerekiyor. İşte bu bölümde bu tür **zorunlu isimli parametreleri** nasıl tanımlayacağımızı öğreneceğiz.

Diyelim ki bir **listedeki herhangi bir ögeyi değiştiren** bir fonksiyon yazmak istiyoruz.

Bunun için şöyle bir kod yazabiliriz:

```
def değiştir(liste, eski, yeni):
    indeks = liste.index(eski)
    liste[indeks] = yeni
    print(liste)
```

Burada öncelikle fonksiyonumuzun **adını** ve **parametrelerini** belirledik:

```
def değiştir(liste, eski, yeni):
```

Fonksiyonumuzun adı **`değiştir()`**. Bu fonksiyon toplam **üç parametre** alıyor: **`liste`**, **`eski`** ve **`yeni`**.

Buna göre bu fonksiyonu çağırırken üç argüman kullanacağız: Üzerinde değişiklik yapmak istediğimiz liste, değiştirmek istediğimiz liste ögesi ve bu ögenin yeni değeri.

Artık fonksiyonumuzun **gövdesini** yazabiliriz. Önce değiştirilecek ögenin liste içindeki **sırasını** tespit edelim. Bunun için listelerin **`index()`** metodunu kullanacağımızı biliyorsunuz:

```
indeks = liste.index(eski)
```

Bu indeks bilgisine göre liste içindeki bir ögenin **yeni değerini** belirleyebiliriz:

```
liste[indeks] = yeni
```

Son olarak da listeyi **ekrana yazdırıyoruz**:

```
print(liste)
```

Bu fonksiyonu iki şekilde kullanabilirsiniz. Parametreleri **sırasına** göre çağırarak ve parametreleri **ismine** göre çağırarak.

Bu anlattıklarımızı somutlaştırmak için bir **liste örneği** verelim:

```
yerler = [“bolvadin”, “kilis”, “siverek”, “iskenderun”, “adana”,
“istanbul”]
```

Şimdi, yukarıda tanımladığımız fonksiyonu **bu liste üzerinde kullanalım**:

```
değiştir(yerler, “bolvadin”, “istinye”)
```

Bu kodları çalıştırdığımızda şu **çıktıyı** alıyoruz:

```
[‘istinye’, ‘kilis’, ‘siverek’, ‘iskenderun’, ‘adana’, ‘istanbul’]
```

Burada **`değiştir()`** adlı fonksiyonu **sıralı argümanlarla** çağırdık. Eğer istersek bu fonksiyonu **isimli argümanlarla** da çağırabiliriz:

```
değiştir(liste=yerler, eski=”bolvadin”, yeni=”istinye”)
```

Ama eğer siz bu fonksiyonun sadece isimli argümanlarla kullanılmasını istiyorsanız **yıldız işaretinden** yararlanabilirsiniz:

```
def değiştir(*, liste, eski, yeni):
    indeks = liste.index(eski)
    liste[indeks] = yeni
    print(liste)
```

Bu fonksiyon tanımına göre * işaretinden sonraki bütün parametrelerin **sadece isimleriyle birlikte** kullanılması gerekiyor:

```
değiştir(liste=yerler, eski="bolvadin", yeni="istinye")
```

Eğer isterseniz bu * işaretini farklı noktalara da yerleştirebilirsiniz. **Mesela**:

```
def değiştir(liste, *, eski, yeni):
```

Bu tanıma göre liste parametresi **ismine** veya **sırasına** göre çağrılabilir. Ama eski ve yeni parametrelerinin isimleriyle çağrılması gerekir:

```
değiştir(yerler, eski="bolvadin", yeni="istinye")
```

Zorunlu Sıralı Parametreler

Eğer tanımladığınız bir fonksiyondaki bazı parametrelerin **sadece sırasına göre** çağrılmasını isterseniz / işaretinden yararlanabilirsiniz:

```
def değiştir(liste, /, eski, yeni):
```

Bu tanıma göre / işaretinden önce gelen parametreler **isimleriyle çağrılamaz**. Bunların yalnızca **sırasına göre çağrılması gerekir**:

```
değiştir(yerler, eski="bolvadin", yeni="istinye")
```

Sınırsız Sayıda Sıralı Parametre

Bildiğiniz gibi **`print()`** fonksiyonuna **istediğimiz sayıda argüman** verebiliyoruz:

```
>>> print(1, 2, 3, “a”, “b”, “c”)
```

Ama mesela **`len()`** fonksiyonu böyle değil. Bu fonksiyon, **sınırlı sayıda argüman** alabiliyor:

```
>>> len(“istihza”)
7

>>> len(“abc”, “def”)
Traceback (most recent call last):
  File “<stdin>”, line 1, in <module>
TypeError: len() takes exactly one argument (2 given)
```

Gördüğünüz gibi bu fonksiyona 1'den fazla argüman verdiğimizde **hata** mesajı alıyoruz.

Bu tür sınırlı sayıda argüman alan fonksiyonları **tanımlamak** kolaydır. Bir fonksiyonu tanımlarken **kaç tane parametre belirtirseniz o sayıda argüman alır**:

```
def fonk(arg):
    print(arg)
```

Peki, acaba **`print()`** gibi **sınırsız sayıda argüman** alabilen fonksiyonları nasıl tanımlarız?

Esasında bu da çok kolaydır. **Dikkatlice bakın**:

```
def fonk(*args):
    for arg in args:
        print(arg)
```

Eğer fonksiyon tanımında tek bir parametre belirtir ve bu tek parametrenin sol tarafına da bir **yıldız işareti** koyarsanız artık o fonksiyon **sınırsız sayıda argümanla çağrılabilir**:

```
fonk(1, 2, 3, 4, "a", "b", "c")
```

Sınırsız Sayıda İsimli Parametre

Python'da sınırsız sayıda sıralı argüman alabilen fonksiyonlar tanımlayabileceğimiz gibi **sınırsız sayıda isimli argüman** alabilen fonksiyonlar da tanımlayabiliriz. Bu iş için yine **yıldız işaretinden** yararlanacağız. Ama bu kez yıldız işaretini **iki kez** kullanacağız. Dikkatlice bakın:

```
def fonk(**kwargs):
    for k, v in kwargs.items():
        print(f"{k} = {v}")

fonk(arg1=1, arg2=2, arg3=3, arg4=4, arg5="a", arg6="b", arg7="c")
```

Eğer arzu ederseniz tek bir fonksiyon içinde hem **sınırsız sayıda sıralı parametre** hem de **sınırsız sayıda isimli parametre** de tanımlayabiliriz:

```
def fonk(*args, **kwargs):
    for arg in args:
        print(arg)

    for k, v in kwargs.items():
        print(k, v)

fonk(1, 2, 3, a="a", b="b")
```

Gömülü Fonksiyonlar

Python'ın en güçlü yanlarından biri de bu dilin yapısında bulunan **gömülü fonksiyonlardır** (*built-in functions*). Peki, bir fonksiyonun gömülü olması ne anlama gelir?

Gömülü demek, "**Python'ın içinde yer alan**" demektir. Yani gömülü fonksiyonlar, Python programlama dilinin içinde yer alan **hazır fonksiyonlardır**. Mesela daha önce öğrendiğimiz ve sık sık kullandığımız **`range()`** fonksiyonu **gömülü bir fonksiyondur**. Bu fonksiyon Python'ın içinde, kullanılmaya hazır bir şekilde bekler. Bu fonksiyonun işlevini yerine getirebilmesi için **tanımlanmasına gerek yoktur**. Python geliştiricileri, bu fonksiyonu tanımlamış ve dilin içine "gömmüşlerdir".

Mesela **`len()`** ve **`sum()`** fonksiyonları da birer **gömülü fonksiyondur**. Python'daki gömülü fonksiyonların listesine *https://docs.python.org/3/library/functions*.html adresinden erişebilirsiniz. Bir program yazarken **özel bir işlevi yerine getirmeniz gerektiğinde** yukarıdaki adresi mutlaka kontrol edin. Bakın bakalım sizden önce birisi tekerleği zaten icat etmiş mi? Örneğin **tam sayıları** (**`integer`**) **ikili sayılara** (**`binary`**) çevirmeniz gerekiyorsa; oturup bu işlemi yapan bir fonksiyon tanımlamaya çalışmanız **boş bir çaba** olur. Bunun yerine hâlihazırda tanımlanıp dilin içine gömülmüş olan **`bin()`** fonksiyonunu kullanabilirsiniz. Yukarıdaki adreste bunun gibi onlarca gömülü fonksiyon göreceksiniz.

Peki, gömülü fonksiyonlarla bizim kendi yazdığımız fonksiyonlar arasında tam olarak ne fark vardır?

Bir defa, gömülü fonksiyonlar oldukça **hızlı** ve **verimlidir**. Çünkü bu fonksiyonlar, Python geliştiricileri tarafından **özel olarak** optimize edilmiştir. İkincisi (*ve belki de en önemlisi*), bu fonksiyonlar **her an hazır ve nazırdır**. Yani bu fonksiyonları kullanabilmek için özel bir şey yapmanıza gerek yoktur. Bu fonksiyonları, istediğinizde veya bu fonksiyonlar lazım olduğunda doğrudan kullanabilirsiniz. **Ama bizim tanımladığımız fonksiyonlar böyle değildir**. Kendi yazdığımız fonksiyonları kullanabilmek için bu fonksiyonu içeren modülü öncelikle **içe aktarmamız** (*import*) gerekir. İlerleyen derslerde bu işlemlerin nasıl yapılacağını ele alacağız.

Global Deyimi

Bu bölümde **global** adlı bir deyimden söz edeceğiz. İsterseniz **global**'in ne olduğunu anlatmaya çalışmak yerine doğrudan bir örnekle işe başlayalım. Diyelim ki şöyle bir şey yazdık:

```
def fonk():
    a = 5
    print(a)

fonk()
```

Burada her zamanki gibi **fonk()** adlı bir **fonksiyon tanımladık** ve daha sonra bu **fonksiyonu çağırdık**. Sonuç olarak bu fonksiyonu çalıştırdığımızda **5** çıktısını aldık.

Gördüğünüz gibi fonksiyon içinde **a** adlı bir değişkenimiz var. Şimdi, şöyle bir şey yazarak bu **a** değişkenine ulaşmaya çalışalım:

```
def fonk():
    a = 5
    print(a)

fonk()
print(f"a'nın değeri: {a}")
```

Bu kodları çalıştırdığımızda şöyle bir **hata** alırız:

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

Bu hata mesajı bize **a** diye bir değişken **olmadığını söylüyor**. Hâlbuki biz **fonk()** adlı fonksiyon içinde bu **a** değişkenini tanımlamıştık, değil mi? O hâlde neden Python **a** değişkenini bulamadığından yakınıyor?

Hatırlarsanız bu bölümün en başında, bir fonksiyonun sınırlarının ne olduğundan söz etmiştik. Buna göre yukarıdaki **`fonk()`** adlı fonksiyon **`def fonk():`** ifadesiyle başlıyor, **`print(a)`** ifadesiyle bitiyor.

Bu fonksiyonun etkisi bu alanla **sınırlıdır**. Python'da buna **isim alanı** (*namespace*) adı verilir. Herhangi bir fonksiyon içinde tanımlanan her şey, o fonksiyonun **isim alanıyla sınırlıdır**. Yani mesela yukarıdaki fonksiyon içinde tanımladığımız **`a`** değişkeni yalnızca bu fonksiyon sınırları dâhilinde geçerlidir. Bu alanın dışına çıkıldığında **`a`** değişkeninin herhangi bir **geçerliliği yoktur**. O yüzden Python, yukarıdaki gibi bir kod yazdığımızda **`a`** değişkenini bulamayacaktır. İsterseniz bu durumu daha iyi anlayabilmek için yukarıdaki örneği şöyle değiştirelim:

```
def fonk():
    a = 5
    print(a)

fonk()
a = 10
print(f"a'nın değeri: {a}")
```

Bu kodları çalıştırdığımızda şöyle bir **çıktı** alırız:

```
5
a'nın değeri: 10
```

Buradaki ilk **`5`** sayısı fonksiyon içindeki **`a`**'nın değerini, alt satırdaki **`10`** değeri ise **`a`**'nın fonksiyon **sınırları dışındaki değerini** gösteriyor. Gördüğünüz gibi **`a`** değişkenini fonksiyon dışında da kullanabilmek için bu değişkeni dışarıda **tekrar tanımlamamız** gerekiyor. Peki, biz fonksiyon içindeki **`a`** değişkenine fonksiyon dışından da erişemez miyiz? *Elbette erişebiliriz*. Ama bunun için **`global`** adlı bir deyimden yararlanmamız gerekir. Dilerseniz yukarıda ilk verdiğimiz örnek üzerinden giderek bu **`global`** deyimini anlamaya çalışalım:

```
def fonk():
    a = 5
    print(a)
```

```
fonk()
print(f"a'nın değeri: {a}")
```

Kodları bu şekilde yazdığımızda Python'ın bize bir **hata** mesajı göstereceğini biliyoruz. Şimdi, bu kodlara şöyle bir **ekleme** yapalım:

```
def fonk():
    global a
    a = 5
    print(a)

fonk()
print(f"a'nın değeri: {a}")
```

Burada yaptığımız şey, fonksiyonu tanımladıktan sonra fonksiyon gövdesinin ilk satırına **`global a`** diye bir satır eklemekten ibaret. Bu ifade fonksiyon içindeki **`a`** adlı değişkenin **`global`** olduğunu, yani fonksiyonun kendi sınırları dışında da **geçerli bir değer** olduğunu söylüyor. Bu kodları çalıştırdığımızda şöyle bir **çıktı** alıyoruz:

```
5
a'nın değeri: 5
```

Gördüğünüz gibi **`global`** deyimi bir fonksiyon içindeki değerlere **fonksiyon dışından da erişmemize yardımcı oluyor**. Şimdi, şöyle bir şey yazdığımızı düşünün:

```
a = 10
def fonk():
    a = 5
    return a

print("a'nın fonksiyon içindeki değeri", fonk())
print("a'nın fonksiyon dışındaki değeri: ", a)
```

Buradaki **`return`** deyimine **takılmayın**, biraz sonra bunun tam olarak ne işe yaradığını göreceğiz. Biz yalnızca yukarıdaki kodların çıktısına odaklanalım.

Burada öncelikle bir **a** değişkeni tanımladık. Bu değişkenin değeri **10**. Ardından da **fonk()** adlı bir fonksiyon oluşturduk. Bu fonksiyon içinde değeri **5** olan bir **a** değişkeni daha tanımladık. Fonksiyon dışında ise **iki adet çıktı** veriyoruz. Bunlardan **ilki** fonksiyon içindeki **a** değişkeninin değerini gösteriyor. **İkincisi** ise fonksiyon dışındaki **a** değişkeninin değerini... Yani bu kodları çalıştırdığımızda şöyle bir **çıktı** elde ediyoruz:

```
a'nın fonksiyon içindeki değeri 5
a'nın fonksiyon dışındaki değeri: 10
```

Burada fonksiyon içinde ve dışında aynı adda iki değişken tanımlamamıza rağmen, Python'daki **isim alanı kavramı** sayesinde bu **iki değişkenin değeri birbirine karışmıyor**. Ama bir de şu kodlara bakın:

```
a = 10

def fonk():
    global a
    a = 5
    return a

print("a'nın fonksiyon içindeki değeri", fonk())
print("a'nın fonksiyon dışındaki değeri: ", a)
```

Burada bir önceki kodlara ilave olarak **global a** satırını yazdık. Bu defa çıktımız şöyle oldu:

```
a'nın fonksiyon içindeki değeri 5
a'nın fonksiyon dışındaki değeri: 5
```

Gördüğünüz gibi **global** deyimi fonksiyon dışındaki **a** değişkenini **sildi**.

Şimdi, bu noktada kendimize şu soruyu sormamız lazım: *Acaba bizim istediğimiz şey bu mu?*

Özellikle üzerinde birkaç farklı kişinin çalıştığı büyük projelerde böyle bir özellik ne tür sorunlara yol açabilir?

Üzerinde pek çok farklı kişinin çalıştığı büyük projelerde, **`global`** deyiminin büyük sıkıntılar doğurabileceği apaçık ortada. Siz programın bir yerine bir değişken tanımlamaya çalışırken, başka bir geliştirici bir fonksiyon içinde **`global`** deyimini kullanarak fonksiyon dışındaki **aynı adlı değişkenlerin değerini birbirine katmış olabilir**. İşte bu tür sıkıntılardan ötürü, her ne kadar faydalı bir araçmış gibi görünse de **`global`** **deyiminin sakıncaları faydalarından çoktur**. O yüzden yazdığınız programlarda **`global`** deyiminden mümkün olduğunca uzak durmanızda fayda var.

RETURN DEYİMİ

Hatırlarsanız bir önceki bölümde şöyle bir fonksiyon tanımlamıştık:

```
def fonk():
    a = 5
    return a
```

Dikkat ederseniz burada **`return`** diye bir şey kullandık. Bu kelime Türkçede "**vermek**, **döndürmek**, **iade etmek**" gibi anlamlara gelir. Buna göre yukarıdaki fonksiyon **a** değişkenini döndürüyor. *Peki, bu ne demek?* Açıklayalım...

Python'da her fonksiyonun bir **dönüş değeri** vardır. Yani Python'daki bütün fonksiyonlar bir **değer döndürür**. **Burada döndürmekten kastımız**: Bir işlemin sonucu olarak ortaya çıkan değeri vermektir. Mesela "**Bu fonksiyonun dönüş değeri bir karakter dizisidir**" veya "**Bu fonksiyon bir karakter dizisi döndürür**" dediğimiz zaman kastettiğimiz şey, bu fonksiyonun işletilmesi sonucu ortaya çıkan değerin bir **karakter dizisi** olduğudur. Örneğin yukarıdaki fonksiyon **a** adlı değişkeni döndürüyor ve bu **a** değişkeni bir tam sayı olduğu için fonksiyonumuzun dönüş değeri bir **tam sayı**dır. *Peki, fonksiyonlar bir değer döndürüyor da ne oluyor? Yani bir fonksiyonun herhangi bir değer döndürmesinin kime ne faydası var?* İsterseniz bunu daha iyi anlayabilmek için yukarıdaki örneği bir de şöyle yazalım:

```
def fonk():
    a = 5
    print(a)
```

Burada herhangi bir **değer döndürmüyoruz**. Yaptığımız şey, bir **a** değişkeni belirleyip **print()** fonksiyonunu kullanarak bunu ekrana bastırmaktan ibaret. **Bu arada şunu da belirtelim**: Her ne kadar biz bir fonksiyonda açık açık bir değer döndürmesek de o fonksiyon **otomatik** olarak bir değer döndürecektir. Herhangi bir değer döndürmediğimiz fonksiyonlar otomatik olarak **None** diye bir değer döndürür. **Bunu şu şekilde test edebiliriz**:

Yukarıdaki fonksiyonu şu şekilde çağıralım:

```
print(fonk())
```

Fonksiyonu bu şekilde çağırdığımızda şöyle bir **çıktı** aldığımızı göreceksiniz:

```
5
None
```

İşte burada gördüğümüz **None** değeri, **fonk()** adlı fonksiyonumuzun **otomatik** olarak döndürdüğü değerdir.

Yukarıdaki fonksiyonu, **print()** olmadan da çağırabileceğimizi biliyorsunuz:

```
fonk()
```

Bu defa **a** değişkeninin değeri ekrana dökülecek, ancak **None** değerini göremeyeceğiz. **Şimdi, şu fonksiyona bakalım**:

```
def fonk():
    a = 5
    return a

fonk()
```

Burada ise ekranda herhangi bir **çıktı** göremeyiz. Bunun nedeni, bu fonksiyonda herhangi bir '**print**' işlemi yapmıyor olmamızdır. Biz burada sadece **a** değişkenini döndürmekle yetiniyoruz. Yani ekrana herhangi bir **çıktı** vermiyoruz.

Bu fonksiyondan **çıktı** alabilmek için fonksiyonu şöyle çağırmamız gerekir:

```
print(fonk())
```

"*Peki, bütün bu anlattıkların ne işe yarar?*" diye sorduğunuzu duyar gibiyim.

Bir defa şunu anlamamız lazım: **`print()`** ve **`return` aynı şeyler değildir**. **`print()`** fonksiyonu, **bir mesajın ekrana basılmasını sağlar**. **`return`** deyimi ise **herhangi bir değerin döndürülmesinden sorumludur**. Siz bir fonksiyondan bir değer döndürdükten sonra, o değerle ne yapacağınız tamamen size kalmış. Eğer tanımladığınız bir fonksiyonda bir değer döndürmek yerine o değeri ekrana **basarsanız** (*yani* **`print()`** *fonksiyonunu kullanırsanız*) fonksiyonun işlevini bir bakıma **kısıtlamış** olursunuz. Çünkü **tanımladığınız bu fonksiyonun tek görevi**, **bir değeri ekrana basmak olacaktır**. Ama eğer o değeri ekrana basmak yerine döndürmeyi tercih ederseniz; fonksiyonu hem ekranda bir mesaj göstermek için hem de başka işler için kullanabilirsiniz.

Bunu anlayabilmek için şöyle bir **örnek** verelim:

```
def ekrana_bas():
    a = 5
    print(a)

print("a değişkeninin değeri: ", ekrana_bas())
```

Bu kodları çalıştırdığımızda şöyle bir **çıktı** alırız:

```
5
a değişkeninin değeri: None
```

Görüyorsunuz, işler hiç de istediğimiz gibi gitmedi. Hâlbuki biz "**`a değişkeninin değeri: 5`**" gibi bir **çıktı** alacağımızı zannediyorduk. Daha önce de dediğimiz gibi bir fonksiyondan herhangi bir değer döndürmediğimizde **otomatik** olarak **`None`** değeri döndürüldüğü için çıktıda bu değeri görüyoruz.

Ama eğer yukarıdaki fonksiyonu şu şekilde tanımlasaydık işimiz daha kolay olurdu:

```
def ekrana_bas():
    a = 5
    return a

print("a değişkeninin değeri: ", ekrana_bas())
```

Gördüğünüz gibi bu defa istediğimiz **çıktıyı** aldık. Bir de şu örneğe bakın:

```
def sayı_işle():
    sor = int(input("bir sayı girin: "))
    return sor

sayı = sayı_işle()

print("girdiğiniz sayı: ", sayı)

if sayı % 2 == 0:
    print("girdiğiniz sayı çifttir")
else:
    print("girdiğiniz sayı tektir")

print("girdiğiniz sayının karesi: ", sayı ** 2)
print("girdiğiniz sayının küpü: ", sayı ** 3)
```

Burada **`sayı_işle()`** adlı fonksiyonda **kullanıcıya bir sayı sorup** bu sayıyı döndürüyoruz. Daha sonra fonksiyonu çağırırken bu döndürdüğümüz değerle **istediğimiz işlemi** yapabiliyoruz. İsterseniz bu fonksiyonu bir de **`return`** yerine **`print()`** ile tanımlamayı deneyin; o zaman ne demek istediğimi gayet net bir biçimde anlayacaksınız.

Son olarak daha önce verdiğimiz şu kodları ele alalım:

```
def değiştir(*, liste, eski, yeni):
    indeks = liste.index(eski)
    liste[indeks] = yeni
    print(liste)
```

Burada da aslında son satırda **`print(liste)`** diyerek kodlarımızın **kapsamını** ve **esnekliğini** sınırlandırmış oluyoruz. Bu hâliyle **`değiştir()`** fonksiyonu yalnızca **`liste`** değerini ekrana yazdıracaktır. **Ama eğer bu kodları şöyle yazarsak işler değişir**:

```
def değiştir(liste, eski, yeni):
    indeks = liste.index(eski)
    liste[indeks] = yeni
    return liste
```

Bu şekilde fonksiyonun bize **`liste`** adlı bir **değer döndürmesini** (*yani sunmasını*) sağladık. Artık sunulan bu değerle ne yapacağımız tamamen bize kalmış. İstersek bu değeri ekrana yazdırabiliriz. Mesela:

```
diller = [“Java”, “Python”, “C++”, “C”]

yeni_liste = değiştir(diller, “Java”, “Go”)

print(yeni_liste)
```

Veya dönen bu değer ile başka **ilave işlemler** yapabiliriz:

```
yeni_liste.append(“Haskell”)

print(yeni_liste)
```

PASS DEYİMİ

'**pass**' kelimesi Türkçede **geçmek**, **aşmak** gibi anlamlara gelir. Python'da ise bu deyim, **herhangi bir işlem yapmadan geçeceğimiz durumlarda** kullanılır. *Peki, bu ne demek?*

Şu örneğe bir bakalım:

```
def deneme():
    liste = []
    while True:
        a = input("Giriniz: ")
        if a == "0":
            pass
        elif a == "iptal":
            break
        else:
            liste.append(a)
            print(liste)

deneme()
```

Burada gördüğümüz gibi eğer kullanıcı **0** değerini girerse bu değer **listeye eklenmeyecek**, Python hiçbir şey yapmadan **bu satırı atlayacaktır**. İşte **pass** buna benzer durumlarda, "**hiçbir şey yapmadan yola devam et!**" anlamı katar kodlarımıza.

pass deyimini yukarıdaki durumlar dışında bir de şöyle bir durumda kullanabilirsiniz: Diyelim ki bir program yazıyorsunuz. Bu programda bir fonksiyon tanımlayacaksınız. Fonksiyonun isminin ne olacağına karar verdiniz, ama fonksiyon içeriğini nasıl yazacağınızı düşünmediniz. Eğer program içinde sadece fonksiyonun ismini yazıp bırakırsanız programınız çalışma sırasında **hata** verecektir. İşte böyle bir durumda **pass** deyimi imdadınıza yetişir. **Bu deyimi kullanarak şöyle bir şey yazabilirsiniz**:

```
def bir_fonksiyon():
    pass
```

Fonksiyon tanımlarken fonksiyon gövdesini **boş bırakamazsınız**. Çünkü dediğimiz gibi eğer gövdeyi boş bırakırsanız **programınız çalışmaz**. Böyle bir durumda, yukarıda gösterdiğimiz gibi fonksiyonu tanımlayıp gövde kısmına da bir **pass** deyimi yerleştirebilirsiniz. Daha sonra gövde kısmına ne yazacağınıza karar verdiğinizde bu **pass** deyimini silebilirsiniz.

Böylece fonksiyonlar konusunu tamamlamış olduk. Artık yeni ve çok önemli bir konu olan "**Modüller**"e başlayabiliriz.

MODÜLLER

21

Bu bölümde, daha önce yalnızca birkaç örnek üzerinden üstünkörü bir şekilde ele aldığımız **modüller** konusunu çok daha *ayrıntılı* bir şekilde inceleyeceğiz.

Hatırlarsanız geçen bölümlerde Python'da **iki tür modül** olduğundan söz etmiştik. Bunlar standart **kütüphane modülleri** ve **haricî modüller**di.

Standart kütüphane modüllerine örnek olarak **`platform`**, **`random`**, **`webbrowser`** ve **`datetime`** modüllerini vermiştik. Haricî modül örneği olarak ise **`cx_Freeze`** adlı modülü kısaca incelemiştik.

Standart kütüphane modüllerinin özelliği, bunların Python programlama dilinin bir **parçası** olmasıdır. Yani bu tür modülleri kullanabilmek için herhangi bir **ek yazılım kurmamıza gerek yoktur**. Bu modülleri **`import`** komutunu kullanarak programlarımıza dâhil edebiliyoruz.

Haricî modüller ise Python programlama dilinin bir **parçası değildir**. O yüzden bu modülleri kullanabilmek için bunları öncelikle sistemimize kurmamız gerekir. Mesela **`cx_Freeze`** adlı haricî modülü sistemimize şu şekilde kurduğumuzu hatırlıyorsunuz:

```
pip install cx_Freeze
```

Bu bölümde hem standart kütüphane modüllerine hem de haricî modüllere daha yakından bakacağız. Ayrıca bu bölümde **kendi modüllerimizi nasıl yazacağımızı da ele alacağız**.

Dilerseniz ilk olarak tüm modüller için geçerli olan birtakım genel noktaları ele alalım.

Modüllerin İçe Aktarılması

Hangi türden olursa olsun Python'daki tüm modüller **`import`** komutuyla **içe aktarılır**. Bir modülü kullanabilmenin ilk şartı o modülü programımız içine **dâhil etmektir**. İşte bu dâhil etme işlemini **`import`** komutuyla gerçekleştiriyoruz.

Ancak Python'da tüm modülleri **`import`** komutuyla içe aktarıyor olsak da bu komutu kullanmanın da *değişik yolları vardır*. Gelin şimdi, bu **farklı içe aktarma yöntemlerini inceleyelim**.

IMPORT MODÜL_ADI

Dediğimiz gibi Python programlama dilinde bir modülü kullanabilmenin **ön şartı o modülü içe aktarmaktır**. Biz, şimdiye kadar yaptığımız tüm örneklerde modülleri hep şu şekilde içe aktardık:

```
>>> import random
>>> import datetime
>>> import webbrowser
```

Eğer arzu ederseniz birkaç farklı modülü tek bir **`import`** komutuyla da içe aktarabilirsiniz:

```
>>> import random, datetime, webbrowser
```

Bir modülü bu şekilde içe aktardıktan sonra **artık modülün adını kullanarak** o modül içindeki **fonksiyon** ve **niteliklere erişebileceğimizi biliyoruz**:

```
>>> webbrowser.open("https://www.google.com")
>>> random.randint(1, 500)
>>> datetime.datetime.today()
```

Python'da bir modülü içe aktarmanın en temel yolu **`import`** komutunu yukarıda anlattığımız şekilde kullanmaktır. Ama tabii bir modülü içe aktarmanın tek yolu bu değildir. Gelin şimdi, modülleri içe aktarmanın başka yöntemlerini de kısa kısa ele alalım.

IMPORT MODÜL_ADI AS BAŞKA_BİR_AD

Bazı durumlarda bir modülü kendi adıyla değil de **başka bir adla içe aktarmak isteyebilirsiniz**. Mesela eğer içe aktaracağınız modülün adı **uzunsa** ve her defasında bu modül adını kullanmak size **zor geliyorsa** modülü **daha kısa bir adla içe aktarmayı tercih edebilirsiniz**:

```
>>> import webbrowser as web
```

`webbrowser` modülünü bu şekilde içe aktardıktan sonra artık bu modül içindeki **fonksiyon ve niteliklere şu şekilde erişebilirsiniz**:

```
>>> web.open("https://www.google.com")
```

Gördüğünüz gibi bu yöntem *daha az kod yazmamızı sağlıyor*.

FROM MODÜL_ADI IMPORT *

Python programlama dilindeki modülleri içe aktarmanın bir başka yolu **`from modül_adı import *`** kalıbını kullanmaktır. Eğer Python'da bir modülü bu kalıpla içe aktarırsanız **o modül içindeki fonksiyon** ve **nitelikleri modül adını belirtmeden de kullanabilirsiniz**. Örneğin:

```
>>> from random import *
```

Bu modülü bu şekilde içe aktardıktan sonra mesela bu modül içindeki **`shuffle()`** fonksiyonunu **hiç modül adı belirtmeden kullanabiliriz**.



shuffle() fonksiyonu bir liste içindeki sayıların rastgele bir şekilde karıştırılmasını sağlar.

Mesela **0** ile **50** arasındaki sayılardan oluşan bir liste olsun elimizde:

```
>>> sayılar = list(range(50))
>>> print(sayılar)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37,
38, 39,
40, 41, 42, 43, 44, 45, 46, 47, 48, 49]
```

Şimdi, bu listenin ögelerini **rastgele** bir şekilde karıştıralım:

```
>>> shuffle(sayılar)
>>> sayılar
[33, 23, 40, 12, 32, 45, 24, 34, 30, 49, 38, 41, 17, 4, 29, 47, 26,
44, 14,
15, 8, 25, 6, 2, 48, 13, 36, 1, 9, 35, 37, 22, 21, 27, 11, 3, 28,
39, 31,
19, 20, 46, 43, 18, 16, 42, 5, 10, 0, 7]
```

Burada **`random`** modülünün **`shuffle()`** fonksiyonunu **modül adı belirtmeden** kullandığımıza dikkat edin. Çünkü biz burada **`random`** modülünü **`from random import *`** şeklinde **içe aktardık**.

Her ne kadar bu yöntem programcıya büyük bir kolaylık sağlıyormuş gibi görünse de *çoğu zaman tercih edilen bir yol değildir*. Çünkü bir modülü bu şekilde içe aktardığımızda o modülün içindeki ögeleri olduğu gibi mevcut çalışma ortamına **boşaltmış** oluyoruz. Bu durumda eğer çalışma ortamında modül içindeki **nitelik** ve **fonksiyonlarla aynı adı taşıyan ögeler varsa bunlar kaybolacaktır**.

Basit bir **örnek** verelim.

Python'da **`open()`** adlı bir **hazır/gömülü** fonksiyon bulunur. Bu fonksiyon yardımıyla bilgisayarımızda bulunan dosyaları açıp bu dosyalar üzerinde **çeşitli işlemler yapabiliyoruz** (*bu fonksiyonu ileride ayrıntılı bir şekilde inceleyeceğiz*).

Bu fonksiyon tıpkı **`print()`**, **`input()`** ve öteki benzer fonksiyonlar gibi **her an kullanımımıza hazırdır**.

Bildiğiniz gibi **webbrowser** modülü içinde de **open()** adlı bir fonksiyon bulunuyor. Bu fonksiyonu kullanarak işletim sistemimizdeki **öntanımlı İnternet tarayıcısı yardımıyla** web sayfalarını açabiliyoruz.

İşte eğer siz **webbrowser** modülünü şu şekilde içe aktarırsanız:

```
>>> from webbrowser import *
```

Bu durumda, modül içinde bulunan **open()** fonksiyonunu, dosyaları açmamızı sağlayan **open()** fonksiyonunun üzerine atmış olursunuz. Böylece dosyaları açan **open()** fonksiyonu **kullanılmaz hâle** gelecektir. **Ama eğer modülü şu şekilde içe aktarırsanız**:

```
>>> import webbrowser
```

veya:

```
>>> import webbrowser as web
```

Bu durumda **open()** fonksiyonları **birbirine karışmaz**:

```
>>> open
<built-in function open>

>>> webbrowser.open
<function open at 0x0000027835341790>

>>> web.open
<function open at 0x00000192979D18B0>
```

Bu durum kendi yazdığınız bir programda, mesela **print** adlı bir değişken tanımlamanıza benzer:

```
>>> print = "yazdır"
>>> print
'yazdır'
>>> print()
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
TypeError: 'str' object is not callable
```

Gördüğünüz gibi **`print()`** fonksiyonunu **kaybettik**. Artık '**`print`**' yazdığımızda **`print()`** fonksiyonu değil, **kendi tanımladığımız değişken çalışıyor**...

Bu tür karışıklıklara yol açmamak için (*birkaç istisnai modül hariç*) modülleri **`from modül_adı import *` şeklinde içe aktarmayı genellikle pek tercih etmiyoruz**.

From modül_adı import metot_veya_fonksiyon

Şimdiye kadar yaptığımız örneklerden de gördüğünüz gibi bir modülün içinde pek çok **nitelik** ve **fonksiyon** bulunabiliyor. Ama çoğu zaman sizin bu nitelik ve fonksiyonların tamamına ihtiyacınız olmaz. Mesela amacınız sadece **`random`** modülü içindeki **`sample()`** fonksiyonunu kullanmaksa tüm modülü olduğu gibi **içe aktarmayı tercih etmeyebilirsiniz**. Bunun yerine modülden sadece **`sample()`** fonksiyonunu alabilirsiniz:

```
>>> from random import sample
```

Hatta eğer isterseniz bu formülü kullanarak **birden fazla ögeyi de aynı anda içe aktarabilirsiniz**:

```
>>> from random import randrange, sample, shuffle
```

Bu şekilde **`random`** modülünün tamamını içe aktarmak yerine sadece **`randrange()`**, **`sample()`** ve **`shuffle()`** fonksiyonlarını **içe aktardık**.

FROM MODÜL_ADI IMPORT METOT_VEYA_FONKSİYON AS BAŞKA_BİR_AD

Python bize, bir modülden tek tek içe aktaracağımız nitelik ve fonksiyonları **farklı bir adla içe aktarma imkânı** da sağlar. Örneğin:

```
>>> from datetime import datetime as dt
```

Böylece **`datetime`** modülünün **`datetime()`** fonksiyonunu **`dt`** adıyla **içe aktarmış** olduk. Artık bu fonksiyonu şöyle kullanabiliriz:

```
>>> şimdi = dt.now()
```

Böylece, Python'da bir modülün **kaç farklı şekilde** içe aktarılabileceğini öğrendik. Gördüğünüz gibi bu yöntemlerin kendine göre **artı** ve **eksileri** var. O anda içinde bulunduğunuz **duruma** ve **ihtiyaçlarına göre** bunlardan herhangi birini seçebilirsiniz.

Şimdi, artık **farklı modül türlerini** incelemeye geçebiliriz. İlk önce **standart kütüphane modüllerinden** başlayalım.

Standart Kütüphane Modülleri

Daha önce de söylediğimiz gibi standart kütüphane modülleri, **Python programlama dilinin bir parçası**dır. Bu modüller, Python programlama dilini geliştiren kişiler tarafından yazılıp dile **monte edilmiştir**. Dolayısıyla bu modülleri kullanabilmek için **herhangi bir yazılım kurmanıza** veya **özel bir işlem yapmanıza gerek yoktur**. Yukarıda gösterdiğimiz içe aktarma yöntemlerinden herhangi birini kullanarak bu modülleri programlarınız içine dâhil edebilir, bu modüller içinde bulunan fonksiyon ve nitelikleri kullanmaya başlayabilirsiniz.

Python'ın standart kütüphanesinde hangi modüllerin olduğunu görmek için *https://docs.python.org/3/library/* **adresini ziyaret edebilirsiniz**.

Aslında Python'ın standart kütüphanesinde yer alan hemen hemen tüm modüller sıradan birer **.py** uzantılı **Python dosyası**dır. Herhangi bir standart kütüphane modülünün kaynak kodlarını görmek isterseniz, kaynak kodları içeren **.py** dosyasının nerede olduğunu şu şekilde bulabilirsiniz:

Diyelim ki **`webbrowser`** modülünün **kaynak kodlarını** görmek istiyorsunuz:

```
>>> import webbrowser
>>> webbrowser.__file__
```

Bu komut size '**webbrowser.py**' adlı dosyanın bilgisayarınızda **hangi konumda** bulunduğunu gösterecektir. Bu dosyayı, herhangi bir metin düzenleyici ile açıp içeriğini inceleyebilirsiniz.

Ancak bazı Python modülleri **C dili** ile yazıldığı için bunların bir **Python kaynak kodu yoktur**. Bu tür modüllerin kaynak kodlarına *https://github.com/python/cpython/tree/master/Python* adresinden ulaşabilirsiniz.

Şimdiye kadar öğrendiğimiz **standart kütüphane** modülleri olan; **`random`**, **`datetime`** ve **`webbrowser`** modüllerinin her üçü de Python'la yazıldığı için bunların **Python kaynak kodlarına** **`__file__`** niteliğini kullanarak rahatlıkla erişebilirsiniz.

Python'ın standart kütüphane modüllerinden biri olup da Python programlama dili ile **yazılmamış** olan en önemli modüllerden biri **sys** adlı modüldür. **Bu modül Python'la değil, C ile yazılmıştır**.

Dilerseniz yeri gelmişken bu önemli modülden biraz bahsedelim.

Tıpkı şimdiye kadar öğrendiğimiz standart kütüphane modüllerinde olduğu gibi **sys** modülü de bize **birtakım faydalı fonksiyon** ve **nitelikler** sunar.

Bu modülün içeriğini nasıl inceleyebileceğinizi artık gayet iyi biliyorsunuz:

```
>>> import sys
>>> dir(sys)
```

Mesela bu listede gördüğünüz **exit()** fonksiyonunu kullanarak programlarınızın **herhangi bir yerinde programdan çıkış yapabilirsiniz**:

```
import sys

veri = input("herhangi bir veri: ")

if not veri:
    print("bir veri girin")
    sys.exit()
else:
    print("girdiğiniz veri:", veri)
```

Burada kullanıcıdan **herhangi bir veri** girmesini istedik. Eğer kullanıcı herhangi bir veri girmeden **ENTER** düğmesine basarsa onu **bir veri girmesi konusunda uyarıp programdan çıkıyoruz**:

```
if not veri:
    print("bir veri girin")
    sys.exit()
```

Aksi durumda ise kullanıcının girdiği veriyi ekrana yazdırıyoruz:

```
else:
    print("girdiğiniz veri:", veri)
```

sys modülü içinde yer alan önemli niteliklerden biri de **argv** niteliğidir. Bu niteliği kullanarak bir Python programının komut satırında **hangi seçeneklerle çalıştırılacağını kontrol edebiliriz**.

Bu niteliğin görevini anlamak için çok basit bir **örnek** verelim:

```
import sys, webbrowser

def site_aç(site):
    webbrowser.open(site)

site_aç(sys.argv[1])
```

Bu kodları '**aç.py**' adlı bir dosyaya **kaydedelim**. Bu programı şu şekilde "**çalıştırmıyoruz"**:

```
python aç.py
```

Eğer programı bu şekilde çalıştırırsak programımız **hata** verecektir. Bu programı mesela şu şekilde çalıştırabiliriz:

```
python aç.py www.google.com
```

Programımızı bu şekilde çalıştırdığımızda *www.google.com* adresi **otomatik** olarak açılacaktır. *www.google.com* yerine **herhangi bir web sitesini yazabiliriz**.

Bu özelliği sağlayan şey, kodlarımız içinde kullandığımız **argv** niteliğidir.

sys modülünün **argv** niteliği, bir Python programının komut satırında **hangi parametrelerle çalıştırıldığını tutar**. Bu durumu daha net görebilmek için şu kodları yazalım:

```
import sys

print(sys.argv)
```

Bu kodları yazdığımız programın nasıl çalıştırıldığına bağlı olarak bu programın **çıktısı** değişecektir.

Mesela bu programı **şöyle çalıştıralım** (programın adı '**deneme.py**' olsun):

```
python deneme.py
```

Bu şekilde şu **çıktıyı** alıyoruz:

```
['deneme.py']
```

Şimdi de programımızı şöyle çalıştıralım:

```
python deneme.py param1
```

Bu defa **çıktı** şöyle oldu:

```
['deneme.py', 'param1']
```

Veya şöyle çalıştıralım:

```
python deneme.py param1 param2
```

Bu kez de **çıktı** şu:

```
['deneme.py', 'param1', 'param2']
```

Gördüğünüz gibi **argv** niteliği, programın adı ile birlikte **programa parametre olarak verilen tüm değerleri bir liste hâlinde tutuyor**. İşte bu özelliği kullanarak şu programı yazdık:

```
import sys, webbrowser

def site_aç(site):
    webbrowser.open(site)

site_aç(sys.argv[1])
```

Burada özellikle **son satıra** dikkat ediyoruz. **sys** modülünün **argv** niteliği aslında **basit bir listeden** ibarettir. Dolayısıyla listelerle yapabildiğiniz her şeyi **argv** ile de yapabilirsiniz.

Mesela yukarıdaki kodlarda **argv** listesini dilimleyerek 1. ögesini aldık ve bu ögeyi **site_aç()** fonksiyonuna **argüman** olarak gönderdik.

argv listesinin neden 1. ögesini dilimlediğimizi görüp **daha iyi anlamak için** kodlarınıza şu eklemeyi yapabilirsiniz:

```
import sys, webbrowser

def site_aç(site):
    webbrowser.open(site)

print(sys.argv[1])

site_aç(sys.argv[1])
```

sys.argv[0], yani **argv** listesinin 0. ögesi **programımızın adını tutuyor**. **sys.argv[1]** ise programımıza verilen **ilk parametrenin adını** gösteriyor. Bize bu programda bu parametre lazım olduğu için de **site_aç()** fonksiyonuna **argüman** olarak **argv** listesinin 1. ögesini gönderdik.

Bu konuyla ilgili bir **örnek** uygulama daha yapalım:

```
import sys, webbrowser

urller = {"google"     : "https://www.google.com/search?q=",
          "bing"       : "https://www.bing.com/search?q=",
          "duckduckgo" : "https://duckduckgo.com/?q=",
          "yandex"     : "https://yandex.com/search/?text="}

def ara(site, sorgu):
    adres = urller.get(site)
    webbrowser.open(adres+sorgu)

ara(sys.argv[1], sys.argv[2])
```

Bu programı da mesela şöyle kullanıyoruz (programımızın adı '**ara.py**' olsun):

```
python ara.py bing java
```

Bu komutla Bing arama motorunda '**java**' kelimesini aramış olduk.

Veya:

```
python ara.py duckduckgo python
```

Bu komutla ise **DuckDuckGo** arama motorunda '**python**' kelimesini aradık.

Gördüğünüz gibi **sys** modülünün **argv** niteliği sayesinde kullanıcıdan komut satırı aracılığıyla **istediğimiz verileri** alabiliyoruz. İsterseniz bu işlevi **input()** fonksiyonu yardımıyla da yerine getirebilirsiniz. İhtiyacınıza ve duruma uygun olan yöntemi seçmek size kalmış.

Üçüncü Şahıs Modülleri

Python programlama dilinde Python geliştiricilerinin yazıp dile entegre ettiği standart kütüphane modüllerinin dışında bir de **üçüncü şahısların yazdığı modüller** bulunur. Bunlar Python geliştiricisi olmayan kişilerce **çeşitli ihtiyaçlara cevap verebilmek için** yazılıp **internet üzerinden dağıtılan yazılımlardır**.

Biz geçen bölümlerde, bu tür modüllere örnek olarak **cx_Freeze** adlı modülü vermiştik. Bu modül *https://anthony-tuininga.github.io/cx_Freeze/* adresinde Anthony Tuininga adlı bir programcı tarafından dağıtılan bir **Python modülüdür**.

Bu modülü kullanabilmek için bunu öncelikle bilgisayarımıza **kurmamız** gerekir.

Çoğu **üçüncü şahıs modülünü** *https://pypi.org/* adresinde bulabilirsiniz.

Üçüncü şahıs modüllerini genellikle şu komutla kurabilirsiniz:

```
pip install modül_adı
```

Bu arada bu komut bir **Python komutu değildir**. Dolayısıyla bu komutu etkileşimli kabukta değil, sistem komut satırında veriyoruz.

Üçüncü şahıs modüllerini bilgisayarımıza kurduktan sonra, **aynı standart kütüphane modülleri gibi kullanıyoruz**.

Bu modüller de tıpkı standart kütüphane modülleri gibi aslında **.py** uzantılı birtakım **Python dosyalarından ibarettir**. Bu modüllerin de kaynak dosyalarına *https://pypi.org/* adresinden veya **ilgili modül yazarının web sitesinden ulaşabilirsiniz**. Hatta bu modülleri bilgisayarınıza kurduktan sonra tıpkı standart kütüphane modüllerinde olduğu gibi ilgili modülün **__file__** niteliğini sorgulayarak o modülün bilgisayarınızdaki yerini tespit edebilirsiniz:

```
>>> import cx_Freeze
>>> cx_Freeze.__file__
```

Bu komut **cx_Freeze** modülünün (*eğer kuruluysa*) bilgisayarınızda hangi konumda yer aldığını gösterecektir.

Kullanici Tanimli Modüller

Bu noktaya kadar **standart kütüphane modüllerinden** ve **üçüncü şahıs modüllerinden** söz ettik. Gelin, şimdi de kendi modüllerimizi nasıl yazacağımızı öğrenelim.

Bu noktada '**sözlük.py**' adlı boş bir Python dosyası oluşturalım ve içine şunları yazalım:

```
sözlük = {"kitap"      : "book",
          "bilgisayar" : "computer",
          "programlama": "programming"}

def ara(sözcük):
    hata = f"{sözcük} kelimesi sözlükte yok!"
    return sözlük.get(sözcük, hata)
```

Tebrikler! Böylece ilk modülünüzü tanımlamış oldunuz...

Hatırlarsanız önceki sayfalarda modüllerin aslında **.py** uzantılı *sıradan birer Python programı olduğunu söylemiştik*. İşte siz de yukarıdaki kodları '**sözlük.py**' adlı bir dosyaya kaydettiğinizde **sözlük** adlı bir **modül** oluşturmuş oluyorsunuz.

Şimdi, '**sözlük.py**' dosyasının bulunduğu konumda bir **komut satırı oturumu başlatın** ve **Python'ın etkileşimli kabuğunu çalıştırın**.

Daha sonra şu komutu verin:

```
>>> import sözlük
```

Gördüğünüz gibi tıpkı başka modüllerde olduğu gibi kendi modüllerimizi de **import** komutuyla **içe aktarabiliyoruz**.

Şimdi bu modülün **içeriğini sorgulayalım**:

```
>>> dir(sözlük)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__', 'ara', 'sözlük']
```

Burada bizi ilgilendirenler **ara** ve **sözlük** ögeleri. Bu ögelere nasıl erişebileceğinizi biliyorsunuz:

```
>>> sözlük.ara("kitap")
'book'

>>> sözlük.sözlük
{'kitap': 'book', 'bilgisayar': 'computer', 'programlama':
'programming'}
```

Hatta **sözlük** adlı bu modülün konumunu da sorgulayabilirsiniz:

```
>>> sözlük.__file__
```

Bu komut, '**sözlük.py**' dosyanızı nereye koyduysanız o konumu işaret edecektir.

Bu bölümün başında bahsettiğimiz **farklı içe aktarma yöntemlerini** kendi modüllerinize de uygulayabilirsiniz:

```
import sözlük
import sözlük as szlk
from sözlük import *
from sözlük import ara, sözlük
```

Zaten aslında kendi modüllerinizi yazdığınızda mantık olarak bir **üçüncü şahıs modülü** yazmış oluyorsunuz.

IF __NAME__ == '__MAIN__'

Bu noktaya kadar Python'daki modüller hakkında epey bilgi edindik. Şimdi yine, modüllerin son derece önemli bir yönünden söz edeceğiz. **Dikkatlice takip edin**.

Biz Python'da modülleri (*yani programları*) **iki farklı şekilde çalıştırma imkânına** sahibiz.

İlk olarak Python programlarımızı çalıştırmak için komut satırında **şu komutu verebiliriz**:

```
python modül_adı.py
```

Eğer programımız kullanıcıdan **birtakım parametreler bekliyorsa** programımızı şu şekilde de çalıştırabiliriz:

```
python modül_adı.py param1 param2 param3
```

Programın adından sonra gelen parametreleri bir **liste hâlinde** alabilmek için **sys** modülünün **argv** niteliğini kullanabileceğimizi biliyorsunuz.

Python modüllerini çalıştırmanın **ikinci yöntemi** ise o modülü **doğrudan çalıştırmak yerine** `import` komutuyla başka bir programın (*veya ortamın*) **içine aktarmaktır**:

```
import modül_adı
```

Bu şekilde modülü doğrudan çalıştırmak yerine, bu modül içindeki nitelik ve fonksiyonları başka programlar içinde kullanmak üzere **o programların hizmetine sunmuş oluyoruz**.

Örnek olarak yukarıda yazdığımız **arama programına** bakalım:

```
import sys, webbrowser

urller = {"google"     : "https://www.google.com/search?q=",
          "bing"       : "https://www.bing.com/search?q=",
          "duckduckgo" : "https://duckduckgo.com/?q=",
          "yandex"     : "https://yandex.com/search/?text="}

def ara(site, sorgu):
    adres = urller.get(site)
    webbrowser.open(adres+sorgu)

ara(sys.argv[1], sys.argv[2])
```

Bu programı normalde, mesela şu şekilde çalıştırabiliyoruz:

```
python ara.py yandex java
```

Burada biz '**`ara.py`**' adlı modülü **bağımsız bir program gibi** kullanıyoruz.

Bir de '**sözlük.py**' adlı şu programa bakalım:

```
sözlük = {"kitap"      : "book",
          "bilgisayar" : "computer",
          "programlama": "programming"}

def ara(sözcük):
    hata = "{} kelimesi sözlükte yok!"
    return sözlük.get(sözcük, hata.format(sözcük))
```

Bunu da **etkileşimli kabukta içe aktaralım**:

```
>>> import sözlük
>>> sözlük.ara("kitap")
'book'
```

Bu şekilde ise '**sözlük.py**' adlı modülü **ayrı** ve **bağımsız** bir program gibi kullanmak yerine, bu modül içindeki **nitelik** ve **fonksiyonları başka programların emrine sunuyoruz**.

Mesela bu modülü başka bir Python programının içine aktarabiliriz. **Dikkatlice bakın**:

Şimdi bu '**sözlük.py**' adlı modülün bulunduğu klasör içinde '**test.py**' adlı **başka bir dosya daha oluşturalım** ve **bu dosyanın içine şu kodları yazalım**:

```
import sözlük

kelime = sözlük.ara("bilgisayar")
print(kelime)
```

Şimdi, şu komutu vererek '**test.py**' programını çalıştıralım:

```
python test.py
```

Gördüğünüz gibi '**sözlük.py**' dosyasını **sözlük** adlı bir modül olarak '**test.py**' programının içine aktardık ve bu modülün **ara()** adlı fonksiyonunu '**test.py**' programı içinde kullandık. Bu sayede **ara()** gibi bir fonksiyona ihtiyacımız olduğunda bu fonksiyonu **tekrar yazmak zorunda kalmadık**.

Esasında, standart kütüphane modüllerinin ve üçüncü şahıs modüllerinin **sağladığı fayda da tam olarak budur**. Yani mesela yazdığımız bir programda, kullanıcının bilgisayarındaki öntanımlı İnternet tarayıcıyı açmamız gerektiğinde, yapmamız işlemleri *kendimiz icat etmek zorunda kalmıyoruz*. Bunun yerine basitçe **webbrowser** modülünün **open()** fonksiyonunu **içe aktarıp kullanıyoruz**.

Aynı şekilde **rastgele sayılar** üretmemiz gerektiğinde bu işlemi yapmamızı sağlayacak yöntemi kendimiz icat etmek zorunda kalmıyoruz. Basitçe **random** modülünün **randint()** fonksiyonunu kullanıyoruz.

Peki ya biz, bir programın aynı anda hem **doğrudan komut satırından çalıştırılabilmesini** hem de **başka programlar içine modül olarak aktarılabilmesini** istersek ne yapacağız?

Gelin şimdi, bu isteğimizi nasıl yerine getirebileceğimizi görelim.

Python'da modüllerin **__name__** adlı bir niteliği bulunur. Bir modülü içe aktardığımızda bu **__name__** niteliğinin değeri modülün kendi adı olacaktır.

Örnek olarak daha önce yazdığımız '**sözlük.py'** programına bakalım:

```
sözlük = {"kitap"      : "book",
          "bilgisayar" : "computer",
          "programlama": "programming"}

def ara(sözcük):
    hata = "{} kelimesi sözlükte yok!"
    return sözlük.get(sözcük, hata.format(sözcük))
```

Bu programı, **etkileşimli kabuğa bir modül olarak aktaralım**:

```
>>> import sözlük
>>> sözlük.__name__
'sözlük'
```

Gördüğünüz gibi **__name__** niteliğinin değeri, **modülün adı**.

Aynı durum, **diğer modül türleri için de geçerlidir**:

```
>>> import random
>>> random.__name__
'random'

>>> import cx_Freeze
>>> cx_Freeze.__name__
'cx_Freeze'
```

Ama eğer bir modülü doğrudan bir program olarak çalıştırırsak bu **__name__** niteliğinin değeri, **'__main__' karakter dizisi olacaktır**. Bunu **doğrulamak için** '**sözlük.py**' **dosyasını açın** ve **en son satıra şu kodu ekleyin**:

```
print(__name__)
```

Yani kodlarınız tam olarak şöyle görünsün:

```
sözlük = {"kitap"      : "book",
          "bilgisayar" : "computer",
          "programlama": "programming"}

def ara(sözcük):
    hata = "{} kelimesi sözlükte yok!"
    return sözlük.get(sözcük, hata.format(sözcük))

print(__name__)
```

Şimdi bu modülü, **etkileşimli kabukta içe aktarın**:

```
>>> import sözlük
```

Bu komutun **çıktısı** şu olacak:

```
sözlük
```

Şimdi de bu modülü doğrudan bir **program olarak çalıştırın**:

```
python sözlük.py
```

Bu defa şu **çıktıyı** alacaksınız:

```
__main__
```

Gördüğünüz gibi **`sözlük.py`** programını **çalıştırma şekline göre `__name__`** niteliği **farklı çıktılar veriyor**. İşte Python'ın bu özelliğinden yararlanarak *programlarınızın farklı durumlarda farklı işlemler yapmasını sağlayabilirsiniz*.

Bu durumu daha net bir şekilde anlayabilmek için şu programı **tekrar önümüze alalım**:

```
import sys, webbrowser

urller = {"google"     : "https://www.google.com/search?q=",
          "bing"       : "https://www.bing.com/search?q=",
          "duckduckgo" : "https://duckduckgo.com/?q=",
          "yandex"     : "https://yandex.com/search/?text="}

def ara(site, sorgu):
    adres = urller.get(site)
    webbrowser.open(adres+sorgu)

ara(sys.argv[1], sys.argv[2])
```

Dediğimiz gibi bu programı, mesela şu şekilde çalıştırabileceğimizi biliyorsunuz:

```
python ara.py yandex java
```

Eğer bu programı bu hâliyle bir modül olarak **içe aktarmaya** çalışırsak **hata mesajı** alırız:

```
>>> import ara
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\Users\hazan\Downloads\ara.py", line 12, in <module>
    ara(sys.argv[1], sys.argv[2])
IndexError: list index out of range
```

Bu hatanın sebebi, **`ara.py` programının ihtiyacı olan parametreleri belirtmemiş olmamız**. Çünkü biz bu programı, modül olarak içe aktarılacak şekilde tasarlamadık.

Bu kodların hem **bağımsız bir program** hem de bir **modül** olarak kullanılabilmesi için programda şu değişikliği yapıyoruz:

```
import sys, webbrowser

urller = {"google"     : "https://www.google.com/search?q=",
          "bing"       : "https://www.bing.com/search?q=",
          "duckduckgo" : "https://duckduckgo.com/?q=",
          "yandex"     : "https://yandex.com/search/?text="}

def ara(site, sorgu):
    adres = urller.get(site)
    webbrowser.open(adres+sorgu)

if __name__ == "__main__":
    ara(sys.argv[1], sys.argv[2])
```

Burada **son iki satıra dikkat edin**:

```
if __name__ == "__main__":
    ara(sys.argv[1], sys.argv[2])
```

Bu kodlarla, şu Türkçe cümleyi **Pythonca'ya çevirmiş olduk**:

```
Eğer __name__ niteliğinin değeri '__main__' ise (yani program
doğrudan komut
satırı üzerinden çalıştırılmışsa) 'ara(sys.argv[1], sys.argv[2])'
komutunu
işlet.
```

Bu ufak kod parçası sayesinde yukarıdaki programı **her iki şekilde de** çalıştırma imkânına sahip oluyoruz. **Yani hem şöyle**:

```
python ara.py duckduckgo programlama
```

Hem de şöyle:

```
>>> from ara import ara
>>> ara("duckduckgo", "programlama")
```

Burada önce ara modülünün **`ara()`** adlı fonksiyonunu **içe aktardık**. Daha sonra da bu fonksiyonu gerekli parametrelerle birlikte kullanarak **bir arama işlemi** gerçekleştirdik.

Bununla ilgili bir **örnek** daha verelim:

```
import sys

sözlük = {"kitap"      : "book",
          "bilgisayar" : "computer",
          "programlama": "programming"}

def bul(sözcük):
    hata = "{} kelimesi sözlükte yok!"
    return sözlük.get(sözcük, hata.format(sözcük))
```

```
if __name__ == '__main__':
    if len(sys.argv) < 2:
        print("kullanım: python sözlük.py sözlükte_aradığınız_
kelime")
        print("örnek: python sözlük.py kitap")

    else:
        sorgu = sys.argv[1]
        print(sözlük.get(sorgu, f"{sorgu} kelimesi sözlükte
bulunamadı"))
```

Burada öncelikle **sys** modülünü **içe aktardık**:

```
import sys
```

Çünkü programımızda bu modülün **argv** niteliğini kullanacağız.

Daha sonra, **Türkçe kelimelerle İngilizce karşılıklarını içeren bir sözlük tanımladık**:

```
sözlük = {"kitap"      : "book",
          "bilgisayar" : "computer",
          "programlama": "programming"}
```

Ardından bu sözlük içinde daha **rahat** ve **pratik** bir şekilde arama işlemleri gerçekleştirebilmek için **bul** adlı bir fonksiyon tanımladık:

```
def bul(sözcük):
    hata = "{} kelimesi sözlükte yok!"
    return sözlük.get(sözcük, hata.format(sözcük))
```

Bu fonksiyon **sözcük** adlı tek bir **argüman** alıyor. Fonksiyon içinde sözlüklerin **get()** metodunu kullanarak **sözlükte kelime araması gerçekleştiriyoruz**. Eğer aranan kelime, sözlük içinde geçmiyorsa **kullanıcıyı bu konuda bilgilendiriyoruz**.

Daha sonra ise şu satırları görüyoruz:

```
if __name__ == '__main__':
    if len(sys.argv) < 2:
        print("kullanım: python sözlük.py sözlükte_aradığınız_
kelime")
        print("örnek: python sözlük.py kitap")

    else:
        sorgu = sys.argv[1]
        print(sözlük.get(sorgu, f"{sorgu} kelimesi sözlükte
bulunamadı"))
```

Buna göre eğer **`__name__` niteliğinin değeri `'__main__'` karakter dizisi** ise (*yani kodlarımız bağımsız bir program gibi çalıştırılmışsa*) **aşağıdaki kodları işletiyoruz**:

```
if len(sys.argv) < 2:
    print("kullanım: python sözlük.py sözlükte_aradığınız_kelime")
    print("örnek: python sözlük.py kitap")

else:
    sorgu = sys.argv[1]
    print(sözlük.get(sorgu, f"{sorgu} kelimesi sözlükte
bulunamadı"))
```

Burada ise öncelikle **`sys.argv`** listesinin **uzunluğunu denetliyoruz**. Eğer **`sys.argv`** listesinin uzunluğu 2'den **azsa** yani listede sadece *programın adı geçiyorsa* dolayısıyla program herhangi bir parametre verilmeden çalıştırılmışsa, **kullanıcıyı programı nasıl kullanması gerektiği konusunda yönlendiriyoruz**:

```
print("kullanım: python sözlük.py sözlükte_aradığınız_kelime")
print("örnek: python sözlük.py kitap")
```

Aksi hâlde (*yani kullanıcı programı gerekli parametrelerle birlikte çalıştırmışsa*) **`sys.argv`** listesinin 1. ögesini (*yani kullanıcının girdiği parametre değerini*) **sorgu adlı bir değişkene atıyoruz**:

```
sorgu = sys.argv[1]
```

Daha sonra da bu değeri **`get()`** metodu yardımıyla **sözlük içinde arıyoruz**.

Gördüğünüz gibi **`if __name__ == '__main__'`** satırı sayesinde olanaklarımız bir hayli genişliyor. Siz de bu satırı kodlarınızda yeri geldiğinde bol bol kullanacaksınız.

SYS.PATH

Yaptığımız örneklerden gördüğünüz gibi hem **standart kütüphanede** yer alan modülleri hem de **üçüncü şahıs** modüllerini **herhangi bir konumdan rahatlıkla içe aktarabiliyoruz**. *Yani bunları programlarımıza dâhil etmek için mutlaka bu modüllerin bulunduğu klasörün içinde olmamız gerekmiyor*.

Ancak **kendi yazdığımız modülleri kullanabilmek için bunların bulunduğu klasör içinde olmamız gerekiyor**. *Acaba diğer modül türleri ile kendi yazdığımız modüller arasındaki bu farka yol açan şey nedir?*

Bu bölümde **sys** adlı bir modülden söz etmiştik. Bu modül içinde **`argv`** niteliği dışında **`path`** adlı bir nitelik de bulunur:

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

Bu komutu verdiğinizde karşınıza bir **dizin listesi** çıkacak. İşte yazdığınız modülleri herhangi bir konumdan çağırabilmek için modülünüzün **`sys.path`** listesinde görünen dizinlerden birinin içinde olması gerekiyor.

Biz bir modülü **`import modül_adı`** komutuyla içe aktarmaya çalıştığımızda Python öncelikle bu komutu verdiğimiz **dizin konumuna** bakar. Eğer aradığımız modülü **burada bulursa bu modülü içe aktarır**. Ama eğer Python aradığımız modülü mevcut çalışma dizini içinde bulamazsa bu **`sys.path`** listesinde görünen dizinlerin içine bakar.

sys.path aslında alelade bir listedir:

```
>>> type(sys.path)
<class 'list'>
```

Dolayısıyla liste veri tipi ile yaptığınız her şeyi **sys.path** ile de yapabilirsiniz. Mesela bu listede bulunmayan bir dizin konumunu gerektiğinde listelerin **append()** veya **insert()** **metodunu kullanarak** **sys.path** **listesine ekleyebilirsiniz**:

```
>>> sys.path.append("herhangi/bir/dizin/konumu")
```

veya:

```
>>> sys.path.insert(0, "herhangi/bir/dizin/konumu")
```

İşte kendi yazdığınız modüllerin de tıpkı **standart kütüphane** modülleri gibi **her konumdan içe aktarılabilmesini** istiyorsanız ya modülünüzü **sys.path** çıktısında görünen dizinlerden birine **kopyalayacaksınız** ya da **modülünüzün bulunduğu dizini** **sys.path** **listesine ekleyeceksiniz**.

Böylece Python'daki modüller hakkında söylememiz gereken her şeyi söylemiş olduk. Dilerseniz şimdi, bu modül konusunu burada noktalayıp yine önemli bir başka konuyu incelemeye geçelim.

DOSYA İŞLEMLERİ

22

Bu bölümde Python programlama dilini kullanarak dosyaları nasıl yöneteceğimizi, yani nasıl **yeni bir dosya oluşturacağımızı**, **bir dosyaya nasıl bir şeyler yazabileceğimizi** ve **buna benzer işlemleri** öğreneceğiz. Burada Python'ı kullanarak sistemimizde yeni dosyalar oluşturmanın yanı sıra, **var olan dosyaları da herhangi bir aracı program kullanmadan doğrudan Python ile açacağız**.

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

Dosya Açmak

Bu bölümde ilk olarak **bilgisayarımızda yeni bir dosyanın nasıl oluşturulacağını** göreceğiz. Anlaması daha kolay olsun diye Python'la ilk dosyamızı **mevcut çalışma dizini** altında oluşturacağız.

Python'da dosya açmak (*veya oluşturmak*) için **`open()`** adlı bir fonksiyonu kullanıyoruz. Mesela o anda içinde bulunduğumuz dizin altında '**`deneme.txt`**' adlı bir **dosya oluşturalım**:

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

Bu komutla mevcut çalışma dizinimiz altında '**`deneme.txt`**' adlı bir **metin dosyası** oluşturmuş olduk.

Ancak burada ufak bir problem var. Biz dosyamızı oluştururken bu dosyanın **hangi dil kodlamasına göre oluşturulacağını belirtmedik**. Dosyalar üzerinde işlem yaparken dil kodlamasının belirtilmesi özellikle **İngilizce olmayan karakterlerin düzgün görüntülenmesi açısından çok önemlidir.** Eğer biz kendimiz bir dil kodlaması belirtmezsek; bu durumda kullandığımız işletim sistemindeki öntanımlı dil kodlaması neyse o **geçerli** olacaktır.

Üzerinde çalıştığınız bir dosyanın öntanımlı dil kodlamasını şu şekilde kontrol edebiliriz:

```
>>> dosya.encoding
```

Eğer yukarıdaki dosyayı **Türkçe Windows** üzerinde oluşturduysanız bu komut muhtemelen '**`cp1254`**' **çıktısı**, **İngilizce Windows** üzerinde oluşturduysanız da '**`cp1252`**' **çıktısı** verecektir. Yani biz burada herhangi bir **dil kodlaması belirtmediğimiz için** dosyamız öntanımlı olarak '**`cp1254`**' **veya** '**`cp1252`**' adlı dil kodlaması ile oluşturuldu.

`dosya.encoding` komutunu GNU/Linux'ta Ubuntu üzerinde verdiğimizde ise dosyamızın '**`utf-8`**' **dil kodlaması ile oluşturulduğunu görürüz**.

Bir dil kodlaması belirtmediğiniz durumlarda **işletim sisteminizin hangi dil kodlamasını seçeceğini** **`locale`** adlı bir **standart kütüphane modülünün** **`getpreferredencoding()`** adlı fonksiyonu **yardımıyla** da öğrenebilirsiniz:

```
>>> import locale
>>> locale.getpreferredencoding()
```

Ancak özellikle Türkçe karakterlerin gösterimi konusunda **sorun yaşamamak için** dil kodlaması belirleme işini **işletim sistemimize bırakmamalı, dosyalarımızın dil kodlamasını kendimiz açıkça belirtmeliyiz**.

Örneğin eğer **işletim sisteminizin öntanımlı dil kodlaması** '**`cp1254`**' ise **`encoding`** **parametresini belirlemeyi unutsanız bile Türkçe karakterler dosyaya düzgün yazılacaktır**. Ancak eğer sisteminizdeki tanımlı dil kodlaması mesela '**cp1252**' ise Türkçe karakterlerin gösteriminde problem yaşarsınız. Dolayısıyla **işimizi şansa bırakmamak için** **`encoding`** parametresini kendimiz belirlemeli ve genel olarak dosyalarımızda '**`utf-8`**' dil kodlamasını **tercih**

etmeliyiz. Zira '**utf-8**', pek çok farklı dile ait karakterleri düzgün bir şekilde görüntüleyebilen, **son derece yaygın kabul görmüş bir dil kodlamasıdır**. O yüzden **open()** fonksiyonunu kullanarak bir metin dosyası oluşturacağımız zaman (*aksi için geçerli bir sebebimiz yoksa*) mutlaka '**utf-8**' dil kodlamasını **tercih etmeliyiz**:

```
>>> dosya = open("deneme.txt", "x", encoding="utf-8")
```

Şimdi, bu komutu biraz inceleyelim. **open()** fonksiyonunun ne olduğu belli. **Bir dosyayı açmaya yarıyor**. Parantez içindeki "**deneme.txt**" karakter dizisinin de ne olduğu açık. Bu parametre, **oluşturmak istediğimiz dosyanın adını gösteriyor**. Tabii oluşturacağımız dosyanın adını **tırnak içine almayı unutmuyoruz**. **encoding** parametresinin de ne işe yaradığını biraz önce açıkladık. Bu parametreye vereceğimiz değer sayesinde özellikle Türkçe karakterleri dosyaya düzgün bir şekilde yazabiliyoruz. *Peki, acaba bu x ne oluyor?*

Python'da sıfırdan yeni bir dosya oluştururken **x** adı verilen bir '**kip**' (*mode*) kullanıyoruz. Bu kip, eğer bilgisayarımızda önceden '**deneme.txt**' adlı bir dosya varsa o dosyanın **silinmesini önleyecektir**.

Eğer amacımız '**deneme.txt**' adlı dosyayı oluşturmak istediğimiz dizin altında zaten '**deneme.txt**' adlı bir dosya olup olmadığına bakmaksızın '**deneme.txt**' adlı bir dosya oluşturmaksa **w** kipini kullanabiliriz:

```
>>> dosya = open("deneme.txt", "w", encoding="utf-8")
```

Fakat '**deneme.txt**' dosyasını oluşturmak istediğimiz dizinde zaten '**deneme.txt**' adlı bir dosya varsa **yukarıda komut herhangi bir soru sormadan varolan dosyanın üzerine yazacaktır**. Çünkü **w** kipi, aynı dosyadan bilgisayarınızda zaten var olup olmadığına bakmaksızın, aynı adda yepyeni bir dosya oluşturacak, *bunu yaparken de eski dosyanın içeriğini silecektir*.

x ve **w** kiplerinin dışında bir de **r** ve **a** kipleri bulunur.

r kipi, bir dosyayı veri eklemek için değil, **sadece okumak için açabilmemizi sağlar**. *Böylece bu dosya üzerinde herhangi bir değişiklik yapılamaz*.

Bir de **a** kipi vardır dedik. **a** kipi bilgisayarımızda zaten **var olan bir dosyaya yeni veri eklemek için kullanılır**. Dolayısıyla dosya içindeki eski verileri koruyup bu dosyaya yeni veriler eklemek istiyorsak **a** kipini kullanmamız gerekecek.

Biraz önce:

```
>>> dosya = open("deneme.txt", "x", encoding="utf-8")
```

komutuyla mevcut çalışma dizini altında '**deneme.txt**' adında, '**utf-8**' dil kodlamasına sahip yeni bir dosya oluşturduk. Burada **open()** fonksiyonu ile açtığımız dosyayı **dosya** adlı bir değişkene atadık.

Şimdi başka bir **örnek** verelim:

```
>>> dosya = open("deneme.txt", "r", encoding="utf-8")
```

Dikkat ederseniz burada **x** kipi yerine **r** kipini kullandık. Biraz önce de açıkladığımız gibi bu kip ilgili dosyanın yazılmak üzere değil **okunmak üzere** açılacağını gösteriyor. Yani eğer biz bir dosyayı bu kipte açarsak dosya içine **herhangi bir veri girişi yapma imkânımız olmaz**. Ayrıca bu kip yardımıyla yeni bir dosya da oluşturamayız. Bu kip bize var olan bir dosyayı **yalnızca okuma kipinde açma imkânı verir**. Yani mesela:

```
>>> dosya = open("deneme.txt", "r", encoding="utf-8")
```

komutunu verdiğimizde eğer bilgisayarda '**deneme.txt**' adlı bir dosya yoksa bu adla **yeni bir dosya oluşturulmayacak**, bunun yerine Python bize bir **hata mesajı** gösterecektir. **Eğer bu komutu şöyle yazarsak**:

```
>>> dosya = open("deneme.txt", "x", encoding="utf-8")
```

Bu durumda eğer bilgisayarda '**deneme.txt**' adlı bir dosya yoksa bu adla **yeni bir dosya oluşturulacaktır**. Ama eğer '**deneme.txt**' adlı bir dosya zaten varsa bu komut da bize bir **hata mesajı** gösterecektir.

Peki şu komutu verirsek ne olur?

```
>>> dosya = open("deneme.txt", "w", encoding="utf-8")
```

Bu komut ise bilgisayarımızda '**deneme.txt**' adlı bir dosya olup olmadığına bakmaksızın **bu adla yeni bir dosya oluşturacaktır**. Eğer bilgisayarımızda '**deneme.txt**' adlı bir dosya varsa bu komut **o dosyanın içeriğini silecektir**. **Dolayısıyla dikkatli kullanılması gereken bir komuttur bu**.

Python'da dosya işlemleri yaparken **içeriği dolu bir dosyayı açıp bu dosyaya eklemeler yapmamız da gerekebilir**. İşte böyle durumlar için **a** adlı bir kipten yararlanacağız.

Bu kipi şöyle kullanıyoruz:

```
>>> dosya = open("deneme.txt", "a", encoding="utf-8")
```

Python'da bir dosyayı **a** kipinde açtığımız zaman, o dosyanın içine **yeni veriler ekleyebiliriz**. Ayrıca **a** kipinin, **r** kipinin aksine, bize yeni dosya oluşturma imkânı da verdiğini aklımızın bir köşesine not edelim.

Bu kipin **w** kipinden farkı, **w** kipinin bir dosyayı **sıfırdan oluşturması**, yani o dosyanın **içeriğini silmesidir**. **a** kipi ise dosya içeriğini silmez. Eğer açmaya çalıştığımız dosya **sistemde yoksa o adla yeni bir dosya oluşturur**. Eğer **açmaya çalıştığımız dosya sistemde varsa o dosyayı açarak içine eklemeler yapabilmemize izin verir**. Yani **w** **yıkıcı**, **a** ise **yapıcı** bir dosya erişim kipidir... Bu kipin **r** kipinden farkı ise **a** kipiyle **açılan dosyalar üzerinde okuma işlemi yapılamayacak olmasıdır**.

Eğer yazdığımız kod içinde yukarıdaki dört kipten (`x`, `w`, `r`, `a`) hiçbirini kullanmazsak Python, **öntanımlı** olarak `r` kipini kullanacaktır:

```
dosya = open("deneme.txt", encoding="utf-8")
```

Bu dosya için bir erişim kipi belirtmediğimiz için '`deneme.txt`' adlı dosya okuma kipinde (`r`) açıldı. Tabii eğer ilgili dizinde bu adda bir **dosya yoksa** bu kod bize bir **hata mesajı** verecektir. Dolayısıyla `r` kipinin bu özelliğinden ötürü, bilgisayarımızda yeni bir dosya oluşturmak istiyorsak **kip belirtmemiz**, yani `x`, `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", "x",
encoding="utf-8")
```

komutu '`/home/kullanıcı_adı/`' dizini altında, '`deneme.txt`' adlı bir dosya oluşturacaktır.

Yalnız burada **küçük bir uyarı** yapalım. Yazdığımız kodlarda yol adı belirtirken kullandığımız **yatık çizgilerin yönüne dikkat etmemiz gerekir**. En emin yol, yukarıda yaptığımız gibi dosya yolunu **sağa yatık bölü işaretiyle ayırmaktır**:

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

Sağa yatık bölü bütün işletim sistemlerinde sorunsuz çalışır. Ama sola yatık bölü problem oluşturabilir:

```
>>> f = open("C:\Users\fozgul\Desktop\filanca.txt", "a",
encoding="utf-8")

Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
IOError: [Errno 22] invalid mode ('a') or filename: 'C:\\Users\
x0cozgul\\Desktop\x0cilanca.txt'
```

Burada sorun, **Python'ın \ işaretini bir kaçış dizisi olarak algılaması**. Hâlbuki biz burada bu işareti yol ayracı olarak kullanmak istedik. Eğer sağa yatık bölü kullanmak isterseniz \ işaretini **çiftlemeniz** gerekir:

```
>>> f = open("C:\\Users\\fozgul\\Desktop\\filanca.txt", "a",
encoding="utf-8")
```

Veya bu tür durumlarda **r** adlı özel bir **kaçış dizisinden** yararlanabilirsiniz:

```
>>> f = open(r"C:\Users\fozgul\Desktop\filanca.txt", "a",
encoding="utf-8")
```

Böylece, dosya yolunu oluşturan karakter dizisi içindeki **tüm kaçış dizilerini işlevsiz hâle getirerek Python'ın hata vermesini engelleyebilirsiniz**.

Dosyaya Yazmak

Şu ana kadar öğrendiğimiz şey, *Python'da dosya açmak ve oluşturmaktan ibarettir*. Ancak **henüz açtığımız bir dosyaya nasıl müdahale edeceğimizi veya nasıl veri girişi yapabileceğimizi bilmiyoruz.** İşte birazdan, bilgisayarımızda hâlihazırda var olan veya bizim sonradan oluşturduğumuz bir dosyaya **nasıl veri girişi yapabileceğimizi göreceğiz**. Mesela '**python.txt**' adlı bir dosya oluşturarak içine "**Guido Van Rossum**" yazalım. Ama bu kez komut satırında değil de **bir metin dosyası** üzerinde yapalım bu işlemi. Hemen **boş bir sayfa açıp içine şunları yazıyoruz**:

```
dosya = open("python.txt", "x", encoding="utf-8")
dosya.write("Guido Van Rossum")
dosya.close()
```

Burada ilk olarak mevcut çalışma dizini altında adı '**python.txt**' ve dil kodlaması '**utf-8**' olan **yeni bir dosya oluşturduk**. Bu dosyayı **x** kipinde açtığımız için eğer **ilgili dizin altında zaten bu adla bir dosya varsa** bu komut bize bir **hata mesajı** gösterecektir.

Daha sonra **write()** adlı bir metot yardımıyla '**python.txt**' dosyasının içine "**Guido Van Rossum**" yazdık. En son da **close()** adlı başka bir metottan yararlanarak **dosyayı kapattık**.

Gelin isterseniz, şimdi bu bölümün başında **`encoding`** parametresi ile ilgili **söylediklerimizi test edelim**.

Eğer bir dosyayı açarken **`encoding`** parametresini açıkça belirtmezsek işletim sistemimizin **dil kodlamasını** kendisinin belirleyeceğini, eğer sistemimizdeki öntanımlı dil kodlaması dosyaya yazmak istediğimiz metinle **uyumsuzsa dosyaya yazmada sorun yaşayacağımızı söylemiştik**.

Bunu test etmek için şöyle bir kod yazalım:

```
f = open("deneme.txt", "w")
f.write("şçöğü")
f.close()
```

Eğer bu kodlar herhangi bir **hata mesajı vermediyse** sisteminizdeki öntanımlı dil kodlaması **Türkçe karakterlerle uyumlu demektir**. Ama eğer bu kodlar **hata verdiyse** sisteminizdeki öntanımlı dil kodlaması **Türkçe karakterleri tanımıyor demektir**.

Bir de şu kodları deneyelim:

```
f = open("deneme.txt", "w")
f.write("Привет")
f.close()
```

Burada da tamamen, kullandığınız işletim sisteminin insafına kalmış durumdasınız. *Eğer sisteminizdeki öntanımlı dil kodlaması Rusça karakterleri tanımıyorsa bu kodlar hata verir.* İşte bu tür durumların ortaya çıkmasını **önlemek** için her zaman **dil kodlamasını kendimiz belirleyeceğiz** ve **aksi için geçerli bir sebebimiz yoksa** '**`utf-8`**' **dil kodlamasını tercih edeceğiz**:

```
f = open("deneme.txt", "w", encoding="utf-8")
f.write("şçöğü\n")
f.write("Привет")
f.close()
```

'**`utf-8`**' dil kodlaması hem Türkçe hem de Rusça karakterleri (*ve başka pek çok dildeki karakterleri*) tanıyabilir.

Dosyaların Metotları

Python'da tıpkı *karakter dizisi, liste, demet, sözlük* ve *küme* veri tiplerinin metotları olduğu gibi **dosyaların da kendine has birtakım metotları bulunur**. Yine **`dir()`** metodunu kullanarak dosyaların metotlarını inceleyebilirsiniz. Mesela yukarıdaki dosyanın **hangi metotlara sahip olduğunu görmek için** şu komutu kullanabilirsiniz:

```
print(dir(dosya))
```

Biz bu bölümde şimdiye kadar dosyaların **`encoding`** niteliğinden ve ayrıca **`write()`** ve **`close()`** metotlarından söz ettik.

Buna göre **`encoding`** niteliği bir dosyanın dil kodlamasını gösteriyor:

```
print(dosya.encoding)
```

`write()` metodu **bir dosyaya veri eklememizi sağlıyor**:

```
dosya.write("falanca")
```

`close()` metodu ise **açtığımız dosyayı kapatmamızı sağlıyor**:

```
dosya.close()
```

Şimdi de şöyle bir şey yapalım:

Biraz önce oluşturduğumuz ve içine "**`Guido Van Rossum`**" yazdığımız dosyamıza **ikinci bir satır ekleyelim**:

```
dosya = open("python.txt", "a", encoding="utf-8")
dosya.write("\nMonty Python")
dosya.close()
```

Gördüğünüz gibi bu kez dosyamızı **`a`** kipiyle açtık. Eğer bu dosyayı **`w`** kipiyle açarsak **eski dosyayı silmiş oluruz**. O yüzden Python'la programlama yaparken bu tür şeylere çok dikkat etmek gerekir.

Dosyamızı **a** kipiyle açtıktan sonra **write()** metodu yardımıyla "**Monty Python**" satırını **eski dosyaya ekledik**. Burada **\n adlı kaçış dizisinden yararlandığımıza da dikkat edin**. Eğer bunu kullanmazsak; eklemek istediğimiz satır bir önceki satırın **hemen arkasına** getirilecektir. Bütün bunlardan sonra da **close()** fonksiyonu yardımıyla **dosyamızı kapattık**.

Bir de şu örneğe bakalım:

```
dosya = open("şiir.txt", "x", encoding="utf-8")
dosya.write("Bütün güneşler batmadan,\nBi türkü daha \
söyleyeyim bu yerde\n\t\t\t\t --Orhan Veli--")
dosya.close()
```

Gördüğünüz gibi "**şiir.txt**" adlı yeni bir metin dosyası oluşturduk.

Bu dosyanın **içine yazılan verilere** ve **burada kaçış dizilerini nasıl kullandığımıza** çok dikkat edin. İkinci mısrayı bir **alt satıra almak için \n** kaçış dizisini kullandık. Daha sonra "**Orhan Veli**" satırını **sayfanın sağına doğru kaydırmak için \t** kaçış dizisinden yararlandık. Bu örnekte **\n** ve **\t** kaçış dizilerini yan yana kullandık. Böylece aynı cümleyi hem **alt satıra almış** hem de **sağa doğru kaydırmış** olduk.

Ayrıca birkaç tane **\t** kaçış dizisini yan yana kullanarak **cümleyi sayfanın istediğimiz noktasına getirdik**.

Yukarıdaki **write()** metodu dışında, çok yaygın kullanılmayan bir de **writelines()** metodu vardır. **Bu metot birden fazla satırı bir kerede dosyaya işlemek için kullanılır**. Şöyle ki:

```
dosya = open("şiir2.txt", "x", encoding="utf-8")
dosya.writelines(["Bilmezler yalnız yaşamayanlar\n",
"Nasıl korku verir sessizlik insana\n",
"İnsan nasıl konuşur kendisiyle\n",
"Nasıl koşar aynalara bir cana hasret\n",
"Bilmezler...\n"])
dosya.close()
```

Burada parantez içindeki **köşeli parantezlere** dikkat edin. Aslında oluşturduğumuz şey bir **liste**.

Dolayısıyla bu fonksiyon bir listenin içeriğini **doğrudan bir dosyaya yazdırmak için faydalı olabilir**. Aynı kodu `write()` metoduyla yazmaya kalkışırsanız alacağınız şey bir **hata mesajı** olacaktır. Eğer bir liste içinde yer alan ögeleri `write()` metodunu kullanarak dosyaya yazdırmak isterseniz `for` döngüsünden yararlanabilirsiniz:

```
>>> liste = [“elma”, “armut”, “kalem”]
>>> f = open(“falanca.txt”, “x”, encoding=”utf-8”)
>>> for i in liste:
...     f.write(i+”\n”)
...
>>> f.close()
```

Gelin, isterseniz bu noktada **küçük bir uygulama** yapalım.

Hatırlarsanız önceki derslerimizden birinde şöyle bir program yazmıştık:

```
import platform

print(“işletim sistemi adı:”, platform.system())
print(“işletim sistemi sürümü: “, platform.release())
print(“işlemci mimarisi”, platform.machine())
print(“işlemci:”, platform.processor())
print(“python sürümü:”, platform.python_version())
```

Bu program, **kullandığımız işletim sistemi** ve **Python sürümü hakkında bize bazı bilgiler veriyordu**.

Ancak o zaman henüz dosyalar konusunu öğrenmediğimiz için **bu verileri nasıl kaydedeceğimizi bilmiyorduk**. Artık dosya işlemlerini nasıl yapacağımızı bildiğimiz için **bu verileri aynı zamanda bir dosyaya da kaydedebiliriz**.

Dikkatlice bakın:

```
import platform

veriler = [
            ("işletim sistemi adı"    , platform.system()),
            ("işletim sistemi sürümü" , platform.release()),
            ("işlemci mimarisi"       , platform.machine()),
            ("işlemci"                , platform.processor()),
            ("python sürümü"          , platform.python_version())
          ]

for a, b in veriler:
    print(f"{a}: {b}")

kayıt = input("\nBu verileri kaydetmek ister misiniz? [e/h] ")

if kayıt in ("e", "evet"):
    f = open("sistem_verileri.txt", "w", encoding="utf-8")
    for a, b in veriler:
        f.write(f"{a}: {b}\n")
    print(f"Veriler {f.name} adlı dosyaya kaydedildi!")

else:
    print("Kayıt yapılmadan çıkılıyor...")
```

Burada öncelikle sistemle ilgili verileri, **iki ögeli demetlerden oluşan bir liste içinde topladık**:

```
veriler = [
            ("işletim sistemi adı"    , platform.system()),
            ("işletim sistemi sürümü" , platform.release()),
            ("işlemci mimarisi"       , platform.machine()),
            ("işlemci"                , platform.processor()),
            ("python sürümü"          , platform.python_version())
          ]
```

`veriler` listesi içindeki demetlerin her biri **`platform`** modülü içindeki **ilgili metotlardan** ve **bu metotların anlamını gösteren etiketlerden** oluşuyor.

Bu etiket ve metot adı çiftlerini şu şekilde düzenleyerek kullanıcılara gösteriyoruz:

```
for a, b in veriler:
    print(f"{a}: {b}")
```

Burada bir **for** döngüsü içinde her bir **etikete** **a**, her bir **metot** adına da **b** dedik. Daha sonra bu **a** ve **b** değişkenlerini kullanıcılarımıza uygun bir şekilde gösterdik.

Tüm verileri gösterdikten sonra kullanıcıya **şu soruyu soruyoruz**:

```
kayıt = input("\nBu verileri kaydetmek ister misiniz? [e/h] ")
```

Eğer kullanıcı bu soruya '**e**' **veya** '**evet**' cevabı verirse:

```
if kayıt in ("e", "evet"):
```

Yani **input()** fonksiyonundan gelen veri ("**e**", "**evet**") demetinin içinde geçiyorsa '**sistem_verileri.txt**' adında, "**utf-8**" dil kodlamasına sahip bir dosya oluşturuyoruz:

```
f = open("sistem_verileri.txt", "w", encoding="utf-8")
```

Ardından da **yukarıdaki iki ögeli demetlerin her birini dosyaya yazıyoruz**:

```
for a, b in veriler:
    f.write(f"{a}: {b}\n")
```

Daha sonra, **verilerin dosyaya kaydedildiği konusunda kullanıcıları bilgilendiriyoruz**:

```
print(f"Veriler {f.name} adlı dosyaya kaydedildi!")
```

Burada dosyaların **name** adlı niteliğini kullandığımıza dikkat edin. Python'da bir dosyanın **name** niteliği **o dosyanın adını** gösterir.

Eğer **`input()`** fonksiyonundan gelen veri (“**e**”, “**evet**”) demeti içinde geçmiyorsa yani kullanıcı “**e**” **veya** “**evet**” dışında bir cevap vermişse “**`Kayıt yapılmadan çıkılıyor...`**” **mesajını gösteriyoruz**:

```
else:
    print("Kayıt yapılmadan çıkılıyor...")
```

Alternatif olarak yukarıdaki programı şöyle de yazabilirsiniz:

```
import platform

veriler = {"işletim sistemi adı"    : platform.system(),
           "işletim sistemi sürümü" : platform.release(),
           "işlemci mimarisi"       : platform.machine(),
           "işlemci"                : platform.processor(),
           "python sürümü"          : platform.python_version()}

for k, v in veriler.items():
    print(f"{k}: {v}")

kayıt = input("\nBu verileri kaydetmek ister misiniz? [e/h] ")

if kayıt in ("e", "evet"):
    f = open("sistem_verileri.txt", "w", encoding="utf-8")
    for k, v in veriler.items():
        f.write(f"{k}: {v}\n")
    print(f"Veriler {f.name} adlı dosyaya kaydedildi!")

else:
    print("Kayıt yapılmadan çıkılıyor...")
```

Burada **`veriler`** değişkenini iki ögeli demetlerden oluşan bir liste yerine bir **sözlük** olarak tanımladığımıza dikkat edin. **Şunu asla aklımızdan çıkarmamalıyız**: Bir programı yazmanın sadece **tek bir doğru yöntemi yoktur**. Her programcı, karşılaştığı bir sorunu kendine has yöntemlerle çözecektir. Benim bulduğum çözüm sizin aklınıza gelmeyebileceği gibi sizin bulduğunuz çözüm de benim aklıma gelmeyebilir. Karşılaştığınız problemleri çözmek için **siz de kendinize has bir yöntem bulup onu takip edebilirsiniz**.

İsterseniz şimdi bu kodları bir kenara bırakıp Python'da dosya işlemlerinin nasıl yapıldığını incelemeye devam edelim.

Dosyayı Okumak

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

Şu **örneğe** bir bakalım:

```
yeni = open("şiir.txt", "w", encoding="utf-8")

metin = """\
Sular çekilmeye başladı köklerden...
Isınmaz mı acaba ellerimde kan?
Ah, ne olur! Bütün güneşler batmadan
Bi türkü daha söyleyeyim bu yerde..."""

yeni.write(metin)
yeni.close()

yeni = open("şiir.txt", encoding="utf-8")
print(yeni.read())
```

`yeni.read()` satırına kadar olan kısmı zaten biliyoruz. Burada kullandığımız **`read()`** metodu **`yeni`** **adlı değişkenin içeriğini okumamızı sağlıyor**. **`yeni`** adlı değişkenin değeri '**`şiir.txt`**' adlı bir dosya olduğu için bu metot '**`şiir.txt`**' adlı dosyanın içeriğini bize gösterecektir.

Bu arada yukarıdaki kodlarda mesela **`encoding`** parametresini **dosyayı yazma kipinde** açarken *belirtip*, **okuma** kipinde açarken *belirtmeyerek* ortaya çıkması muhtemel etkiyi incelemenizi tavsiye ederim.

Python'da **`read()`** dışında bir de **`readlines()`** adlı bir metot bulunur. **Eğer yukarıdaki komutu**:

```
yeni.readlines()
```

şeklinde verecek olursak **çıktının bir liste olduğunu görürüz**.

Bir de eğer bu, **`readlines()`** metodunun sonundaki '**s**' harfini atıp:

```
>>> yeni.readline()
```

şeklinde bir kod yazarsak **dosya içeriğinin yalnızca ilk satırı okunacaktır**.

Python'ın **`readline()`** metodunu değerlendirirken kullandığı ölçüt şudur:

"Dosyanın başından itibaren ilk **\n** ifadesini gördüğün yere kadar oku".

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

```
yeni = open("şiir.txt", encoding="utf-8")
for satır in yeni:
    print(satır)
```

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'ın dosya içinde gezdirdiği bir imleç var.*) **dosyada ikinci satırın başına kaydırıyor**. Eğer bu komutu bir daha verirsek **üçüncü satır okunacaktır**. S**on bir kez daha bu komutu verirsek artık dosyanın sonuna ulaşıldığı için ekrana hiçbir şey yazılmayacaktır**. Böyle bir durumda dosyayı başa sarmak için şu fonksiyonu kullanıyoruz. (*Dosyamızın adının dosya olduğunu varsayıyoruz*):

```
>>> dosya.seek(0)
```

Böylece imleci tekrar dosyanın **en başına almış olduk**. Tabii siz isterseniz bu imleci **farklı noktalara da taşıyabilirsiniz**. Mesela:

```
>>> dosya.seek(10)
```

komutu, imleci **10. karakterin** başına getirecektir (*Saymaya her zamanki gibi 0'dan başlıyoruz.*) Bu **seek()** metodu, 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.

Python'da dosya işlemleri yaparken bilmemiz gereken en önemli noktalardan biri de şudur: Python ancak karakter dizilerini dosyaya yazdırabilir. Sayıları yazdıramaz. Eğer biz, sayıları da yazdırmak istiyorsak önce bu sayıları karakter dizisine çevirmemiz gerekir.

Bir **örnek** verelim:

```
>>> sayı = 50
>>> dosya = open("deneme.txt", "x", encoding="utf-8")
>>> dosya.write(sayı)
```

Bu kodlar bize şu **çıktıyı** verir:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: write() argument must be str, not int
```

Gördüğünüz gibi Python bize bir **hata mesajı** gösterdi. Çünkü **`sayı`** değişkeninin değeri **bir sayı**. Hâlbuki **karakter dizisi olması gerekiyor**. Bu meseleyi çözmek için komutumuzu şu şekilde veriyoruz.

En baştan alırsak:

```
>>> sayı = 50
>>> dosya = open("deneme.txt", "x", encoding="utf-8")
>>> dosya.write(str(sayı))
```

Burada:

```
>>> str(sayı)
```

komutuyla, bir sayı olan **`sayı`** değişkenini karakter dizisine çevirdik.

Dosya Silmek

Peki, oluşturduğumuz bu dosyaları nasıl sileceğiz? Python'da herhangi bir şekilde oluşturduğumuz bir dosyayı silmenin **en kestirme yolu `os` adlı bir standart kütüphane modülü içinde yer alan `remove()` fonksiyonunu kullanmaktır**.

`os` da tıpkı daha önce öğrendiğimiz öteki modüller gibi kullanılır. Dolayısıyla bu modülü kullanabilmek için öncelikle bunu **`import`** komutuyla **içe aktarmamız gerekir**:

```
>>> import os
```

Artık bu modül içindeki **`remove()`** fonksiyonunu kullanarak **herhangi bir dosyayı silebiliriz**:

```
>>> os.remove("dosya/yolu")
```

Mesela masaüstündeki '**`deneme.txt`**' dosyasını şöyle siliyoruz:

```
>>> import os
>>> os.remove("/home/kullanıcı_adı/Desktop/deneme.txt")
```

Eğer masaüstü zaten sizin **mevcut çalışma dizininiz ise** bu işlem çok daha basittir:

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

Dosyaya Rastgele Satır Eklemek

Şimdiye kadar hep dosya sonuna satır ekledik. *Peki, ya bir dosyanın ortasına bir yere satır eklemek istersek ne yapacağız?* Şimdi, diyelim ki elimizde '**deneme.txt**' adlı bir dosya var ve **içinde şunlar yazılı**:

```
Birinci Satır
İkinci Satır
Üçüncü Satır
Dördüncü Satır
Beşinci Satır
```

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

```
>>> kaynak = open("deneme.txt", encoding="utf-8")
```

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", "x", encoding="utf-8")
```

Bu dosyayı ise **yazma modunda açtık**. Çünkü kaynak dosyadan okuduğumuz verileri buraya yazdıracağız. Şimdi de yapacağımız **okuma işlemini tanımlayalım**:

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

Böylece kaynak dosya üzerinde yapacağımız satır okuma işlemini de tanımlamış olduk.

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

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

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

Artık işimiz bittiğine göre **hedef** ve **kaynak** dosyaları **kapatalım**:

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

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

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

Tabii bu son işlemleri yapmadan önce **os** modülünü **içe aktarmayı unutmuyoruz**. Ayrıca burada **os** modülünün **rename()** adlı bir başka fonksiyonundan daha yararlandığımıza dikkat edin.

Bu fonksiyon bize bir dosyanın **adını değiştirme imkânı** verir. Bu fonksiyon iki parametre alır:

```
os.rename(dosyanın_eski_adı, dosyanın_yeni_adı)
```

Esasında yukarıdaki işlemleri yapmanın daha **pratik** bir yolu bulunur. Diyelim ki elimizde, içeriği şu olan '**liste.txt**' adlı bir dosya var:

```
ahmet
mehmet
selim
cevat
sait
```

Biz bu dosyada **mehmet** ile **selim** arasına **fatih** satırını **eklemek istiyoruz**. Dilerseniz bu işlemi nasıl yapacağımızı gösteren kodları verelim:

```
oku = open("liste.txt", encoding="utf-8")

yeni_satır = 'fatih'

içerik = oku.readlines()
içerik.insert(2, yeni_satır+"\n")

oku.close()

yaz = open("liste.txt", "w", encoding="utf-8")
yaz.writelines(içerik)

yaz.close()
```

Şimdi de bu kodları **tek tek inceleyelim**:

```
oku = open("liste.txt", encoding="utf-8")
```

satırı yardımıyla '**liste.txt**' adlı dosyayı okumak üzere açıyoruz. Burada herhangi bir kip belirtmediğiniz için Python'ın bu dosyayı **otomatik** olarak okuma kipinde açtığına dikkat edin.

Daha sonra dosyaya eklemek istediğimiz satırı bir **değişkene atıyoruz**.

Hemen ardından **`readlines()`** adlı metodu kullanarak '**`liste.txt`**' adlı dosyanın **tüm içeriğini bir liste içinde topluyoruz**. *Böylece dosya içeriğini yönetmek çok daha kolay olacak*. Bildiğiniz gibi **`readlines()`** metodu bize **çıktı** olarak bir **liste** veriyor.

Bir sonraki satırda dosyaya eklemek istediğimiz metni, **`readlines()`** metodu ile oluşturduğumuz listenin **3. sırasına yerleştiriyoruz**. Burada listelerin **`insert()`** metodunu kullandığımıza dikkat edin.

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

Şimdi yapmamız gereken şey, gerekli bilgileri **dosyaya yazmak** olacak. O yüzden bu defa '**`liste.txt`**' adlı dosyayı **yazma kipinde** açıyoruz:

```
yaz = open("liste.txt", "w", encoding="utf-8").
```

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

Son olarak dosyayla işimizi bitirdiğimize göre **dosyamızı kapatmayı unutmuyoruz**.

Dosyadan Rastgele Satır Silmek

Bazen üzerinde çalıştığımız bir dosyanın **herhangi bir satırını silmemiz de gerekebilir**. Bunun için yine *bir önceki bölümde anlattığımız yöntemi kullanabiliriz*. Dilerseniz yine yukarıda bahsettiğimiz **`.txt`** dosyasını **örnek** alalım:

```
ahmet
mehmet
selim
cevat
sait
```

Şimdi diyelim ki biz bu dosyanın '**selim**' satırını silmek istiyoruz. Bu işlemi şu kodlarla hâlledebiliriz:

```
oku = open("liste.txt", encoding="utf-8")
icerik = oku.readlines()
del icerik[2]

oku.close()

yaz = open("liste.txt", "w", encoding="utf-8")
yaz.writelines(icerik)

yaz.close()
```

Burada da tıpkı bir önceki bölümde olduğu gibi öncelikle **readlines()** metodunu kullanarak **dosya içeriğini bir listeye gönderdik**. Daha sonra bu listede silmek istediğimiz satıra karşılık gelen ögenin sıra numarasını kullanarak **del** deyimi yardımıyla **ilgili ögeyi listeden kaldırdık**. Son olarak da elimizdeki **değiştirilmiş listeyi bir dosyaya yazdırdık**.

Gördüğünüz gibi *bir dosyaya veri girmek veya dosyadan veri çıkarmak son derecek kolay bir işlem*. Yapmamız gereken tek şey **dosya içeriğini bir listeye alıp bu liste üzerinde gerekli değişiklikleri yapmak**. Daha sonra da bu değiştirilmiş listeyi dosyaya yazdırarak amacımıza ulaşabiliyoruz.

Dosyaları Otomatik Kapatma

Python'da bir dosyayı açıp bu dosya üzerinde **gerekli işlemleri yaptıktan sonra bu dosyayı açık bırakmamak büyük önem taşır**. *Kaynak israfına yol açmamak için* üzerinde işlem yaptığımız bütün dosyaları, **işimiz bittikten sonra mutlaka kapatmalıyız**. Çünkü bir dosya açıldığında işletim sistemi, sistem kaynaklarının bir kısmını **bu dosyaya ayırır**. Eğer dosyayı açık bırakırsak sistem kaynaklarını gereksiz yere meşgul etmiş oluruz. Ancak farklı sebeplerden, **dosyalar açıldıktan sonra kapanmayabilir**. Örneğin açtığınız dosyayı kapatmayı unutmuş olabilirsiniz. Yani programınızın hiçbir yerinde **`close()`** metodunu kullanmamışsınızdır. Bunun dışında, programınızdaki **bir hata da dosyaların kapanmasını engelleyebilir**. Örneğin bir dosya açıldıktan sonra programda beklenmeyen bir hata gerçekleşirse programınız asla **`close()`** satırına ulaşamayabilir. Bu durumda da **açılan dosya kapanmadan öylece bekler**.

Bu tür durumlara karşı, dosyalarımızı **otomatik** olarak kapatabilmemiz için Python bize oldukça **pratik** bir yöntem sunar. **Dikkatlice bakın**:

```
with open("dosyaadı", "r") as dosya:
    print(dosya.read())
```

Dosyalarımızı bu şekilde açıp üzerlerinde işlemlerimizi yaptığımızda, Python dosyayı bizim için **kendisi kapatacaktır**. Bu şekilde bizim ayrıca bir **`close()`** satırı yazmamıza gerek yok. **`with`** deyimini kullanmamız sayesinde, dosya açıldıktan sonra arada bir hata oluşsa bile Python, dosyayı sağsalim kapatıp **sistem kaynaklarının israf edilmesini önleyecektir**.

Mesela yukarıda **`with`** kullanmadan yazdığımız, **txt** dosyasına satır ekleyen kodları **`with`** ile şöyle yazabiliriz:

```
yeni_satır = 'fatih'

with open("liste.txt", encoding="utf-8") as oku:
    içerik = oku.readlines()
    içerik.insert(3, yeni_satır+"\n")

with open("liste.txt", "w", encoding="utf-8") as yaz:
    yaz.writelines(içerik)
```

Gördüğünüz gibi bu sistem, dosyaların işi bittiğinde **otomatik** olarak kapanmasını sağlamanın yanı sıra kodlarımızı da daha derli toplu gösteriyor.

HATA YAKALAMA 23

Hatalar, **programcılık deneyiminizin bir parçasıdır**. Programcılık maceranız boyunca hatalarla sürekli karşılaşacaksınız. Ancak bizim burada kastettiğimiz, programı yazarken **sizin yapacağınız hatalar değil**. Kastettiğimiz şey, **programınızı çalıştıran kullanıcıların sebep olduğu** ve **programınızın çökmesine yol açan kusurlar**.

Dilerseniz ne demek istediğimizi anlatmak için şöyle bir **örnek** verelim:

```
sayı = int(input("Lütfen bir sayı girin: "))
print("Girdiğiniz sayı", sayı)
print("Teşekkürler. Hoşçakalın!")
```

Gayet basit bir program bu, değil mi? Bu program görünüşte herhangi bir kusur taşımıyor. Eğer kullanıcımız burada, **bizim istediğimiz gibi bir sayı girerse bu programda hiçbir sorun çıkmaz**. Programımız görevini başarıyla yerine getirir. *Peki, ama ya kullanıcı sayı yerine harf girerse ne olacak?* Mesela kullanıcı yukarıdaki programı çalıştırıp '**a**' harfine basarsa ne olur?

Böyle bir durumda programımız şöyle bir **hata mesajı** verir:

```
Traceback (most recent call last):
  File "arayuz.py", line 1, in <module>
    sayı = int(input("Lütfen bir sayı girin: "))
ValueError: invalid literal for int() with base 10: 'a'
```

Gördüğünüz gibi ortaya çıkan **hata nedeniyle programın son satırı ekrana basılamadı**. Yani, programımız hata yüzünden çöktüğü için **işlem yarıda kaldı**. Burada böyle bir hata almamızın nedeni, **kullanıcıdan sayı beklediğimiz hâlde kullanıcının harf girmesi**. Biz yukarıdaki programda **`int()`** fonksiyonunu kullanarak kullanıcıdan aldığımız **karakter dizilerini sayıya dönüştürmeye** çalışıyoruz. Mesela kullanıcı **23** değerini girerse bu karakter dizisi **rahatlıkla sayıya dönüştürülebilir**. Dilerseniz bunu **etkileşimli kabukta test edelim**:

```
>>> int("23")
23
```

Gördüğünüz gibi **23** adlı karakter dizisi rahatlıkla sayıya dönüştürülebiliyor.

Peki ya şu?

```
>>> int("a")

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

İşte bizim programımızın yapmaya çalıştığı şey de tam olarak budur. Yani programımız kullanıcıdan aldığı '**a**' harfini sayıya dönüştürmeye çalışıyor ve tabii ki **başarısız oluyor**.

Bir kod yazarken kodları işletecek kullanıcıların **hangi değerleri girebileceğini her zaman göz önünde bulundurmanız gerektiğini asla unutmayın**. Program çalıştırılırken kullanıcıların ne gibi hatalar yapabileceklerini kestirmeye çalışın. Çünkü kullanıcılar her zaman sizin istediğiniz gibi davranmayabilir. Siz, programın yazarı olarak kodlarınızı tanıdığınız için programınızı nasıl kullanmanız gerektiğini biliyorsunuzdur, *ama son kullanıcı böyle değildir*.

Yukarıdaki kodun çok uzun bir programın parçası olduğunu düşünürsek **kullanıcının yanlış veri girişi koskoca bir programın çökmesine** veya **durmasına yol açabilir**. Bu tür durumlarda Python gerekli **hata mesajını** ekrana yazdırarak kullanıcıyı **uyaracaktır** ama tabii ki Python'ın sunduğu karmaşık hata mesajlarını kullanıcının anlamasını bekleyemeyiz.

Böylesi durumlar için Python'da **`try... except`** ifadeleri kullanılır. İşte biz de bu bölümde bu tür ifadelerin ne zaman ve nasıl kullanılacağını anlamaya çalışacağız.

TRY... EXCEPT...

Dediğimiz gibi **Python'da hata yakalamak için `try... except`** bloklarından yararlanılır. Gelin, isterseniz bunun nasıl kullanılacağına bakalım. *Giriş bölümünde verdiğimiz örneği hatırlıyorsunuz*. O **örnek**, kullanıcının sayı yerine **harf** girmesi durumunda **hata** veriyordu. Şimdi, **hata mesajına tekrar bakalım**:

```
Traceback (most recent call last):
    File "deneme.py", line 4, in <module>
sayı = int(input("Lütfen bir sayı girin: "))
ValueError: invalid literal for int() with base 10: 'a'
```

Burada önemli kısım **`ValueError`'**dır. Hatayı yakalarken bu ifade işimize yarayacak.

Python'da, **hata yakalama işlemleri iki adımdan oluşur**. Önce ne tür bir hata ortaya çıkabileceği tespit edilir. Daha sonra, bu hata meydana geldiğinde ne yapılacağına karar verilir. Yukarıdaki örnekte biz **birinci adımı** uyguladık. Yani **ne tür bir hata ortaya çıkabileceğini tespit ettik**. Buna göre kullanıcı sayı yerine **harf girerse `ValueError`** tipinde bir **hata** meydana geliyormuş...

Şimdi de böyle bir hata ortaya çıkarsa ne yapacağımıza karar vermemiz gerekiyor. Mesela öyle bir durumda kullanıcıya, "**`Lütfen harf değil, sayı girin!`**" gibi bir **uyarı mesajı** gösterebiliriz. Dilerseniz bu dediklerimizi somutlaştıralım. **Kodlarımızın ilk hâli şöyleydi**:

```
sayı = int(input("Lütfen bir sayı girin: "))
print("Girdiğiniz sayı", sayı)
print("Teşekkürler. Hoşçakalın!")
```

Bu tarz bir kodlamanın, **`ValueError`** tipinde bir **hata** vereceğini biliyoruz. **Şimdi şuna bakın**:

```
try:
    sayı = int(input("Lütfen bir sayı girin: "))
    print("Girdiğiniz sayı", sayı)
except ValueError:
    print("Lütfen harf değil, sayı girin!")

print("Teşekkürler. Hoşçakalın!")
```

Burada, **hata vereceğini bildiğimiz kodları** bir **`try`** bloğu içine aldık. Ardından bir **`except`** bloğu açarak ne tür bir hata beklediğimizi belirttik. Buna göre beklediğimiz **hata türü `ValueError`**. Son olarak da **hata durumunda kullanıcıya göstereceğimiz mesajı yazdık**. Artık kullanıcılarımız sayı yerine harfe basarsa programımız çökmeyecek, **aksine çalışmaya devam edecektir**. Dikkat ederseniz **`print("Teşekkürler. Hoşçakalın!")`** satırı **her koşulda ekrana basılıyor**. Yani kullanıcı **doğru olarak sayı da girse**; **yanlışlıkla sayı yerine harf de girse** programımız yapması gereken işlemleri tamamlayıp yoluna devam edebiliyor.

Konuyu daha iyi anlayabilmek için bir **örnek** daha verelim. Şimdi şöyle bir program yazdığımızı düşünün:

```
ilk = int(input("Bölme işlemi için ilk sayıyı girin: "))
ikinci = int(input("Şimdi de ikinci sayıyı girin: "))
sonuç = ilk / ikinci
print(sonuç)
```

Eğer burada kullanıcı ikinci sayıya **`0`** cevabı verirse programımız şöyle bir **hata mesajı verip çökecektir**:

```
Traceback (most recent call last):
  File "arayuz.py", line 3, in <module>
    sonuç = ilk / ikinci
ZeroDivisionError: division by zero
```

Çünkü bir sayı 0'a bölünemez...

Böyle bir durumda hata alacağımızı bildiğimize göre ilk adım olarak **ne tür bir hata mesajı alabileceğimizi tespit ediyoruz**. Buna göre alacağımız hatanın türü **`ZeroDivisionError`**. Şimdi de böyle bir hata durumunda ne yapacağımıza karar vermemiz gerekiyor. İsterseniz yine kullanıcıya bir **uyarı mesajı** gösterelim.

Kodlarımızı yazıyoruz:

```
ilk = int(input("Bölme işlemi için ilk sayıyı girin: "))
ikinci = int(input("Şimdi de ikinci sayıyı girin: "))

try:
    sonuç = ilk / ikinci
    print(sonuç)

except ZeroDivisionError:
    print("Lütfen sayıyı 0'a bölmeye çalışmayın!")
```

Bir önceki örnekte de yaptığımız gibi burada da **hata vereceğini bildiğimiz kodları `try` bloğu içine aldık**. Böyle bir durumda alınacak hatanın **`ZeroDivisionError`** olduğunu da bildiğimiz için **`except`** bloğunu da buna göre yazdık. Son olarak da kullanıcıya **gösterilecek uyarıyı belirledik**. Böylece programımız hata karşısında çökmeden yoluna devam edebildi.

Burada önemli bir problem dikkatinizi çekmiş olmalı. Biz yukarıdaki kodlarda, **kullanıcının bir sayıyı 0'a bölmesi ihtimaline karşı `ZeroDivisionError`** hatasını yakaladık. **Ama ya kullanıcı sayı yerine harf girerse ne olacak? `ZeroDivisionError`** ile birlikte **`ValueError`** hatasını da yakalamamız gerekiyor.

Eğer yukarıdaki kodları çalıştıran bir kullanıcı sayı yerine harf girerse şöyle bir **hatayla** karşılaşır:

```
Traceback (most recent call last):
    File "deneme.py", line 4, in <module>
ilk = int(input("Bölme işlemi için ilk sayıyı girin: "))
ValueError: invalid literal for int() with base 10: 'a'
```

Buradan anladığımıza göre **hata veren satır şu**: **`ilk = int(input("Bölme işlemi için ilk sayıyı girin: "))`** Dolayısıyla bu satırı da **`try`** bloğu içine almamız gerekiyor.

Şu kodları dikkatlice inceleyin:

```
try:
    ilk = int(input("Bölme işlemi için ilk sayıyı girin: "))
    ikinci = int(input("Şimdi de ikinci sayıyı girin: "))
    sonuç = ilk / ikinci
    print(sonuç)

except ZeroDivisionError:
    print("Lütfen sayıyı 0'a bölmeye çalışmayın!")

except ValueError:
    print("Lütfen harf değil, sayı girin!")
```

Gördüğünüz gibi **hata vereceğini bildiğimiz kodların hepsini** bir **`try`** bloğu içine aldık. Ardından da verilecek hataları birer **`except`** bloğu içinde teker teker yakaladık. **Eğer her iki hata için de aynı mesajı göstermek isterseniz şöyle bir şey yazabilirsiniz**:

```
try:
    ilk = int(input("Bölme işlemi için ilk sayıyı girin: "))
    ikinci = int(input("Şimdi de ikinci sayıyı girin: "))
    sonuç = ilk / ikinci
    print(sonuç)

except (ZeroDivisionError, ValueError):
    print("Girdiğiniz veri hatalı!")
```

Hata türlerini **nasıl grupladığımıza dikkat edin**. Hataları birbirinden **virgülle ayırıp parantez içine alıyoruz**. Böylece programımız her iki hata türü için de **aynı uyarıyı** gösteriyor kullanıcıya.

Bu bölüm, hata yakalama konusuna iyi bir giriş yapmamızı sağladı. İleride çok daha farklı hata türleriyle karşılaştığınızda bu konuyu çok daha net bir şekilde içinize sindirmiş olacaksınız.

İsterseniz şimdi, daha önce fonksiyonlar bölümünde ilk kez bahsettiğimiz ve bu konuyla bağlantılı olduğunu düşündüğümüz **pass** deyimine tekrar bakalım.

pass Deyimi

Fonksiyonlar bölümünde '**pass**' kelimesinin Türkçede **geçmek**, **aşmak** gibi anlamlara geldiğini söylemiştik. O zaman bu **pass** deyimini anlatmak için şöyle bir **örnek** vermiştik:

```
def deneme():
    liste = []
    while True:
        a = input("Giriniz: ")
        if a == "0":
            pass
        elif a == "iptal":
            break
        else:
            liste.append(a)
            print(liste)

deneme()
```

Burada kullanıcı eğer '**0**' tuşuna basarsa **pass** deyiminin etkisiyle hiçbir işlem yapılmadan sonraki yönergelere geçiliyor.

Gördüğünüz gibi Python'da bu deyimi **herhangi bir işlem yapmadan geçeceğimiz durumlar için kullanıyoruz**. Yani bu deyimi Pyhon'da "**görmezden gel**", "**hiçbir şey yapma**" anlamında kullanacağız.

Mesela bir hata ile karşılaşan programınızın hiçbir şey yapmadan yoluna devam etmesini istediğiniz zamanlarda da bu deyimi kullanabilirsiniz:

```
try:
    ilk = int(input("Bölme işlemi için ilk sayıyı girin: "))
    ikinci = int(input("Şimdi de ikinci sayıyı girin: "))
    sonuç = float(ilk) / ikinci
    print(sonuç)

except (ZeroDivisionError, ValueError):
    pass
```

Böylece programınız **`ZeroDivisionError`** veya **`ValueError`** ile karşılaştığında sessizce yoluna devam edecek ve kullanıcımız programda ters giden bir şeyler olduğunu dahi anlamayacaktır. **Yazdığınız programlarda bunun iyi bir şey mi yoksa kötü bir şey mi olduğuna programcı olarak sizin karar vermeniz gerekiyor**. Eğer bir hatanın kullanıcıya gösterilmesinin gerekmediğini düşünüyorsanız yukarıdaki kodları kullanın. Ama eğer verilen hata **önemli** bir hataysa ve kullanıcının bu durumdan haberdar olması gerektiğini düşünüyorsanız bu hatayı **`pass`** ile geçiştirmek yerine, kullanıcıya **hatayla ilgili makul ve anlaşılır bir mesaj göstermeyi düşünebilirsiniz**.

Yukarıda anlatılan durumların dışında, **`pass`** deyimini kodlarımızın **henüz taslak aşamasında olduğu zamanlarda da kullanabiliyoruz**. Diyelim ki bir kod yazıyoruz. Programın gidişatına göre bir noktada yapmamız gereken bir işlem var ama henüz ne yapacağımıza karar vermedik. Böyle bir durumda **`pass`** deyiminden yararlanabiliriz.

Mesela birtakım **`if`** deyimleri yazmayı düşünüyor olalım:

```
if .....:
    böyle yap

elif .....:
    şöyle yap

else:
    pass
```

Burada henüz **`else`** bloğunda ne yapılacağına karar vermemiş olduğumuz için oraya bir **`pass`** koyarak durumu şimdilik geçiştiriyoruz. Program son hâline gelene kadar oraya bir şeyler yazmış olacağız.

Sözün özü; **`pass`** deyimlerini **herhangi bir işlem yapılmasının gerekli olmadığı durumlar için kullanıyoruz**. İleride işe yarar programlar yazdığınızda, bu **`pass`** deyiminin göründüğünden **daha faydalı** bir araç olduğunu anlayacaksınız.

24 KARAKTER DİZİLERİNİN METOTLARI

Bu bölüme kadar *karakter dizilerinden, listelerden, demetlerden, sözlüklerden, kümelerden* ve *dosyalardan* söz ettik. Ayrıca *karakter dizileri dışındaki veri tiplerinin metotlarından* da bahsettik. İşte bu bölümde bu kez **Python'daki karakter dizilerinin sahip olduğu metotları** inceleyeceğiz. Bu metotlar sayesinde, karakter dizilerinin niteliklerini kolaylıkla değiştirebilecek veya bunlara yeni özellikler katabileceğiz.

İsterseniz öncelikle Python'ın bize **hangi karakter dizisi metotlarını** sunduğunu topluca görelim.

Karakter dizilerinin metotlarını listelemek için birkaç **farklı yöntemden faydalanabiliriz**.

Bunlardan **ilki şöyle** olabilir:

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

Burada **`dir()`** fonksiyonuna argüman olarak **`str`** adını geçiyoruz.

Bunun dışında, karakter dizilerini listelemek için **şu yöntemden de yararlanabiliriz**:

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

Bu komutlardan herhangi birini verdiğimizde şu **çıktıyı** elde edeceğiz:

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__
mod__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__
repr__',
'__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', 'capitalize', 'casefold', 'center', 'count',
'encode',
'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index',
'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit',
'isidentifier',
'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle',
'isupper',
'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition',
'replace',
'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
'split',
'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate',
'upper', 'zfill']
```

Her zaman olduğu gibi bizi **bu listede ilgilendiren metotlar** şunlar:

```
>>> for i in dir(""):
...     if "__" not in i:
...         print(i)
```

Şimdi sıra geldi bu metotları tek tek incelemeye...

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
```



Aşağıdaki metotları Türkçe karakter içeren karakter dizileri ile birlikte kullanmaya çalıştığınızda, beklediğiniz sonuçları alamadığınızı görebilirsiniz. Bu metotların Türkçe karakterler ile birlikte nasıl kullanılması gerektiğini **bölüm sonunda** açıklayacağız.

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

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

Eğer isterseniz bu metodu bir **for** döngüsü içinde **liste ögelerine** de tek tek uygulayabilirsiniz:

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

upper() Metodu

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

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

lower() Metodu

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

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

casefold() Metodu

Bu metot işlev olarak **lower()** metoduna çok benzer. Hatta Türkçe açısından, bu metodun **lower()** metodundan **hiçbir farkı yoktur**. Ancak bazı başka dillerde bu metot, bazı harfler için **lower()** metodunun verdiğinden farklı bir çıktı verir. Örneğin Almancadaki '**ß**' harfi bu duruma bir **örnek** olabilir:

```
>>> "ß".lower()
'ß'
>>> "ß".casefold()
'ss'
```

Gördüğünüz gibi **lower()** ve **casefold()** metotları bu harfe farklı davranıyor.

swapcase() Metodu

Bu metot da karakter dizilerindeki harflerin **büyüklüğü**/**küçüklüğü** ile ilgilidir. Metodumuz bize, bir karakter dizisinin o anda sahip olduğu harflerin **büyüklük** ve **küçüklük özellikleri arasında geçiş yapma imkânı sağlıyor**. Yani eğer o anda bir harf büyükse bu metodu kullandığımızda o harf küçülüyor. Eğer bu harf o anda küçükse; **bu metot o harfi büyük harfe çeviriyor**. Gördüğünüz gibi bu metodun işlevini anlatarak açıklamak zor oluyor. O yüzden hemen birkaç **örnek** yapalım:

```
>>> a = “kebap”
>>> a.swapcase()
KEBAP

>>> b = “KEBAP”
>>> b.swapcase()
kebap

>>> c = “KeBaP”
>>> c.swapcase()
kEbAp
```

title() Metodu

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

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

center() Metodu

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

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

İstersek boşluk yerine **kendi belirlediğimiz bir karakteri de yerleştirebiliriz**:

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

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

ljust() Metodu

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

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

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

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

rjust() Metodu

Bu metot ise **`ljust()`** metodunun **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 ufak bir **örnek** verelim:

```
import time

while True:
    for i in range(21):
        time.sleep(1)
        print(str(i).zfill(2))
        if i > 20:
            continue
```

Burada **standart kütüphane** modüllerinden biri olan **`time`** modülünü kullandığımıza dikkat edin. Bu modül, **saat ile ilgili işlemler yapabilmemizi sağlayan nitelik ve fonksiyonlar içeriyor**. Mesela bu modül içinde yer alan **`sleep()`** adlı fonksiyonu kullanarak **kodlarımızın kendi belirlediğimiz bir süre boyunca duraksamasını sağlayabiliyoruz**.

Mesela yukarıdaki örnekte **`for`** döngüsünün her bir dönüşü arasına **birer saniyelik duraksamalar ekledik**.

REPLACE() METODU

Python'daki karakter dizisi metotları içinde **belki de en çok işimize yarayacak metotlardan birisi** bu **`replace()`** metodudur. '**`replace`**' kelimesi İngilizcede **değiştirmek**, **yerine koymak** gibi anlamlara gelir. Dolayısıyla anlamından da anlaşılacağı gibi **bu metot yardımıyla bir karakter dizisi içindeki karakterleri başka karakterlerle değiştiriyoruz**. Metot, **şu formül üzerine işler**:

```
karakter_dizisi.replace("eski_karakter", "yeni_karakter")
```

Hemen bir **örnek** vererek durumu somutlaştıralım:

```
>>> karakter = "Kahramanmaraşlı Abdullah"
>>> karakter.replace("a","o")
Kohromonmoroşlı Abdulloh
```

Gördüğünüz gibi **`replace()`** metodu yardımıyla karakter dizisi içindeki bütün '**a**' harflerini kaldırıp yerlerine '**o**' harfini koyduk.

Bu metodu, isterseniz bir karakteri **silmek** için de kullanabilirsiniz. **O zaman şöyle bir şey yapmamız gerekir**:

```
>>> karakter = "Adanalı istihza"
>>> karakter_dgs = karakter.replace("a", "")
>>> karakter_dgs
Adnlı istihz
```

Burada bir karakteri silmek için **içi boş bir karakter dizisi** oluşturduğumuza dikkat edin.

`replace()` metodunun, yukarıdaki formülde belirtmediğimiz *üçüncü bir parametresi daha vardır*. Dikkat ettiyseniz yukarıdaki kod örneklerinde **`replace()`** **metodu karakter dizisi içindeki bir karakteri**, **dizi içinde geçtiği her yerde değiştiriyordu**. Yani örneğin **`a.replace("b", "c")`** dediğimizde, **a** değişkeninin sakladığı karakter dizisi içinde ne kadar '**b**' harfi varsa bunların hepsi '**c**' harfine dönüşüyor.

Bahsettiğimiz **üçüncü parametre yardımıyla**, karakter dizisi içinde geçen harflerin **kaç tanesinin değiştirileceğini belirleyebiliyoruz**:

```
>>> karakter = "Adanalı istihza"
>>> karakter_dgs = karakter.replace("a", "", 2)
>>> karakter_dgs
Adnlı istihza
```

Burada, "**Adanalı istihza**" karakter dizisi içinde geçen '**a**' harflerinden **2 tanesini siliyoruz**.

'**a**' harfi ile '**A**' harfinin Python'ın gözünde **birbirlerinden farklı iki karakter 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 hâlde ise yanlış anlamına gelen **`False` çıktısını** elde ediyoruz.

Bu metot sayesinde **karakter dizilerini ilk harflerine göre sorgulayıp sonuca göre istediğimiz işlemleri yaptırabiliyoruz**:

```
>>> liste = ["elma", "erik", "ev", "elbise",
... "karpuz", "armut", "kebap"]
>>> for i in liste:
...     if i.startswith("e"):
```

```
...         i.replace(“e”, “i”)
...
‘ilma’
‘irik’
‘iv’
‘ilbisi’
```

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

ENDSWITH() METODU

Bu metot, yukarıda anlattığımız **`startswith()`** metodunun yaptığı işin **tam tersini** yapıyor. Hatırlarsanız **`startswith()`** metodu ile bir *karakter dizisinin hangi harfle başladığını denetliyorduk*. İşte bu **`endswith()`** metoduyla da **karakter dizisinin hangi harfle bittiğini denetleyeceğiz**. Kullanımı **`startswith()`** metoduna çok benzer:

```
>>> a = “elma”
>>> a.endswith(“a”)
True

>>> b = “armut”
>>> a.endswith(“a”)
False
```

Bu metot yardımıyla, **cümle sonlarında bulunan istemediğiniz karakterleri ayıklayabilirsiniz**:

```
>>> kd1 = “ekmek elden su gölden!”
>>> kd2 = “sakla samanı gelir zamanı!”
>>> kd3 = “damlaya damlaya göl olur...”
>>> kd4 = “el elden üstündür...”
>>> for i in kd1, kd2, kd3, kd4:
...     if i.endswith(“!”):
...         i.replace(“!”, “”)
...
ekmek elden su gölden
sakla samanı gelir zamanı
```

COUNT() METODU

`count()` metodu bize bir karakter dizisi içinde bir **karakterden kaç adet bulunduğunu denetleme imkânı verecek**. Hemen bir örnek verelim:

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

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

ISALPHA() METODU

Bu metot yardımıyla, **bir karakter dizisinin alfabetik olup olmadığını denetleyeceğiz**. *Peki, alfabetik ne demek?*

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 dizisidir**:

```
>>> a = “12345”
```

Metodumuz yardımıyla bunu **doğrulayabiliriz**:

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

Ama şu karakter dizisi **sayısal değildir**:

```
>>> b = “123445b”
```

Hemen kontrol edelim:

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

ISALNUM() METODU

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

Daha önce bahsettiğimiz metotlardan hatırlayacaksınız:

Alfabetik karakter dizileri, **alfabe harflerinden oluşan karakter dizileridir**. Sayısal karakter dizileri, **sayılardan oluşan karakter dizileridir**.

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

```
>>> a = “123abc”
```

İsterseniz hemen bu yeni metodumuz yardımıyla bunu **doğrulayalım**:

```
>>> a.isalnum()
True
```

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

```
>>> b = “123abc>”
>>> b.isalnum()
False
```

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

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

ISLOWER() METODU

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

```
>>> kent = “istanbul”
>>> kent.islower()
True
```

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

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

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

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

isupper() Metodu

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

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

istitle() Metodu

Daha önce öğrendiğimiz metotlar arasında **`title()`** adlı bir metot vardı. Bu metot yardımıyla *tamamı küçük harflerden oluşan bir karakter dizisinin ilk harflerini büyütebiliyorduk*. İşte şimdi, öğreneceğimiz **`istitle()`** metodu da **bir karakter dizisinin ilk harflerinin büyük olup olmadığını kontrol etmemizi** sağlayacak:

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

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

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

isspace() Metodu

Bu metot ile bir karakter dizisinin **tamamen boşluk karakterlerinden oluşup oluşmadığını kontrol ediyoruz**. Eğer bir karakter dizisi *tamamen boşluk karakterinden oluşuyorsa bu metot* **`True` çıktısı** verecektir. Aksi hâlde 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.

isidentifier() Metodu

Bu metodu kullanarak bir karakter dizisinin; **değişken**, **fonksiyon** veya **sınıf adı** (*yani bir tanımlayıcı*) **olarak kullanılıp kullanılamayacağını** denetleyebilirsiniz:

```
>>> "a".isidentifier()
True
```

Demek ki "**a**" karakter dizisini bir **tanımlayıcı** olarak kullanabilirmişiz. Yani şu şekilde tanımlamalar yapabiliyoruz:

```
>>> a = "fırat"
>>> def a:
>>> class a:
```

Bir de şuna bakalım:

```
>>> "0".isidentifier()
False
```

Buna göre "**0**" karakter dizisini bir **tanımlayıcı olarak kullanamıyoruz**. Yani şu şekilde tanımlamalar yapamıyoruz:

```
>>> 0 = "fırat"
>>> def 0:
>>> class 0:
```

EXPANDTABS() METODU

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

```
>>> a = "elma\tbir\tmeyvedir"
>>> 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 imkânı da sunar**. Bunun için şöyle bir söz dizimini kullanabiliriz:

```
"karakter_dizisi".find("aranacak_karakter",
                       başlangıç_noktası,
                       bitiş_noktası)
```

Bir **örnek** verelim:

```
>>> a = "adana"
```

Burada normal bir şekilde "**a**" harfini arayalım:

```
>>> a.find("a")
0
```

Doğal olarak **find()** metodu, karakter dizisi içinde ilk bulduğu "**a**" harfinin **konumunu** söyleyecektir. Bizim örneğimizde "**a**" harfi kelimenin başında geçtiği için çıktıda "**0**" ifadesini görüyoruz. Demek ki bu karakter dizisi içindeki ilk "**a**" harfi **0. konumdaymış**.

İstersek şöyle bir **arama yöntemi** de kullanabiliriz:

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

Bu arama **yöntemi şu sonucu** verecektir:

```
2
```

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

```
>>> a[1:3]
"da"
```

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

rfind() Metodu

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

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

>>> a.rfind("a")
4
```

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

index() Metodu

index() metodu, yukarıda anlattığımız **find()** metoduna çok benzer. **İki metot da aynı işi yapar**:

```
>>> a = "istanbul"
>>> a.index("t")
2
```

Bu metot da bize, tıpkı **`find()`** metodunda olduğu gibi **konuma göre arama olanağı sunar**:

```
>>> b = "kahramanmaraş"
>>> b.index("a", 8, 10)
9
```

Demek ki **`b`** değişkeninin tuttuğu karakter dizisinin 8 ve 10 numaralı konumları arasında "**`a`**" harfi **9. sırada yer alıyormuş**.

Peki, bu **`index()`** metodunun **`find()`** metodundan farkı nedir?

Hatırlarsanız **`find()`** metodu **aradığımız karakteri bulamadığı zaman** -1 sonucunu veriyordu. **`index()`** metodu ise aranan karakteri bulamadığı zaman bir **hata mesajı** gösterir bize. **Örneğin**:

```
>>> c = "istanbul"
>>> c.index("m")
```

Bu kodlar şu **çıktıyı** verir:

```
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
ValueError: substring not found
```

RINDEX() METODU

`rindex()` metodu da **`index()`** metodu ile **aynıdır**. Farkları, **`rindex()`** metodunun karakter dizisini **sağdan sola** doğru, **`index()`** metodunun ise **soldan sağa** doğru okumasıdır:

```
>>> c = "adana"
>>> c.index("a")
0

>>> c.rindex("a")
4
```

JOIN() METODU

Bu metodu açıklamak biraz **zor** ve **kafa karıştırıcı** olabilir. O yüzden açıklama yerine doğrudan bir örnekle, bu metodun ne işe yaradığını göstermeye çalışalım. Şöyle bir **karakter dizimiz** olsun:

```
>>> a = "Linux"
```

Şimdi, **şöyle bir işlem yapalım**:

```
>>> ".".join(a)
```

Elde edeceğimiz **çıktı** şöyle olur:

```
L.i.n.u.x
```

Sanırım burada `join()` metodunun ne iş yaptığını anladınız. "`Linux`" karakter dizisi içindeki bütün karakterlerin arasına birer tane . (*nokta*) koydu. Tabii ki nokta yerine **başka karakterler** de kullanabiliriz:

```
>>> "*".join(a)
L*i*n*u*x
```

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

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

Bu listenin ögelerini **karakter dizileri hâlinde** ve **belli bir ölçüte göre almak için şu kodu** kullanıyoruz:

```
>>> "; ".join(a)
python; php; perl; C++; java
```

İstersek bu kodu bir değişken içinde depolayıp **kalıcı hâle** 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() VE MAKETRANS() METOTLARI

Karakter dizilerinin **`translate()`** ve **`maketrans()`** metotları, herhangi bir karakter dizisini, **bir çeviri tablosu içinde belirtilen ölçütler çerçevesinde başka bir karakter dizisine çevirmek için** kullanılır. Bu soyut tanımın tam olarak ne anlama geldiğini anlayabilmek için basit bir **örnek** verelim.

Diyelim ki biz bir cümle içinde geçen her bir harfi, alfabede kendinden sonraki ilk harfle değiştirerek ilgili cümleyi **şifrelemek** istiyoruz. Yani mesela elimizde "**`python`**" diye bir karakter dizisi varsa biz bu karakter dizisi içindeki **her bir harfi alfabede kendinden bir sonraki harfle değiştirerek** "**`rzuıöo`**" hâline dönüştürmek istiyoruz. Bunun için ilk olarak alfabedeki **hangi harfin hangi harfe karşılık gelmesini istediğimizi belirlememiz gerekiyor**. Dikkatlice bakın:

```
kaynak= "abcçdefgğhıijklmnoöprsştuüvyz"
hedef = "bcçdefgğhıijklmnoöprsştuüvyza"
```

Gördüğünüz gibi **`kaynak`** adlı değişken içinde öncelikle alfabenin **normal hâlini** yazdık. **`hedef`** adlı değişken içinde ise **`kaynak`** değişkeni içinde yer alan her bir harfin hangi harfe karşılık gelmesini istediğimiz belirttik. Buna göre mesela '**a**' harfinin '**b**' harfine, '**j**' harfinin '**k**' harfine, '**z**' harfinin de '**a**' harfine karşılık gelmesini istiyoruz.

Bu işlemleri tamamladıktan sonra **kaynak** ve **hedef** değişkenlerini temel alarak bir **çeviri tablosu oluşturmamız gerekiyor**. İşte bunun için karakter dizilerinin **maketrans()** adlı metodunu kullanacağız:

```
çeviri_tablosu = str.maketrans(kaynak, hedef)
```

Burada **maketrans()** metodunu herhangi bir karakter dizisi üzerinde değil doğrudan **str** üzerinde çalıştırdığımıza dikkat edin. Eğer arzu ederseniz bu metodu **rastgele herhangi bir karakter dizisi üzerinde de çalıştırabilirsiniz**:

```
çeviri_tablosu = ''.maketrans(kaynak, hedef)
```

Burada bu metodu **boş bir karakter dizisi üzerinde** çalıştırdık. Bu karakter dizisinin ne olduğunun hiçbir önemi yoktur:

```
çeviri_tablosu = 'ghihfsfd'.maketrans(kaynak, hedef)
```

Ancak **en makul seçimin**, karakter dizilerine işaret eden **str** nesnesi olacağını söyleyebiliriz:

```
çeviri_tablosu = str.maketrans(kaynak, hedef)
```

Kodlarımız şu ana kadar şöyle görünüyor:

```
kaynak= "abcçdefgğhıijklmnoöprsştuüvyz"
hedef = "bcçdefgğhıijklmnoöprsştuüvyza"

çeviri_tablosu = str.maketrans(kaynak, hedef)
```

Bu aşamada kullanıcıya h**angi karakter dizisini şifrelemek istediğini sormamız gerekiyor**:

```
metin = input("şifrelemek istediğiniz metni girin:")
```

Şimdi de yukarıda oluşturduğumuz çeviri tablosuna göre kullanıcıdan aldığımız karakter dizisini bir **dönüştürme işlemine** tabi tutacağız. Bunun için de **`translate()`** metodunu kullanacağız:

```
çıktı = metin.translate(çeviri_tablosu)
```

Gördüğünüz gibi **`translate()`** metodunu **`metin`** adlı değişken üzerine uyguluyoruz. Bu metodun parametresi ise yukarıda tanımladığımız **`çeviri_tablosu`** adlı değişken. **Yani bu komutla tam olarak şöyle demiş oluyoruz**:

`metin` adlı karakter dizisini **`çeviri_tablosu'`**na göre çevir!

Son olarak **`çıktı`** değişkeninin **değerini ekrana yazdıralım**:

```
print(çıktı)
```

Dilerseniz kodlarımızın **son hâlini** topluca görelim:

```
kaynak= "abcçdefgğhıijklmnoöprsştuüvyz"
hedef = "bcçdefgğhıijklmnoöprsştuüvyza"

çeviri_tablosu = str.maketrans(kaynak, hedef)

metin = input("şifrelemek istediğiniz metni girin:")

çıktı = metin.translate(çeviri_tablosu)

print(çıktı)
```

Şimdi, bu kodları çalıştıralım. Diyelim ki programımızın sorduğu soruya şu cevabı verdik:

```
istanbul büyükşehir belediyesi
```

Programımız bu karakter dizisi üzerinde **gerekli dönüştürme işlemlerini** yaparak bize şu **çıktıyı** verecektir:

```
jşubocüm cvzvltfıjs cfmfejzfşj
```

Tam da istediğimiz gibi bu karakter dizisindeki her bir harf, orijinal karakter dizisindeki her bir harfin **alfabe sırasına göre bir önündeki harfe karşılık geliyor**.

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**, **yeni satır** (**`\n`**) ve **sekme** (**`\t`**) gibi karakterleri **siler**:

```
>>> a = " boşluk "
>>> a.strip()
'boşluk'

>>> b = "boşluk\n"
>>> b.strip()
'boşluk'

>>> c = "boşluk\t"
>>> c.strip()
'boşluk'
```

Bu metot ayrıca **karakter dizisi tipinde bir argüman** da alabilir:

```
>>> a = "kahramanmaraş"
>>> a.strip("rhakş")
'manm'
```

Burada **`strip()`** metodu, kendisine argüman olarak verilen karakter dizisi içindeki her bir harfi, ana karakter dizisinin hem **baş** tarafında hem de **son** tarafında arar ve argüman içindeki harfi **karakter dizisi içinde bulamadığı anda** da **arama işlemini durdurur**:

```
>>> a = "adana"
>>> a.strip("da")
n
```

Gördüğünüz gibi **`strip()`** metodu, kendisine argüman olarak verilen "**`da`**" karakter dizisi içindeki harfleri "**`adana`**" karakter dizisi içinde hem soldan hem de sağdan arayarak **kırpa kırpa ilerliyor**. '**`n`**' harfine rastladığında ise arama işlemini durduruyor.

Bir de şuna bakalım:

```
>>> a = "python"
>>> a.strip("poth")
'ython'
```

Burada **`strip()`** metodu, "**`python`**" karakter dizisinin hem **sol** hem de **sağ** tarafına baktı ve sol tarafta "**`p`**" harfini gördü. Bu harf argüman listesinde ("**`poth`**") yer aldığı için bunu **kırpıp attı**. Ardından tekrar karakter dizisinin soluna ve sağına baktı. Bu kez sol tarafta "**`y`**" sağ tarafta ise "**`n`**" harfini gördü. Bu iki harf de **argüman listesinde bulunmadığı için arama işlemini sona erdirdi**.

Bu yapının tam olarak nasıl işlediğini anlamak için kendi kendinize denemeler yapmanızı tavsiye ederim.

rstrip() Metodu

Bu metot da tıpkı **strip()** metodu gibi çalışır, ama ondan farklı olarak **rstrip()** metodu, **bir karakter dizisinin sadece sonunda** (*sağında*) **yer alan boşluk ve yeni satır** (\n) **gibi karakterleri kırpar**:

```
>>> a = "boşluk "
>>> a.rstrip()
'boşluk'

>>> b = "boşluk\n"
>>> b.rstrip()
'boşluk'

>>> "kazak".rstrip("k")
'kaza'
```

lstrip() Metodu

Bu metot da tıpkı **strip()** metodu gibi çalışır, ama ondan farklı olarak **lstrip()** metodu, **bir karakter dizisinin sadece başında** (*solunda*) **yer alan boşluk ve yeni satır** (\n) **gibi karakterleri kırpar**:

```
>>> a = "boşluk "
>>> a.lstrip()
'boşluk'

>>> b = "boşluk\n"
>>> b.lstrip()
'boşluk'

>>> "kazak".lstrip("k")
'azak'
```

SPLITLINES() METODU

Bu metot yardımıyla, **bir karakter dizisini satır kesme noktalarından bölerek** bölünen ögeleri **liste hâline** getirebiliyoruz:

```
>>> satır = "Birinci satır\nİkinci satır"
>>> print(satır.splitlines())
["Birinci satır", 'İkinci satır']
```

SPLIT() METODU

Bu metot biraz, **`join()`** metodunun yaptığı işi tersine çevirmeye benzer. Hatırlarsanız **`join()`** metodu yardımıyla **bir listenin ögelerini etkili bir şekilde karakter dizisi hâlinde 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 ögesi hâline geldi**:

```
>>> yeni[0]
'Debian'

>>> yeni[1]
'Pardus'

>>> yeni[2]
'Ubuntu'

>>> yeni[3]
'SuSe'
```

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

```
>>> c = b.split(“,”, 1)
>>> print c
[‘Debian’, ‘ Pardus, Ubuntu, SuSe’]
```

Gördüğünüz gibi parantez içinde “**,**” ölçütünün yanına bir adet “**1**” sayısı koyduk. Çıktıyı dikkatle incelediğimizde **split()** metodunun bu parametre yardımıyla **karakter dizisi içinde sadece bir adet bölme işlemi yaptığını görüyoruz**. Yani oluşan listenin bir ögesi “**Debian**”, öteki ögesi de “**Pardus, Ubuntu, SuSe**” oldu. Bunu şu şekilde daha açık görebiliriz:

```
>>> c[0]
‘Debian’

>>> c[1]
‘ Pardus, Ubuntu, SuSe’
```

Gördüğünüz gibi listenin 0. ögesi **Debian** iken; listenin 1. ögesi “**Pardus, Ubuntu, Suse**” üçlüsü. Yani **bu üçlü tek bir karakter dizisi şeklinde tanımlanmış**. Yukarıda tanımladığımız **yeni** adlı listeyle “**c**” adlı listenin uzunluklarını karşılaştırarak durumu daha net görebiliriz:

```
>>> len(yeni)
4

>>> len(c)
2
```

Parantez içindeki **1** parametresini **değiştirerek** kendi kendinize denemeler yapmanız, metodu daha iyi anlamanıza yardımcı olacaktır.

rsplit() 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’]
```

Herhâlde bu şekilde **ikisi arasındaki fark daha belirgin** olmuştur.

Öyle değil, diyorsanız bir de şuna bakın:

```
>>> orneksplit[0]
‘www’

>>> ornekrsplit[0]
‘www.python.quotaless’
```

format() Metodu

Bu metodu **karakter dizilerini biçimlendirmek** için kullanıyoruz. **`format()`** metodunun yaptığı iş **f**'li karakter dizilerinin yaptığı işe çok benzer. Mesela çok basit bir **örnek** verelim:

```
>>> "{} ve {} iyi bir ikilidir!".format("Django", "Python")
'Django ve Python iyi bir ikilidir!'
```

Gördüğünüz gibi **`format()`** metodunu da tıpkı **f**'li karakter dizilerinde olduğu gibi **karakter dizilerinin içine birtakım değerleri monte edebilmek için kullanıyoruz**. Yukarıda işlemi **f**'li karakter dizileriyle şu şekilde gerçekleştirebiliriz:

```
>>> f"{'Django'} ve {'Python'} iyi bir ikilidir!"
'Django ve Python iyi bir ikilidir!'
```

`format()` metodunda küme parantezlerini, yukarıdaki örneklerde görüldüğü gibi **içi boş** olarak kullanabilirsiniz:

```
>>> print("Ad: {}\nSoyad: {}".format("Fırat", "Özgül"))
Ad: Fırat
Soyad: Özgül
```

Böyle bir durumda Python, karakter dizisi içindeki küme parantezleriyle, karakter dizisi dışındaki değerleri **teker teker** ve **sırasıyla** eşleştirecektir. Ama isterseniz küme parantezleri içine birer **sayı** yazarak karakter dizisi dışındaki değerlerin **hangi sırayla kullanılacağını belirleyebilirsiniz**. **Örneğin**:

```
>>> print("Soyad: {1}\nAd: {0}".format("Fırat", "Özgül"))
Soyad: Özgül
Ad: Fırat
```

Küme parantezleri içinde sayı kullanabilme imkânı sayesinde **değerlerin sırasını istediğiniz gibi düzenleyebilirsiniz**. Hatta bu özellik sayesinde değerleri bir kez yazıp **birden fazla sayıda tekrar edebilirsiniz**:

```
>>> "{0} {1} ({1} {0})".format("Fırat", "Özgül")
'Fırat Özgül (Özgül Fırat)'
```

Karakter dizilerinin **`format()`** metodu son derece **esnek** ve **detaylı bir araçtır**. Bu metodun yetenekleri burada anlattığımızdan çok daha geniştir. Ancak biz burada bu metodun sadece en temel özelliklerinden söz etmekle yetindik.

encode() Metodu

Herhangi bir verinin bir bilgisayarda depolanabilmesi için o verinin öncelikle '**`bayt`**' denen bir **veri tipine dönüştürülmesi gerekir**. Zira aslında **bilgisayarlar** *karakter dizilerinden, listelerden, demetlerden*, *sözlüklerden* veya *kümelerden* **anlamaz**. Bilgisayarların **anlayabildiği tek veri tipi baytlar**dır. Biz herhangi bir veriyi bilgisayara kaydetmek istediğimizde Python *gerekli dönüştürme işlemlerini alttan alta hâlledecektir*. Dolayısıyla biz genellikle arka planda dönen bu işlemlere şahit olmayız.

Örneğin bir dosyaya **veri yazdığımızı** düşünelim:

```
>>> dosya = open("falanca.txt", "w", encoding="utf-8")
>>> dosya.write("bir şeyler...")
>>> dosya.close()
```

Burada Python, '**`falanca.txt`**' adlı dosyaya yazmak istediğimiz verileri, **`open()`** fonksiyonuna verdiğimiz **`encoding`** parametresinde belirttiğimiz dil kodlamasına göre **arka planda bayt veri tipine çevirerek dosyaya yazacaktır**.

Dosya işlemleri konusunu anlatırken de söylediğimiz gibi eğer bu kodlarda bir **encoding** parametresi belirtmezsek bu dosyayı oluşturduğumuz işletim sistemi, dosyanın **dil kodlamasının ne olacağına kendisi karar verecektir**. Bir dil kodlaması belirtmediğiniz durumlarda, işletim sisteminizin **hangi dil kodlamasını seçeceğini şu komutla görebileceğimizi biliyoruz**:

```
>>> import locale
>>> locale.getpreferredencoding()
```

İşte, eğer bir **encoding** parametresi belirtmezseniz dosyaya yazacağınız metinler, **varsayılan olarak** yukarıdaki komutun çıktısında görünen dil kodlaması ile kodlanıp **dosyaya yazılacaktır**. Eğer bu varsayılan dil kodlaması dosyaya yazmaya çalıştığınız metinle **uyumlu değilse o metni o dosyaya yazamazsınız**:

```
>>> f = open("deneme.txt", "w")
>>> f.write("Привет")
UnicodeEncodeError: 'charmap' codec can't encode characters
```

Burada aslında olan şey şu: Python '**deneme.txt**' adlı dosyaya yazmak istediğimiz '**Привет**' karakter dizisini **locale.getpreferredencoding()** çıktısında görünen dil kodlaması ile kodlayarak '**bayt**' veri tipine dönüştürmeye çalışıyor. Çünkü başta da dediğimiz gibi **herhangi bir verinin bilgisayara kaydedilebilmesi için Python tarafından öncelikle** 'bayt' **adlı bir veri tipine dönüştürülmesi gerekir**.

Benim bu dosyayı oluşturduğum sistemin öntanımlı dil kodlaması **cp1252** olduğu için bu dil kodlaması "**Привет**" kelimesindeki **harfleri tanımıyor**. Ama **utf-8** adlı dil kodlaması bu kelime içindeki harfleri tanıyabilir:

```
>>> f = open("deneme.txt", "w", encoding="utf-8")
>>> f.write("Привет")
```

Burada dosyanın dil kodlamasını belirleme işini işletim sistemimize bırakmak yerine doğrudan kendimiz belirledik. Böylece dosyamız düzgün şekilde yazıldı.

İşte eğer siz de yazdığınız programlarda, karakter dizilerini belirli bir dil kodlamasına göre kodlayarak **bayt** veri tipine **dönüştürmek durumunda kalırsanız karakter dizilerinin `encode()` metodundan yararlanabilirsiniz**:

```
>>> veri = "Привет"
>>> veri.encode("utf-8")
b'\xd0\x9f\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\x82'
```

Gördüğünüz gibi "**`Привет`**" karakter dizisi utf-8 adlı dil kodlaması ile kodlandığında **`b'\xd0\x9f\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\x82'`** şeklinde temsil ediliyor. Burada **toplam 12 baytlık bir veri** var.

Benim sistemimdeki öntanımlı dil kodlaması olan **`cp1252`** ise "**`Привет`**" karakter dizisini hiç kodlayamıyor:

```
>>> veri.encode("cp1252")
UnicodeEncodeError: 'charmap' codec can't encode characters
```

Birkaç **örnek** daha verelim:

```
>>> veri = "kâğıt"
>>> veri.encode("utf-8")
b'k\xc3\xa2\xc4\x9f\xc4\xb1t'

>>> veri.encode("cp1254")
b'k\xe2\xf0\xfdt'
```

Gördüğünüz gibi farklı dil kodlamaları **aynı karakter dizisini farklı şekillerde kodlayıp** `bayt` veri tipine **çeviriyor**.

Eğer bir karakter dizisinin hangi dil kodlaması ile kodlandığını biliyorsanız, aynı dil kodlamasını kullanarak o karakter dizisinin **kodunu** da çözebilirsiniz. Yani, herhangi bir dil kodlaması kullanılarak bayt veri tipine dönüştürülmüş bir karakter dizisini, **yine aynı dil kodlamasını kullanarak bayt veri tipinden karakter dizisine de dönüştürebilirsiniz**:

```
>>> veri = b'k\xc3\xa2\xc4\x9f\xc4\xb1t'
>>> veri.decode("utf-8")
'kâğıt'
```

veya:

```
>>> veri = b'\xd0\x9f\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\x82'
>>> veri.decode("utf-8")
'Привет'
```

ya da:

```
>>> veri = b'k\xe2\xf0\xfdt'
>>> veri.decode("cp1254")
>>> kâğıt
```

Bu kodlarda bayt adlı veri tipinin **`decode()`** metodunu kullandığımıza dikkat edin.

Burada eğer **yanlış dil kodlamasını** kullanırsanız dönüştürme işlemi **başarılı olmayacaktır**:

```
>>> veri = b'k\xc3\xa2\xc4\x9f\xc4\xb1t'
>>> veri.decode("cp1254")
'kÃ¢ÄŸÄ±t'
```

veya:

```
>>> veri = b'k\xe2\xf0\xfdt'
>>> veri.decode("utf-8")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe2 in position
1: invalid continuation byte
```

Gördüğünüz gibi dil kodlamasını yanlış tahmin ettiğimizde programımızın **hata vererek çökme riski de var**. Eğer yazdığınız programlarda hangi dil kodlaması ile kodlandığını bilmediğiniz **bayt** tipli verilerle muhatap oluyorsanız en azından dil kodlamasını yanlış tahmin ettiğiniz durumlarda, **programınızın çökmemesi için** `decode()` fonksiyonunu **ikinci bir parametre** ile birlikte kullanabilirsiniz:

```
>>> veri = b'k\xe2\xf0\xfdt'
>>> veri.decode("utf-8", "replace")
'k???t'
```

"**`replace`**" parametresi sayesinde dil kodlaması **yanlış bile olsa** programımız çökmek yerine **doğru çözümlenemeyen baytların yerine bir soru işareti koyacaktır**.

Sözün özü, eğer herhangi bir şekilde bir karakter dizisini belirli bir dil kodlamasına göre **`bayt`** veri tipine dönüştürmeniz gerekirse karakter dizilerinin **`encode()`** metodunu kullanacaksınız.

Aynı şekilde, eğer **bayt** tipindeki bir veriyi belirli bir dil kodlamasına göre **karakter dizisine dönüştürmeniz gerekirse** baytların **`decode`** metodunu kullanacaksınız.

Böylece karakter dizisi metotlarını bitirmiş olduk. Sıra geldi, *bu metotların Türkçe karakterler ile olan uyumsuzluklarını ve bu sorunu nasıl çözeceğimizi anlatmaya...*

METOTLARDA TÜRKÇE KARAKTER SORUNU

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

Mesela şu örneklere bir bakalım:

```
>>> a = "istanbul"
>>> a.upper() #"İSTANBUL" vermeli.
ISTANBUL

>>> a = "KİTAP"
>>> a.lower() #"kitap" vermeli.
'ki'tap'

>>> a = "kİtAp"
>>> a.swapcase()
'KiTaP'
```

Gördüğünüz gibi Türkçedeki "**i**" harfi büyütüldüğünde "**İ**" hâline dönüşmediği gibi "**İ**" harfi de küçültüldüğünde "**i**" harfine **dönüşmüyor**.

Bu sorunu çözebilmek için **replace()** metodundan yararlanabilirsiniz:

```
>>> a = "istanbul"
>>> a.replace('i', 'İ').upper()
'İSTANBUL'

>>> a = "KİTAP"
>>> a.replace("İ", "i").lower()
'kitap'
```

`swapcase()` metodundaki sorunu düzeltmek için ise daha karmaşık bir yol izlemek gerekiyor. Bunu birazdan göreceğiz. Ama isterseniz yukarıda yaptığımız işlemleri **bir fonksiyon olarak tanımlayarak bunların kullanımını kolaylaştıralım**:

```
def tr_upper(kardiz):
    return kardiz.replace('i', 'İ').upper()
```

Burada basitçe **`tr_upper()`** adlı bir fonksiyon tanımladık. Bu fonksiyon **tek bir parametre** alıyor. Fonksiyon gövdesinde bu parametre içinde geçen '**`i`**' harfini '**`İ`**' harfine dönüştürüyoruz. Geri kalan harflerde bir problem olmadığı için de bunları büyütmek için normal **`upper()`** fonksiyonunu kullanıyoruz.

Bu fonksiyonu **test edelim**:

```
kelimeler = ["kadın", "çilek", "şçöğü", "istanbul", "ırmak",
             "sabun", "kağıt"]

for kelime in kelimeler:
    print(kelime.ljust(10), tr_upper(kelime))
```

Gördüğünüz gibi içinde '**`i`**' harfi olan kelimeleri de düzgün bir şekilde büyütebildik.

Aynı **Türkçe karakter probleminin** **`lower()`** metodu için de **geçerli** olduğunu biliyoruz. Bu metot da içinde '**`İ`**' ve "**`I`**" harfleri geçen kelimeleri düzgün küçültemiyor:

```
>>> "KADIN".lower()
'kadin'

>>> "IŞIK".lower()
'ışik'
```

Tıpkı **`upper()`** metodu için yaptığımız gibi **`lower()`** metodu için de **kendi `tr_lower()` metodumuzu tanımlayabiliriz**:

```
def tr_lower(kardiz):
    return kardiz.replace("I", 'ı').replace('İ', 'i').lower()
```

Burada da fonksiyon gövdesinde '**`I`**' harfini '**`ı`**' harfine, '**`İ`**' harfini ise '**`i`**' harfine dönüştürdük. Geri kalan harfler için ise **normal şekilde `lower()` metodunu kullandık**.

Gelin şimdi, bu fonksiyonu da **test edelim**:

```
kelimeler = ["KADIN", "ÇİLEK", "ŞÇÖĞÜİI", "İSTANBUL", "IRMAK",
             "SABUN", "KAĞIT"]

for kelime in kelimeler:
    print(kelime.ljust(10), tr_lower(kelime))
```

Gördüğünüz gibi artık Türkçe karakterler içeren büyük harfli kelimeleri de **küçük harfli kelimelere dönüştürebiliyoruz**.

Gelelim **`swapcase()`** metoduna...

Bu metot da tıpkı **`lower()`** ve **`upper()`** metodunda olduğu gibi bazı Türkçe karakterleri **düzgün büyütüp küçültemez**. O yüzden burada da kendi metodumuzu tanımlayacağız.

Dikkatlice bakın:

```
def tr_swapcase(kardiz):
    kelime = ''

    for s in kardiz:
        if s.isupper():
            kelime += tr_lower(s)
        else:
            kelime += tr_upper(s)

    return kelime
```

Bu fonksiyonu da **test edelim**:

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

Sonuç başarılı...

Burada biraz önce tanımladığımız **`tr_lower()`** ve **`tr_upper()`** fonksiyonlarından yararlandığımıza dikkat edin. Bu kodların çalışabilmesi için elbette bu iki fonksiyonun da **ulaşılabilir durumda olması gerekiyor**:

```
def tr_upper(kardiz):
    return kardiz.replace('i', 'İ').upper()

def tr_lower(kardiz):
    return kardiz.replace("I", 'ı').replace('İ', 'i').lower()

def tr_swapcase(kardiz):
    kelime = ''

    for s in kardiz:
        if s.isupper():
            kelime += tr_lower(s)
        else:
            kelime += tr_upper(s)

    return kelime

print(tr_swapcase("şEkErLi çAy"))
```

Eğer arzu ederseniz bu üç fonksiyonu, mesela '**`tr_metotlar.py`**' adlı ayrı bir modül içine kaydedip gerektiğinde bu metotları **`import`** komutuyla programlarınıza **dâhil edebilirsiniz**.

Yani öncelikle içeriği şöyle olan, '**`tr_metotlar.py`**' adlı bir dosya oluşturun:

```
def tr_upper(kardiz):
    return kardiz.replace('i', 'İ').upper()

def tr_lower(kardiz):
```

```
    return kardiz.replace("I", 'ı').replace('İ', 'i').lower()

def tr_swapcase(kardiz):
    kelime = ''

    for s in kardiz:
        if s.isupper():
            kelime += tr_lower(s)
        else:
            kelime += tr_upper(s)

    return kelime
```

Artık **`tr_upper()`** **`tr_lower()`** ve **`tr_swapcase()`** metotlarına ihtiyacınız olduğunda, bu metotları **tekrar tekrar tanımlamak yerine** **`tr_metotlar`** adlı modülden **içe aktarabilirsiniz**:

```
from tr_metotlar import *

print(tr_upper("çilek"))
print(tr_lower("ÇİLEK"))
print(tr_swapcase("çİlEk"))
```

Böylece Python'daki tüm karakter dizisi metotlarını incelemiş olduk. Bu konuyu iyice sindirebilmek için kendi kendinize *bolca örnek* ve *denemeler* yapmanızı, **bu konuyu arada sırada tekrar etmenizi öneririm**.

DÜZENLİ İFADELER (REGULAR EXPRESSIONS)

25

Düzenli ifadeler, Python Programlama Dili'ndeki en çetrefilli konulardan biridir. Hatta düzenli ifadelerin Python içinde ayrı bir dil olarak düşünülmesi gerektiğini söyleyenler dahi vardır. Bütün zorluklarına rağmen, programlama deneyimimizin bir noktasında mutlaka karşımıza çıkacak olan bu yapıyı öğrenmemizde **büyük fayda** var. Düzenli ifadeleri öğrendikten sonra, elle yapılması saatler sürecek bir işlemi *saliseler içinde* yapabildiğinizi gördüğünüzde eminim düzenli ifadelerin ne büyük bir nimet olduğunu anlayacaksınız. Tabii her güzel şey gibi düzenli ifadelerin nimetlerinden yararlanabilecek düzeye gelmek de biraz kan ve gözyaşı istiyor.

Peki, düzenli ifadeleri kullanarak neler yapabiliriz? Çok genel bir ifadeyle, bu yapıyı kullanarak metinleri veya karakter dizilerini parmağımızda oynatabiliriz. Örneğin bir web sitesinde dağınık hâlde duran verileri bir çırpıda ayıklayabiliriz. Bu veriler, **toplu hâlde görmek istediğimiz web adreslerinin bir listesi olabilir**. Bunun dışında örneğin, çok sayıda belge üzerinde **tek hareketle** istediğimiz değişiklikleri yapabiliriz.

Genel bir kural olarak düzenli ifadelerden **kaçabildiğimiz müddetçe kaçmamız gerekir**. Eğer Python'daki karakter dizisi metotları, o anda yapmak istediğimiz şey için **yeterli geliyorsa mutlaka o metotları kullanmalıyız**. Çünkü karakter dizisi metotları, düzenli ifadelere kıyasla hem daha **basit** hem de **çok daha hızlı**dır. Ama bir noktadan sonra karakter dizilerini kullanarak yazdığınız kodlar iyice karmaşıklaşmaya başlamışsa kodların her tarafı `if` deyimleriyle dolmuşsa hatta basit bir işlemi gerçekleştirmek için yazdığınız

kod sayfa sınırlarını zorlamaya başlamışsa işte o noktada **artık düzenli ifadelerin dünyasına adım atmanız gerekiyor olabilir**. Ama bu durumda Jamie Zawinski'nin şu sözünü de aklınızdan çıkarmayın: "*Bazıları, bir sorunla karşı karşıya kaldıklarında şöyle der: 'Evet, düzenli ifadeleri kullanmam gerekiyor.' İşte onların bir sorunu daha vardır artık...*"

Başta da söylediğim gibi düzenli ifadeler, bize **zorlukları unutturacak kadar büyük kolaylıklar** sunar. Emin olun düzenli ifadeleri öğrenip kullanmak, yüzlerce dosya üzerinde tek tek elle değişiklik yapmaktan **daha zor değildir**. Hem zaten biz de bu sayfalarda bu sevimsiz konuyu olabildiğince sevimli hâle getirmek için elimizden gelen çabayı göstereceğiz. Sizin de çaba göstermeniz, bol bol alıştırma yapmanız durumunda düzenli ifadeleri kavramak o kadar da zorlayıcı olmayacaktır. Unutmayın, düzenli ifadeler ne kadar uğraştırıcı olsa da programcının en önemli silahlarından biridir. Hatta **düzenli ifadeleri** öğrendikten sonra onsuz geçen yıllarınıza acıyacaksınız.

Şimdi, lafı daha fazla uzatmadan işimize koyulalım.

Düzenli İfadelerin Metotları

Python'daki düzenli ifadelere ilişkin her şey bir modül **içinde** tutuluyor. Bu modülün adı **`re`**. Tıpkı daha önce öğrendiğimiz bütün modüllerde olduğu gibi düzenli ifadeleri kullanabilmemiz için de öncelikle bu **`re`** modülünü **içe aktarmamız gerekecek**. Bu işlemi nasıl yapacağımızı çok iyi biliyorsunuz:

```
>>> import re
```

Düzenli ifadeler bir programcının en önemli silahlarından biridir. Şu hâlde silahımızın özelliklerine bakalım. Yani bu yapının **bize sunduğu araçları** şöyle bir listeleyelim. Etkileşimli kabukta şu kodu yazıyoruz:

```
>>> dir(re)
```

Tabii yukarıdaki **`dir(re)`** komutunu yazmadan önce **`import re`** şeklinde modülümüzü içe aktarmış olmamız gerekiyor. Gördüğünüz gibi **`re`** modülü içinde epey **metot**/**fonksiyon** var. Biz bu sayfada ve ilerleyen sayfalarda, yukarıdaki metotların/fonksiyonların en sık kullanılanlarını size olabildiğince yalın bir şekilde anlatmaya çalışacağız.

Eğer isterseniz şu komutu kullanarak yukarıdaki metotlar/fonksiyonlar hakkında yardım da alabileceğinizi biliyorsunuz:

```
>>> help(metot_veya_fonksiyon_adı)
```

Bir **örnek** vermek gerekirse:

```
>>> help(re.match)
Help on function match in module re:
match(pattern, string, flags=0)
Try to apply the pattern at the start of the string,
returning a match object, or None if no match was found.
```

Ne yazık ki Python'ın **yardım dosyaları hep İngilizce**. Dolayısıyla eğer İngilizce bilmiyorsanız bu yardım dosyaları pek işinize yaramayacaktır. Bu arada yukarıdaki yardım bölümünden çıkmak için **klavyedeki q düğmesine basmanız gerekir**.

MATCH() METODU

Bir önceki bölümde metotlar hakkında yardım almaktan bahsederken ilk örneğimizi **`match()`** metoduyla vermiştik, o hâlde **`match()`** metodu ile devam edelim. **`match()`** metodunu tarif etmek yerine, isterseniz bir **örnek** yardımıyla bu metodun **ne işe yaradığını anlamaya çalışalım**. Diyelim ki elimizde şöyle bir **karakter dizisi** var:

```
>>> a = "python güçlü bir programlama dilidir."
```

Varsayalım ki biz bu karakter dizisi içinde "**`python`**" **kelimesi geçip geçmediğini öğrenmek istiyoruz**. Ve bunu da **düzenli ifadeleri** kullanarak yapmak istiyoruz. Düzenli ifadeleri bu örneğe uygulayabilmek için yapmamız gereken şey, öncelikle bir **düzenli ifade kalıbı oluşturup** daha sonra bu kalıbı yukarıdaki **karakter dizisi ile karşılaştırmak**. Biz bütün bu işlemleri, **`match()`** metodunu kullanarak yapabiliriz:

```
>>> re.match("python", a)
```

Burada "**python**" şeklinde bir **düzenli ifade kalıbı oluşturduk**. Düzenli ifade kalıpları **match()** metodunun **ilk argümanıdır** (*yani parantez içindeki ilk değer*). **İkinci argümanımız ise** (*yani parantez içindeki ikinci değer*), hazırladığımız kalıbı kendisiyle eşleştireceğimiz **karakter dizisi** olacaktır. Klavyede **ENTER** tuşuna bastıktan sonra karşımıza şöyle bir **çıktı** gelecek:

```
<re.Match object; span=(0, 6), match='python'>
```

Bu çıktı, düzenli ifade kalıbının karakter dizisi ile **eşleştiği** anlamına geliyor. Yani aradığımız şey, **karakter dizisi içinde** bulunmuş.

Yukarıdaki çıktıda gördüğümüz ifadeye, Python'cada **eşleşme nesnesi** (**match object**) adı veriliyor. Çünkü **match()** metodu yardımıyla yaptığımız şey aslında bir eşleştirme işlemidir ('*match' kelimesi İngilizce'de eşleşmek anlamına geliyor*). Biz burada "**python**" düzenli ifadesinin **a** değişkeniyle eşleşip eşleşmediğine bakıyoruz. Yani **re.match("python", a)** komutu aracılığıyla, "**python**" **karakter dizisi ile a değişkeninin tuttuğu karakter dizisinin eşleşip eşleşmediğini sorguluyoruz**.

Bizim örneğimizde "**python**", **a** değişkeninin tuttuğu karakter dizisi ile eşleştiği için bize bir **eşleşme nesnesi döndürülüyor**.

Bir de şu örneğe bakalım:

```
>>> re.match("Java", a)
```

Burada **ENTER** tuşuna bastığımızda hiçbir **çıktı** almıyoruz. Aslında biz görmesek de Python burada **None çıktısı veriyor**. Eğer yukarıdaki komutu şöyle yazarsak **None çıktısını biz de görebiliriz**:

```
>>> print(re.match("Java", a))
None
```

Gördüğünüz gibi ENTER tuşuna bastıktan sonra **None** çıktısı geldi. Demek ki "**Java**" karakter dizisi, **a** değişkeninin tuttuğu karakter dizisi ile **eşleşmiyormuş**. Buradan çıkardığımız sonuca göre; Python **match()** metodu yardımıyla aradığımız şeyi **eşleştirdiği zaman**, **bir eşleşme nesnesi** (**match object**) **döndürüyor**. Eğer eşleşme **yoksa** o zaman da **None değerini döndürüyor**.

Biraz kafa karıştırmak için şöyle bir **örnek** verelim:

```
>>> a = “Python güçlü bir dildir”
>>> re.match(“güçlü”, a)
```

Burada “**a**” değişkeninde “**güçlü**” ifadesi geçtiği hâlde **match()** metodu bize bir **eşleşme nesnesi döndürmedi**. Aslında bu *normal*. Çünkü **match()** metodu bir karakter dizisinin **sadece en başına bakar**. Yani “**Python güçlü bir dildir**” ifadesini tutan **a** değişkenine **re.match(“güçlü”, a)** gibi bir fonksiyon uyguladığımızda, **match()** metodu **a** değişkeninin yalnızca en başına bakacağı için ve **a** değişkeninin en başında “**güçlü**” yerine “**python**” olduğu için **match()** metodu bize **olumsuz yanıt veriyor**.

Aslında **match()** metodunun yaptığı bu işi, karakter dizilerinin **split()** **metodu yardımıyla da yapabiliriz**:

```
>>> a.split()[0] == “python”
True
```

Demek ki **a** değişkeninin en başında “**python**” ifadesi varmış.

Bir de şuna bakalım:

```
>>> a.split()[0] == “güçlü”
False
```

Veya aynı işi sadece **startswith()** metodunu kullanarak dahi yapabiliriz:

```
>>> a.startswith(“python”)
```

Eğer düzenli ifadelerden tek beklentiniz, bir karakter dizisinin en başındaki veriyle **eşleştirme işlemi yapmaksa;** **split()** veya **startswith()** metotlarını kullanmak daha mantıklıdır. Çünkü **split()** ve **startswith()** metotları **match()** metodundan çok daha **hızlı** çalışacaktır.

`match()` metodunu kullanarak birkaç **örnek** daha yapalım:

```
>>> sorgu = "1234567890"
>>> re.match("1", sorgu)
<re.Match object; span=(0, 1), match='1'>

>>> re.match("1234", sorgu)
<re.Match object; span=(0, 4), match='1234'>

>>> re.match("124", sorgu)
None
```

İsterseniz şimdiye kadar öğrendiğimiz şeyleri şöyle bir gözden geçirelim:

1. Düzenli ifadeler Python'ın **çok güçlü** araçlarından biridir.
2. Python'daki düzenli ifadelere ilişkin bütün fonksiyonlar, **`re`** adlı bir **modül içinde yer alır**.
3. Dolayısıyla düzenli ifadeleri kullanabilmek için öncelikle bu **`re`** modülünü **`import re`** diyerek **içe aktarmamız gerekir**.
4. **`re`** modülünün içeriğini **`dir(re)`** komutu **yardımıyla** listeleyebiliriz.
5. **`match()`** metodu **`re`** **modülü içindeki fonksiyonlardan biridir**.
6. **`match()`** metodu bir karakter dizisinin yalnızca **en başına** bakar.
7. Eğer aradığımız şey karakter dizisinin en başında yer alıyorsa **`match()`** metodu bir **eşleştirme nesnesi döndürür**.
8. Eğer aradığımız şey karakter dizisinin **en başında yer almıyorsa** **`match()`** metodu **`None`** değeri döndürür.

Daha önce söylediğimiz gibi **`match()`** metodu ile bir **eşleştirme** işlemi yaptığımızda, eğer **eşleşme varsa Python bize bir eşleşme nesnesi döndürecektir**. Eğer eşleşen **karakter dizisini almak istersek** **`group()`** metodunu kullanabiliriz:

```
>>> a = “perl, python ve ruby yüksek seviyeli dillerdir.”
>>> b = re.match(“perl”, a)
>>> print(b.group())
perl
```

Burada, **`re.match(“perl”, a)`** fonksiyonunu bir değişkene atadık. Hatırlarsanız, bu fonksiyonu komut satırına yazdığımızda bir eşleşme nesnesi elde ediyorduk. İşte burada değişkene atadığımız şey, **aslında bu eşleşme nesnesinin kendisi oluyor**.

Bu durumu şu şekilde **teyit edebilirsiniz**:

```
>>> type(b)
<class ‘re.Match’>
```

Gördüğünüz gibi **`b`** değişkeninin tipi bir **eşleşme nesnesi**. İsterseniz bu nesnenin metotlarına bir göz gezdirebiliriz:

```
>>> dir(b)
```

Dikkat ederseniz yukarıda kullandığımız **`group()`** metodu listede görünüyor. Bu metot, **doğrudan doğruya düzenli ifadelerin değil, eşleşme nesnelerinin bir metodudur**. Listedeki öbür metotları da sırası geldiğinde inceleyeceğiz. Şimdi, isterseniz bir **örnek** daha yapıp bu konuyu kapatalım:

```
>>> iddia = “Adana memleketlerin en güzelidir!”
>>> nesne = re.match(“Adana”, iddia)
>>> print(nesne.group())
```

Peki, eşleştirmek istediğimiz düzenli ifade kalıbı bulunamazsa ne olur? Öyle bir durumda yukarıdaki kodlar **hata** verecektir. Hemen bakalım:

```
>>> nesne = re.match("İstanbul", iddia)
>>> print(nesne.group())
```

Hata mesajımız:

```
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'group'
```

Böyle bir hata, yazdığınız bir programın **çökmesine neden olabilir**. O yüzden kodlarımızı şuna benzer bir şekilde yazmamız daha mantıklı olacaktır:

```
>>> nesne = re.match("İstanbul", iddia)
>>> if nesne:
...     print("eşleşen ifade: ", nesne.group())
... else:
...     print("eşleşme başarısız!")
```

Şimdi, isterseniz bu **`match()`** metoduna bir ara verip başka bir metodu inceleyelim.

SEARCH() METODU

Bir önceki bölümde incelediğimiz **`match()`** metodu, karakter dizilerinin sadece en başına bakıyordu. Ama her zaman istediğimiz şey bununla **sınırlı olmayacaktır**. **`match()`** metodunun, karakter dizilerinin sadece başına bakmasını engellemenin yolları olmakla birlikte, düzenli ifadelerin bizim işimizi görecek **çok daha kullanışlı bir metodu** vardır. Önceki bölümde **`dir(re)`** şeklinde gösterdiğimiz listeye tekrar bakarsanız orada **`re`** modülünün **`search()`** adlı bir metodu olduğunu göreceksiniz. İşte bu yazımızda inceleyeceğimiz metot bu **`search()`** metodu olacaktır.

search() metodu ile **match()** metodu arasında **çok önemli bir fark vardır**. **match()** metodu bir karakter dizisinin en başına bakıp bir eşleştirme işlemi yaparken **search()** metodu **karakter dizisinin genelinde bir arama işlemi yapar**. *Yani biri eşleştirir, öbürü arar*. Zaten '**search**' kelimesi de İngilizcede **aramak** anlamına gelir...

Hatırlarsanız, **match()** metodunu anlatırken şöyle bir **örnek** vermiştik:

```
>>> a = "Python güçlü bir dildir"
>>> re.match("güçlü", a)
```

Yukarıdaki kod, karakter dizisinin başında bir eşleşme bulamadığı için bize **None** değeri **döndürüyordu**. Ama eğer aynı işlemi şöyle yaparsak daha farklı bir sonuç elde ederiz:

```
>>> a = "Python güçlü bir dildir"
>>> re.search("güçlü", a)
<re.Match object; span=(7, 12), match='güçlü'>
```

Gördüğünüz gibi **search()** metodu "**güçlü**" kelimesini buldu. Çünkü **search()** metodu, **match()** metodunun aksine bir karakter dizisinin **sadece baş tarafına bakmakla yetinmiyor, karakter dizisinin geneli üzerinde bir arama işlemi gerçekleştiriyor**.

Tıpkı **match()** metodunda olduğu gibi **search()** metodunda da **group()** metodundan faydalanarak bulunan şeyi görüntüleyebiliriz:

```
>>> a = "Python güçlü bir dildir"
>>> b = re.search("güçlü", a)
>>> print(b.group())
"güçlü"
```

Şimdiye kadar hep karakter dizileri üzerinde çalıştık. İsterseniz biraz da **listeler üzerinde örnekler verelim**.

Şöyle bir listemiz olsun:

```
>>> liste = [“elma”, “armut”, “kebap”]
>>> re.search(“kebap”, liste)
```

Ne oldu? Hata aldınız, değil mi? Bu **normal**. Çünkü düzenli ifadeler, **karakter dizileri üzerinde işler**. Bunlar doğrudan listeler üzerinde işlem yapamaz. O yüzden bizim Python’a biraz yardımcı olmamız gerekiyor:

```
>>> for i in liste:
...     b = re.search(“kebap”, i)
...     if b:
...         print(b.group())
...
kebap
```

Eğer arzu ederseniz bu kodları Python’daki ‘**mors**’ adlı **özel bir işleç ile de** yazabilirsiniz.

Dikkatlice bakın:

```
>>> for i in liste:
...     if b := re.search(“kebap”, i):
...             print(b.group())
...
kebap
>>>
```

Burada mors işlecini (**:=**) kullanarak **`b = re.search(“kebap”, i)`** ve **`if b:`** satırlarını **birleştirdik** ve **şöyle bir kod elde ettik**:

```
if b := re.search(“kebap”, i):
```

Mors işlecinin, iki satırda yazılabilebilecek bir kodu **tek satıra indirmek dışında bir avantajı yok**. Dolayısıyla bu işleci kullanıp kullanmamak tamamen sizin tercihinize kalmış.

FINDALL() METODU

Python komut satırında, yani **etkileşimli kabukta**, **`dir(re)`** yazdığımız zaman aldığımız listeye tekrar bakarsak orada bir de **`findall()`** adlı bir metodun olduğunu görürüz. İşte bu bölümde **`findall()`** adlı bu **önemli metodu** incelemeye çalışacağız.

Önce şöyle bir metin alalım elimize:

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

Bu metin içinde geçen bütün "**`Python`**" **kelimelerini bulmak istiyoruz**:

```
print(re.findall("Python", metin))
['Python', 'Python', 'Python', 'Python', 'Python', 'Python']
```

Gördüğünüz gibi metinde geçen bütün "**`Python`**" kelimelerini bir çırpıda **liste** olarak aldık. Aynı işlemi **`search()`** metodunu kullanarak yapmak istersek yolu biraz uzatmamız gerekir:

```
>>> liste = metin.split()
>>> for i in liste:
...     if nesne := re.search("Python", i):
...         print(nesne.group())
```

```
...
Python
Python
Python
Python
Python
Python
```

Bu kodları **mors işleci olmadan** şöyle yazabileceğimizi biliyorsunuz:

```
>>> liste = metin.split()
>>> for i in liste:
...     nesne = re.search("Python", i)
...     if nesne:
...         print(nesne.group())
```

Gördüğünüz gibi metinde geçen bütün "**Python**" kelimelerini **search()** metodunu kullanarak bulmak için öncelikle **metin** adlı karakter dizisini, daha önce karakter dizilerini işlerken gördüğümüz **split()** metodu yardımıyla bir **liste hâline** getiriyoruz. Ardından bu liste üzerinde bir **for** döngüsü kurup **search()** ve **group()** metotlarını kullanarak bütün "**Python**" kelimelerini **ayıklıyoruz**. Eğer karakter dizisini yukarıdaki şekilde **listeye dönüştürmezsek şöyle bir netice alırız**:

```
>>> nesne = re.search("Python", metin)
>>> print(nesne.group())
Python
```

Bu şekilde metinde geçen sadece ilk "**Python**" kelimesini alabiliyoruz. Eğer, doğrudan karakter dizisine, listeye dönüştürmeksizin **for** döngüsü uygulamaya kalkarsak şöyle bir şey olur:

```
>>> for i in a:
...     nesne = re.search("Python", metin)
...     if nesne:
...         print(nesne.group())
```

Gördüğünüz gibi yukarıdaki kod, ekranımızı "**Python**" çıktısıyla dolduruyor. Eğer en sonda **print(nesne.group()** yazmak yerine **print(i)** yazarsanız Python'ın *ne yapmaya çalıştığını ve neden böyle bir çıktı verdiğini anlayabilirsiniz*.

Metakarakterler

Şimdiye kadar düzenli ifadelerle ilgili olarak verdiğimiz örnekler, sizi biraz şaşırtmış olabilir. "*Zor dediğin bunlar mıydı?*" diye düşünmüş olabilirsiniz. Haklısınız, zira zor dediğim, buraya kadar olan kısımda verdiğim örneklerden ibaret değildir. Buraya kadar olan bölümde verdiğim örnekler **işin en temel kısmını gözler önüne sermek içindi**.

Şimdiye kadar olan bölümde mesela, "**python**" karakter dizisiyle **eşleştirme yapmak için** "**python**" kelimesini kullandık. Esasında bu, düzenli ifadelerin **en temel özelliğidir**. Yani "**python**" karakter dizisini bir düzenli ifade sayacak olursak (*ki zaten öyledir*) bu düzenli ifade, **en başta kendisiyle eşleşecektir**. *Bu ne demek?* Şöyle ki: Eğer aradığınız şey "**python**" karakter dizisi ise kullanmanız gereken düzenli ifade de "**python**" olacaktır. Diyoruz ki: "*Düzenli ifadeler en başta kendileriyle eşleşirler*". **Buradan şu anlam çıkıyor**: Demek ki bir de kendileriyle eşleşmeyen düzenli ifadeler var. İşte bu durum, Python'daki düzenli ifadelere kişiliğini kazandıran şeydir. Biraz sonra ne demek istediğimizi daha açık anlayacaksınız. Artık gerçek anlamıyla düzenli ifadelere giriş yapıyoruz!

Öncelikle, elimizde aşağıdaki gibi bir **liste** olduğunu varsayalım:

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

Diyelim ki biz bu liste içinden "**özcan**", "**özkan**" ve "**özhan**" ögelerini **ayıklamak**/**almak** istiyoruz. Bunu yapabilmek için yeni bir bilgiye ihtiyacımız var: **Metakarakterler**.

Metakarakterler; kabaca, programlama dilleri için **özel anlam ifade eden** sembollerdir. Örneğin daha önce gördüğümüz **\n** bir bakıma bir metakarakterdir. Çünkü **\n** sembolü Python için özel bir anlam taşır. Python bu sembolü gördüğü yerde, **yeni bir satıra geçer**. Yukarıda kendisiyle eşleşmeyen karakterler ifadesiyle kastettiğimiz şey de işte bu metakarakterlerdir.

Örneğin "**a**" harfi yalnızca kendisiyle eşleşir. Tıpkı "**istihza**" kelimesinin yalnızca kendisiyle eşleşeceği gibi... Ama mesela **\t** ifadesi **kendisiyle eşleşmez**. Python bu işareti gördüğü yerde **sekme** (*tab*) düğmesine basılmış gibi **tepki verecektir**. İşte düzenli ifadelerde de buna benzer metakarakterlerden yararlanacağız.

Düzenli ifadeler içinde de özel anlam ifade eden pek çok sembol, yani **metakarakter** vardır. Bu metakarakterlerden biri de **[ ]** sembolüdür. Şimdi, yukarıda verdiğimiz listeden "**özcan**", "**özhan**" ve "**özkan**" ögelerini bu sembolden yararlanarak nasıl ayıklayacağımızı görelim:

```
>>> re.search("öz[chk]an", liste)
```

Bu kodu böyle yazmamamız gerektiğini artık biliyoruz. **Aksi hâlde hata alırız**. Çünkü daha önce de dediğimiz gibi düzenli ifadeler karakter dizileri üzerinde işlem yapabilir. **Listeler üzerinde değil**. Dolayısıyla komutumuzu şu şekilde vermemiz gerekiyor:

```
>>> for i in liste:
...     if nesne := re.search("öz[chk]an", i):
...         print(nesne.group())
```

veya mors işlecini kullanmadan:

```
>>> for i in liste:
...     nesne = re.search("öz[chk]an", i)
...     if nesne:
...             print(nesne.group())
```

Gördüğünüz gibi uygun kodları kullanarak; "**özcan**", "**özhan**" ve "**özkan**" ögelerini listeden **kolayca ayıkladık**. Bize bu imkânı veren şey ise **[]** adlı metakarakter oldu. Aslında **[]** metakarakterinin ne işe yaradığını az çok anlamış olmalısınız.

Ama biz yine de şöyle bir bakalım bu metakaraktere:

`[]` adlı metakarakter, yukarıda verdiğimiz listedeki "**öz**" ile başlayıp,"**c**", "**h**" veya "**k**" harflerinden herhangi biri ile devam eden ve "**an**" ile biten bütün ögeleri ayıklıyor. Gelin bununla ilgili bir **örnek** daha yapalım:

```
>>> for i in liste:
... nesne = re.search("es[mr]a", i)
...     if nesne:
...         print(nesne.group())
```

veya:

```
>>> for i in liste:
...     if nesne := re.search("es[mr]a", i):
...         print(nesne.group())
```

Gördüğünüz gibi "**es**" ile başlayıp "**m**" veya "**r**" harflerinden herhangi biriyle devam eden ve sonunda da "**a**" harfi bulunan bütün ögeleri ayıkladık. Bu da bize "**esma**" ve "**esra**" **çıktılarını verdi**...

Dediğimiz gibi metakarakterler programlama dilleri için **özel anlam ifade eden sembollerdir**. "**Normal**" karakterlerden farklı olarak metakarakterlerle karşılaşan bir bilgisayar, **normalden farklı bir tepki verecektir**.

Yukarıda metakarakterlere örnek olarak **\n** ve **\t** **kaçış dizilerini** vermiştik. Örneğin Python'da **print("\n")** gibi bir komut verdiğimizde, Python ekrana **\n** yazdırmak yerine **bir alt satıra geçecektir**. Çünkü **\n** Python için özel bir anlam taşımaktadır. Düzenli ifadelerde de birtakım metakarakterlerin kullanıldığını öğrendik. **Bu metakarakterler, düzenli ifadeleri düzenli ifade yapan şeydir**. Bunlar olmadan düzenli ifadelerle yararlı bir iş yapmak mümkün olmaz.

Bu giriş bölümünde düzenli ifadelerde kullanılan metakarakterlere **örnek** olarak `[]` sembolünü verdik. Herhangi bir düzenli ifade içinde `[]` sembolünü gören Python, doğrudan doğruya bu sembolle eşleşen bir karakter dizisi aramak yerine, **özel bir işlem gerçekleştirecektir**. Yani `[]` sembolü kendisiyle eşleşmeyecektir...

Python'da bulunan temel metakarakterleri **topluca görelim**:

```
[ ] . * + ? { } ^ $ | ( )
```

Doğrudur, yukarıdaki karakterler, çizgi romanlardaki küfürlere benziyor. Endişelenmeyin, biz bu metakarakterleri olabildiğince sindirilebilir hâle getirmek için elimizden gelen çabayı göstereceğiz.

Bu bölümde düzenli ifadelerin zor kısmı olan metakarakterlere, okurlarımızı ürkütmeden, yumuşak bir giriş yapmayı amaçladık. Şimdi, artık metakarakterlerin temelini attığımıza göre üste kat çıkmaya başlayabiliriz.

[ ] (Köşeli Parantez)

`[]` adlı metakaraktere önceki bölümde değinmiştik. Orada verdiğimiz **örnek** şuydu:

```
>>> for i in liste:
...     nesne = re.search("öz[chk]an", i)
...      if nesne:
...         print(nesne.group())
```

Yukarıdaki örnekte, bir liste içinde geçen "**özcan**", "**özhan**" ve "**özkan**" ögelerini ayıklıyoruz. Burada bu üç ögedeki farklı karakterleri ("**c**", "**h**" ve "**k**") köşeli parantez içinde **nasıl belirttiğimize** dikkat edin. Python, köşeli parantez içinde gördüğü bütün karakterleri **tek tek liste ögelerine uyguluyor**. Önce "**öz**" ile başlayan bütün ögeleri alıyor, ardından "**öz**" hecesinden sonra "**c**" harfiyle devam eden ve "**an**" hecesi ile biten ögeyi buluyor. Böylece "**özcan**" ögesini bulmuş oldu. Aynı işlemi, "**öz**" hecesinden sonra "**h**" harfini barındıran ve "**an**" hecesiyle biten ögeye uyguluyor. Bu şekilde ise "**özhan**" ögesini bulmuş oldu. En son hedef ise "**öz**" ile başlayıp "**k**" harfi ile devam eden ve "**an**" ile biten öge. Yani listedeki "**özkan**" ögesi... **En nihayetinde de elimizde** "**özcan**", "**özhan**" ve "**özkan**" **ögeleri kalmış oluyor**.

Bir önceki bölümde yine `[]` metakarakteriyle ilgili olarak şu örneği de vermiştik:

```
>>> for i in liste:
...     nesne = re.search("es[mr]a", i)
...     if nesne:
...         print(nesne.group())
```

Bu örneğin de "**`özcan`**", "**`özhan`**" ve "**`özkan`**" örneğinden bir farkı yok. Burada da Python köşeli parantez içinde gördüğü bütün karakterleri **tek tek liste ögelerine uygulayıp** "`esma`" ve "`esra`" **ögelerini bize veriyor**.

Şimdi bununla ilgili yeni bir **örnek** verelim.

Diyelim ki elimizde şöyle bir **liste** var:

```
>>> a = ["23BH56", "TY76Z", "4Y7UZ", "TYUDZ", "34534"]
```

Mesela biz bu listedeki ögeler içinde, sayıyla başlayanları ayıklayalım. Şimdi, şu kodları dikkatlice inceleyin:

```
>>> for i in a:
...     if re.match("[0-9]", i):
...         print(i)
...
23BH56
4Y7UZ
34534
```

Burada, **parantez içinde kullandığımız ifadeye dikkat edin**. 0 ile 9 arasındaki bütün ögeleri içeren bir karakter dizisi tanımladık. Yani kısaca, içinde herhangi bir sayı barındıran ögeleri kapsama alanımıza aldık. Burada ayrıca **`search()`** yerine **`match()`** metodunu kullandığımıza da dikkat edin. **`match()`** metodunu kullanmamızın nedeni, bu metodun bir karakter dizisinin sadece en başına bakması. Amacımız **sayı ile başlayan bütün ögeleri ayıklamak** olduğuna göre yukarıda yazdığımız kod, liste ögeleri içinde yer alan ve sayı ile başlayan bütün ögeleri **ayıklayacaktır**.

Biz burada Python'a şu emri vermiş oluyoruz:

"**Bana sayı ile başlayan bütün ögeleri bul! Önemli olan bu ögelerin sayıyla başlamasıdır! Sayıyla başlayan bu ögeler ister harfle devam etsin ister başka bir karakterle... Sen yeter ki bana sayı ile başlayan ögeleri bul!**"

Bu emri alan Python, hemen liste ögelerini gözden geçirecek ve bize "**23BH56**", "**4Y7UZ**" ve "**34534**" ögelerini verecektir. Dikkat ederseniz, Python bize **listedeki** "**TY76Z**" ve "**TYUDZ**" **ögelerini vermedi**. Çünkü "**TY76Z**" içinde sayılar olsa da bunlar bizim ölçütümüze uyacak şekilde en başta yer almıyor. "**TYUDZ**" ögesinde ise **tek bir sayı bile yok...**

Şimdi de isterseniz listedeki "**TY76Z**" ögesini nasıl alabileceğimize bakalım:

```
>>> for i in a:
...     if re.match("[A-Z][A-Z][0-9]", i):
...         print(i)
```

Burada dikkat ederseniz düzenli ifademizin başında "**A-Z**" diye bir şey yazdık. Bu ifade "**A**" ile "**Z**" harfleri arasındaki bütün karakterleri temsil ediyor. Biz burada yalnızca büyük harfleri sorguladık. Eğer küçük harfleri sorgulamak isteseydik "**A-Z**" yerine "**a-z**" diyecektik. Düzenli ifademiz içinde geçen birinci "**A-Z**" ifadesi aradığımız karakter dizisi olan "**TY76Z**" içindeki "**T**" harfini, ikinci "**A-Z**" ifadesi "**Y**" harfini, "**0-9**" ifadesi ise "**7**" sayısını temsil ediyor. Karakter dizisi içindeki geri kalan harfler ve sayılar **otomatik olarak eşleştirilecektir**. O yüzden onlar için ayrı bir şey yazmaya gerek yok. Yalnız bu söylediğimiz son şey sizi aldatmasın. Bu otomatik eşleştirme işlemi bizim şu anda karşı karşıya olduğumuz karakter dizisi için geçerlidir. *Farklı nitelikteki karakter dizilerinin söz konusu olduğu başka durumlarda işler böyle yürümeyebilir.* Düzenli ifadeleri başarılı bir şekilde kullanabilmenin **ilk şartı**, **üzerinde işlem yapılacak karakter dizisini tanımaktır**. Bizim örneğimizde yukarıdaki gibi bir düzenli ifade kalıbı oluşturmak işimizi görüyor. Ama başka durumlarda, duruma uygun başka kalıplar yazmak **gerekebilir/gerekecektir**. Dolayısıyla, tek bir düzenli ifade kalıbıyla hayatın geçmeyeceğini unutmamalıyız.

Şimdi, yukarıdaki kodu **search()** ve **group()** metotlarını kullanarak yazmayı deneyin. Elde ettiğiniz sonuçları dikkatlice inceleyin. **match()** ve **search()** metotlarının ne gibi farklılıklara sahip olduğunu kavramaya çalışın.

Bu arada, düzenli ifadelerle ilgili daha fazla şey öğrendiğimizde, yukarıdaki kodu **çok daha sade** bir biçimde yazabileceğiz.

. (NOKTA)

Bir önceki bölümde `[]` adlı metakarakteri incelemiştik. Bu bölümde ise farklı bir metakarakteri inceleyeceğiz. **İnceleyeceğimiz metakarakter**: ".

Bu metakarakter, **yeni satır karakteri hariç bütün karakterleri temsil etmek için kullanılır**. Mesela:

```
>>> for i in liste:
...     nesne = re.match("es.a", i)
...     if nesne:
...         print(nesne.group())
...
esma
esra
```

Gördüğünüz gibi daha önce `[]` metakarakterini kullanarak yazdığımız bir düzenli ifadeyi bu kez **farklı şekilde** yazıyoruz. Unutmayın, **bir düzenli ifade birkaç farklı şekilde yazılabilir**. Biz bunlar içinde *en basit* ve *en anlaşılır* olanını seçmeliyiz. Ayrıca yukarıdaki kodu birkaç farklı şekilde de yazabilirsiniz. Mesela şu yazım da bizim durumumuzda geçerli bir seçenek olacaktır:

```
>>> for i in liste:
...     if re.match("es.a", i):
...         print(i)
```

Tabii ki biz, o anda çözmek durumunda olduğumuz soruna **en uygun** olan seçeneği tercih etmeliyiz...

Yalnız, **unutmamamız gereken şey**, bu '**.**' adlı metakarakterin sadece tek bir karakterin yerini tutuyor olmasıdır. Yani şöyle bir kullanım bize **istediğimiz sonucu vermez**:

```
>>> liste = [“ahmet”,”kemal”, “kamil”, “mehmet”]
>>> for i in liste:
...     if re.match(“.met”, i):
...         print(i)
```

Burada '**.**' sembolü "**ah**" ve "**meh**" hecelerinin yerini tutamaz. '**.**' sembolünün görevi sadece **tek bir karakterin yerini tutmaktır** (*yeni satır karakteri hariç*). Ama biraz sonra öğreneceğimiz metakarakter yardımıyla "**ah**" ve "**meh**" hecelerinin yerini de tutabileceğiz.

'**.**' sembolünü kullanarak bir **örnek** daha yapalım. Bir önceki bölümde verdiğimiz **a** listesini hatırlıyorsunuz:

```
>>> a = [‘23BH56’, ‘TY76Z’, ‘4Y7UZ’, ‘TYUDZ’, ‘34534’]
```

Önce bu listeye **bir öge daha** ekleyelim:

```
>>> a.append(“1agAY54”)
```

Artık elimizde şöyle bir **liste** var:

```
>>> a = [‘23BH56’, ‘TY76Z’, ‘4Y7UZ’, ‘TYUDZ’,
... ‘34534’, “1agAY54”]
```

Şimdi bu listeye şöyle bir **düzenli ifade** uygulayalım:

```
>>> for i in a:
...     if re.match(“.[0-9a-z]”, i):
...         print(i)
...
23BH56
34534
1agAY54
```

Burada yaptığımız şey çok basit. **Şu özelliklere sahip bir karakter dizisi arıyoruz**:

1. Herhangi bir karakter ile başlayacak. Bu karakter **sayı**, **harf** veya **başka bir karakter** olabilir.
2. Ardından bir **sayı** veya **alfabedeki küçük harflerden** herhangi birisi gelecek.
3. Bu ölçütleri karşıladıktan sonra, aradığımız karakter dizisi **herhangi bir karakter ile devam edebilir**.

Yukarıdaki ölçütlere uyan karakter dizilerimiz: "`23BH56`", "`34534`", "`1agAY54`". Yine burada da kendinize göre birtakım değişiklikler yaparak **farklı yazım şekilleri** ve **farklı metotlar** kullanarak ne olup ne bittiğini daha iyi kavrayabilirsiniz. Düzenli ifadeleri gereği gibi anlayabilmek için *bol bol uygulama* yapmamız gerektiğini unutmamalıyız.

* (Yıldız)

Bu metakarakter, kendinden önce gelen bir düzenli ifade kalıbını **sıfır** veya **daha fazla sayıda eşleştirir**. Tanımı biraz karışık olsa da **örnek** yardımıyla bunu da anlayacağız:

```
>>> yeniliste = [“st”, “sat”, “saat”, “saaat”, “falanca”]
>>> for i in yeniliste:
...     if re.match(“sa*t”, i):
...         print(i)
```

Burada `*` sembolü kendinden önce gelen "`a`" karakterini sıfır veya daha fazla sayıda eşleştiriyor. Yani mesela "`st`" içinde sıfır adet "`a`" karakteri var. Dolayısıyla bu karakter yazdığımız **düzenli ifadeyle eşleşiyor**. "`sat`" içinde bir adet "`a`" karakteri var. **Dolayısıyla bu da eşleşiyor**. "`saat`" ve "`saaat`" karakter dizilerinde sırasıyla iki ve üç adet "`a`" karakteri var. Tabii ki bunlar da **yazdığımız düzenli ifadeyle eşleşiyor**. Listemizin en son ögesi olan "`falanca`"da da ilk hecede bir adet "`a`" karakteri var. Ama bu ögedeki sorun, bunun "`s`" **harfiyle başlamaması**. Çünkü biz yazdığımız düzenli ifadede, aradığımız şeyin "`s`" harfi ile başlamasını, sıfır veya daha fazla sayıda "`a`" karakteri ile devam etmesini ve ardından da "`t`" harfinin gelmesini istemiştik. "`falanca`" ögesi **bu koşulları karşılamadığı için süzgecimizin dışında kaldı**.

Burada dikkat edeceğimiz nokta, ***** metakarakterinin **kendinden önce gelen yalnızca bir karakterle ilgileniyor olması**. Yani bizim örneğimizde ***** sembolü sadece "**a**" harfinin sıfır veya daha fazla sayıda *bulunup bulunmamasıyla ilgileniyor*. Bu ilgi, en baştaki "**s**" harfini kapsamıyor. "**s**" harfinin de sıfır veya daha fazla sayıda eşleşmesini istersek düzenli ifademizi "**s*a*t**" veya "**[sa]*t**" biçiminde yazmamız gerekir. Bu iki seçenek içinde "**[sa]*t**" şeklindeki yazımı **tercih etmenizi tavsiye ederim**. Burada, daha önce öğrendiğimiz **[]** metakarakteri yardımıyla "**sa**" harflerini **nasıl grupladığımıza dikkat edin**.

Şimdi, '**.**' metakarakterini anlatırken istediğimiz sonucu alamadığımız listeye dönelim. Orada "**ahmet**" ve "**mehmet**" ögelerini listeden başarıyla ayıklayamamıştık. O durumda bizim **başarısız olmamıza neden olan kullanım şöyleydi**:

```
>>> liste = [“ahmet”, “kemal”, “kamil”, “mehmet”]
>>> for i in liste:
...     if re.match(“.met”, i):
...         print(i)
```

Ama artık elimizde ***** gibi bir araç olduğuna göre şimdi istediğimiz şeyi yapabiliriz. Yapmamız gereken tek şey "**.**" sembolünden sonra ***** sembolünü getirmek:

```
>>> for i in liste:
...     if re.match(“.*met”, i):
...         print(i)
```

Gördüğünüz gibi "**ahmet**" ve "**mehmet**" ögelerini bu kez **başarıyla ayıkladık**. Bunu yapmamızı sağlayan şey de ***** adlı metakarakter oldu... **Burada Python'a şu emri verdik**: "Bana kelime başında herhangi bir karakteri (*"." sembolü herhangi bir karakterin yerini tutuyor*) sıfır veya daha fazla sayıda içeren ve sonu da "**met**" ile biten bütün ögeleri ver!"

Bir önceki örneğimizde "**a**" harfinin sıfır veya daha fazla sayıda *bulunup bulunmamasıyla ilgilenmiştik*. Bu son örneğimizde ise herhangi bir **harfin/karakterin sıfır veya daha fazla sayıda bulunup bulunmamasıyla ilgilendik.**

Dolayısıyla "**.*met**" şeklinde yazdığımız düzenli ifade, "**ahmet**", "**mehmet**", "**muhammet**", "**ismet**", "**kısmet**" ve hatta tek başına "**met**" gibi bütün ögeleri kapsayacaktır. Kısaca ifade etmek gerekirse sonu "**met**" **ile biten her şey** ("**met**" *ifadesinin kendisi de dâhil olmak üzere*) **kapsama alanımıza girecektir**. Bunu günlük hayatta nerede kullanabileceğinizi hemen anlamış olmalısınız. Mesela bir dizin içindeki bütün "**mp3**" dosyalarını bu düzenli ifade yardımıyla listeleyebiliriz:

```
>>> import os
>>> import re
>>> dizin = os.listdir(os.getcwd())
>>> for i in dizin:
...     if re.match(".*mp3", i):
...         print(i)
```

match() metodunu anlattığımız bölümde bu metodun bir karakter dizisinin yalnızca **başlangıcıyla** ilgilendiğini söylemiştik. Mesela o bölümde verdiğimiz şu örneği hatırlıyorsunuzdur:

```
>>> a = "python güçlü bir dildir"
>>> re.match("güçlü", a)
```

Bu örnekte Python bize **çıktı** olarak **None** değerini vermişti. Yani herhangi bir eşleşme bulamamıştı. Çünkü dediğimiz gibi **match()** metodu bir karakter dizisinin **yalnızca en başına bakar**. *Ama geldiğimiz şu noktada artık bu kısıtlamayı nasıl kaldıracağınızı biliyorsunuz*:

```
>>> re.match(".*güçlü", a)
```

Ama **match()** metodunu bu şekilde zorlamak yerine, **performans açısından en doğru yol bu tür işler için search() metodunu kullanmak olacaktır**.

Bunu da geçtiğimize göre artık yeni bir metakarakteri incelemeye başlayabiliriz.

+ (Artı)

Bu metakarakter, bir önceki metakarakterimiz olan * ile benzerdir. Hatırlarsanız, * metakarakteri kendisinden önceki sıfır veya daha fazla sayıda tekrar eden karakterleri ayıklıyordu. + metakarakteri ise **kendisinden önceki bir veya daha fazla sayıda tekrar eden karakterleri ayıklar**. Bildiğiniz gibi önceki örneklerimizden birinde "**ahmet**" ve "**mehmet**" ögelerini şu şekilde ayıklamıştık:

```
>>> for i in liste:
...     if re.match(".*met", i):
...         print(i)
```

Burada "**ahmet**" ve "**mehmet**" dışında "**met**" şeklinde bir öge de **bu düzenli ifadenin kapsamına girecektir**.

Mesela listemiz şöyle olsa idi:

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

Yukarıdaki düzenli ifade bu listedeki "**met**" ögesini de içine alacaktı. Çünkü * adlı metakarakter, **0 sayıda tekrar eden karakterleri de ayıklıyor**. Ama bizim istediğimiz her zaman bu olmayabilir. Bazen de ilgili karakterin **en az bir kez** tekrar etmesini isteriz. Bu durumda yukarıdaki düzenli ifadeyi şu şekilde yazmamız gerekir:

```
>>> for i in liste:
...     if re.match(".+met", i):
...         print(i)
```

Burada şu komutu vermiş olduk: "Bana sonu '**met**' ile biten bütün ögeleri ver! Ama bana '**met**' ögesini yalnız başına verme!"

Aynı işlemi **search()** metodunu kullanarak da yapabileceğimizi biliyorsunuz:

```
>>> for i in liste:
...     nesne = re.search(".+met", i)
...     if nesne:
...         nesne.group()
...
ahmet
mehmet
```

Bir de daha önce verdiğimiz şu örneğe bakalım:

```
>>> yeniliste = ["st", "sat", "saat", "saaat", "falanca"]
>>> for i in yeniliste:
...     if re.match("sa*t", i):
...         print(i)
```

Burada yazdığımız düzenli ifadenin özelliği nedeniyle **st** de kapsama alanı içine giriyordu. Çünkü burada ***** sembolü, "**a**" karakterinin **hiç bulunmadığı durumları da içine alıyor**. Ama eğer biz "**a**" karakteri **en az bir kez geçsin istiyorsak düzenli ifademizi şu şekilde yazmalıyız**:

```
>>> for i in yeniliste:
...     if re.match("sa+t", i):
...         print(i)
```

Hatırlarsanız önceki derslerimizden birinde **köşeli parantezi** anlatırken şöyle bir örnek vermiştik:

```
>>> a = ["23BH56", "TY76Z", "4Y7UZ", "TYUDZ", "34534"]
>>> for i in a:
...     if re.match("[A-Z][A-Z][0-9]", i):
...         print(i)
```

Burada amacımız sadece "**TY76Z**" ögesini almaktı. Dikkat ederseniz; ögenin başındaki "**T**" ve "**Y**" harflerini bulmak için **iki kez** "**[A-Z]**" **yazdık**. Ama artık **+** metakarakterini öğrendiğimize göre aynı işi daha basit bir şekilde yapabiliriz:

```
>>> for i in a:
...     if re.match("[A-Z]+[0-9]", i):
...         print(i)
...
TY76Z
```

Burada "**[A-Z]**" düzenli ifade kalıbını iki kez yazmak yerine **bir kez yazıp yanına da + sembolünü koyarak** bu ifade kalıbının bir veya daha fazla sayıda tekrar etmesini istediğimizi belirttik...

+ sembolünün ne iş yaptığını da anladığımıza göre *artık yeni bir metakarakteri incelemeye başlayabiliriz.*

? (Soru İşareti)

Hatırlarsanız, ***** karakteri *0 ya da daha fazla sayıda eşleşmeleri*; **+** ise *bir ya da daha fazla sayıda eşleşmeleri kapsıyordu*. İşte şimdi göreceğimiz **?** sembolü de **eşleşme sayısının 0 veya 1 olduğu durumları kapsıyor**. Bunu daha iyi anlayabilmek için önceden verdiğimiz şu örneğe bakalım:

```
>>> yeniliste = ["st", "sat", "saat", "saaat", "falanca"]
>>> for i in yeniliste:
...     if re.match("sa*t", i):
...         print(i)
...
st
sat
saat
saaat

>>> for i in yeniliste:
...     if re.match("sa+t", i):
...         print(i)
...
sat
```

```
saat
saaat
```

***** ve **+** sembollerinin hangi karakter dizilerini ayıkladığını görüyoruz. Şimdi de **?** sembolünün ne yaptığına bakalım:

```
>>> for i in yeniliste:
...     if re.match("sa?t", i):
...         print(i)
...
st
sat
```

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

Şimdi, bu metakarakteri kullanarak gerçek hayatta karşımıza çıkabilecek bir **örnek** verelim. Bu metakarakterin tanımına tekrar bakarsak "*olsa da olur olmasa da olur*" diyebileceğimiz durumlar için bu metakarakterin rahatlıkla kullanılabileceğini görürüz. Şöyle bir **örnek** verelim: Diyelim ki bir metin üzerinde arama yapacaksınız. Aradığınız kelime "**uluslararası**":

```
metin = """Uluslararası hukuk, uluslar arası ilişkiler altında bir
disiplindir. Uluslararası ilişkilerin hukuksal boyutunu bilimsel bir
disiplin içinde inceler. Devletlerarası hukuk da denir. Ancak
uluslar arası
ilişkilere yeni aktörlerin girişi bu dalı sadece devletlerarası
olmaktan
çıkarmıştır."""
```

Şimdi, yapmak istediğimiz şey "**uluslararası**" kelimesini bulmak. Ama dikkat ederseniz metin içinde "**uluslararası**" kelimesi aynı zamanda "**uluslar arası**" şeklinde de geçiyor. Bizim bu iki kullanımı da kapsayacak bir düzenli ifade yazmamız gerekecek.

```
>>> nesne = re.findall("[Uu]luslar ?arası", metin)
>>> for i in nesne:
...     print(i)
```

Verdiğimiz **düzenli ifade kalıbını dikkatlice inceleyin**. Bildiğiniz gibi **?** metakarakteri, **kendinden önce gelen karakterin** (*düzenli ifade kalıbını*) **0 veya 1 kez geçtiği durumları arıyor**. Burada **?** sembolünü " " karakterinden, yani boşluk karakterinden sonra kullandık. Dolayısıyla **boşluk karakterinin 0 veya 1 kez geçtiği durumları hedefledik**. Bu şekilde hem "**uluslar arası**" hem de "**uluslararası**" kelimesini **ayıklamış olduk**. Düzenli ifademizde ayrıca şöyle bir şey daha yazdık: "**[Uu]**". Bu da gerekiyor. Çünkü metnimiz içinde "**uluslararası**" kelimesinin **büyük harfle başladığı yerler de var**... Bildiğiniz gibi "**uluslar**" ve "**Uluslar**" kelimeleri **asla aynı değildir**. Dolayısıyla hem "**u**" harfini hem de "**U**" harfini bulmak için daha önce öğrendiğimiz **[]** metakarakterini kullanıyoruz.

NOT Bu metin http://tr.wikipedia.org/wiki/Uluslararas%C4%B1_hukuk adresinden alınıp üzerinde ufak değişiklikler yapılmıştır.

{ } (Küme Parantezi)

{} adlı metakarakterimiz yardımıyla **bir eşleşmeden kaç adet istediğimizi belirtebiliyoruz**. Yine aynı örnek üzerinden gidelim:

```
>>> for i in yeniliste:
...     if re.match("sa{3}t", i):
...         print(i)
...
saaat
```

Burada "**a**" karakterinin **3 kez tekrar etmesini** istediğimizi belirttik. Python da bu emrimizi hemen yerine getirdi.

Bu metakarakterin ilginç bir özelliği daha vardır. **Küme içinde iki farklı sayı yazarak bir karakterin en az ve en çok kaç kez tekrar etmesini istediğimizi belirtebiliriz**. Örneğin:

```
>>> for i in yeniliste:
... if re.match("sa{0,3}t", i):
...     print(i)
...
st
sat
saat
saaat
```

"**sa{0,3}t**" ifadesiyle, "**a**" harfinin **en az 0 kez**, **en çok da 3 kez** tekrar etmesini istediğimiz söyledik. Dolayısıyla, "**a**" harfinin 0, 1, 2 ve 3 kez tekrar ettiği durumlar **ayıklanmış oldu**. Bu sayı çiftlerini değiştirerek daha farklı sonuçlar elde edebilirsiniz. Ayrıca hangi sayı çiftinin daha önce öğrendiğimiz **?** metakarakteriyle aynı işi yaptığını *bulmaya çalışın*...

^ (ŞAPKA)

^ sembolünün iki işlevi var. **Birinci işlevi**, **bir karakter dizisinin en başındaki veriyi sorgulamaktır**. Yani aslında **match()** metodunun varsayılan olarak yerine getirdiği işlevi bu metakarakter yardımıyla açıkça belirterek yerine getirebiliyoruz.

Şu örneğe bakalım:

```
>>> a = ['23BH56', 'TY76Z', '4Y7UZ', 'TYUDZ',
... '34534', '1agAY54']

>>> for i in a:
...     if re.search("[A-Z]+[0-9]", i):
...         print(i)
...
23BH56
TY76Z
4Y7UZ
1agAY54
```

Bir de şuna bakalım:

```
>>> for i in a:
... nesne = re.search("[A-Z]+[0-9]", i)
...     if nesne:
...         print(nesne.group())
...
BH5
TY7
Y7
AY5
```

Dikkat ederseniz, şu son verdiğimiz kod oldukça **hassas bir çıktı** verdi bize. Çıktıdaki bütün değerler, aynen düzenli ifademizde belirttiğimiz gibi yan yana bir veya daha fazla harf içeriyor ve sonra da bir sayı ile devam ediyor. Bu farklılığın nedeni, **ilk kodlarda `print(i)` ifadesini kullanmamız**. Bu durumun çıktılarımızı nasıl değiştirdiğine dikkat edin. Bir de şu örneğe bakalım:

```
>>> for i in a:
... if re.match("[A-Z]+[0-9]", i):
...     print(i)
...
TY76Z
```

Burada sadece "**`TY76Z`**" çıktısını almamızın nedeni, **`match()` metodunun karakter dizilerinin en başına bakıyor olması**.

Aynı etkiyi **`search()`** metoduyla da elde etmek için başlıkta geçen **`^`** (*şapka*) sembolünden yararlanacağız:

```
>>> for i in a:
... nesne = re.search("^[A-Z]+[0-9]", i)
... if nesne:
...     print(nesne.group())
...
TY7
```

Gördüğünüz gibi ^ metakarakteri **`search()`** metodunun, **karakter dizilerinin sadece en başına bakmasını sağladı**. O yüzden de bize sadece "**TY7**" çıktısını verdi. Hatırlarsanız aynı kodu, **şapkasız olarak** şu şekilde kullanmıştık yukarıda:

```
>>> for i in a:
... nesne = re.search("[A-Z]+[0-9]", i)
... if nesne:
...     print(nesne.group())
...
BH5
TY7
Y7
AY5
```

Gördüğünüz gibi **şapka sembolü olmadığında** **`search()`** metodu karakter dizisinin başına bakmakla yetinmiyor, aynı zamanda **karakter dizisinin tamamını tarıyor**. Biz yukarıdaki koda bir ^ sembolü ekleyerek metodumuzun sadece karakter dizisinin en başına bakmasını istedik. O da emrimize sadakatle uydu. Burada dikkatimizi çekmesi gereken başka bir nokta da **`search()`** metodundaki **çıktının kırpılmış olması**. Dikkat ettiyseniz **`search()`** metodu bize **ögenin tamamını** vermedi. Ögelerin yalnızca "**`[A-Z]+[0-9]`**" kalıbına uyan kısımlarını kesip attı önümüze. Çünkü biz ona tersini söylemedik. Eğer ögelerin tamamını istiyorsak bunu açık açık belirtmemiz gerekir:

```
>>> for i in a:
...     nesne = re.search("[A-Z]+[0-9].*", i)
...     if nesne:
...         print(nesne.group())
...
BH56
TY76Z
Y7UZ
AY54
```

Veya metodumuzun **karakter dizisinin sadece en başına bakmasını istersek**:

```
>>> for i in a:
...     nesne = re.search("^[A-Z]+[0-9].*", i)
...     if nesne:
...         print(nesne.group())
...
TY76Z
```

Bu kodlarda düzenli ifade kalıbının sonuna **`.*`** sembolünü eklediğimize dikkat edin. **Böylelikle metodumuzun**, **sonu herhangi bir şekilde biten ögeleri bize vermesini sağladık**...

Başta da söylediğimiz gibi **`^`** metakarakterinin, karakter dizilerinin en başına demir atmak dışında başka bir görevi daha vardır; "**hariç**" **anlamına gelmek**... Bu görevini, sadece **`[]`** metakarakterinin içinde kullanıldığı zaman yerine getirir. Bunu bir örnekle görelim. Yukarıdaki listemiz üzerinde öyle bir **süzgeç** uygulayalım ki "**`1agAY54`**" ögesi çıktılarımız arasında görünmesin... Bu ögeyi avlayabilmek için kullanmamız gereken **düzenli ifade** şöyle olacaktır: **`[0-9A-Z][^a-z]+`**

```
>>> for i in a:
...     nesne = re.match("[0-9A-Z][^a-z]+", i)
...     if nesne:
...         print(nesne.group())
```

Burada şu ölçütlere sahip bir öge arıyoruz:

- Aradığımız öge bir sayı veya büyük harf ile başlamalı.
- En baştaki sayı veya büyük harften sonra **küçük harf gelmemeli** (*Bu ölçütü ^ işareti sağlıyor*).
- Üstelik bu küçük harf gelmeme durumu **bir veya daha fazla sayıda tekrar etmeli**. Yani baştaki sayı veya büyük harften sonra kaç tane olursa olsun **asla küçük harf gelmemel**i (*Bu ölçütü de + işareti sağlıyor*).

Bu ölçütlere uymayan tek öge "**`1agAY54`**" olacaktır. Dolayısıyla bu öge çıktıda görünmeyecek...

Burada ^ işaretinin nasıl kullanıldığına ve küçük harfleri nasıl dışarıda bıraktığına dikkat edin. Unutmayalım, bu ^ işaretinin **hariç anlamı sadece `[]` metakarakterinin içinde kullanıldığı zaman geçerlidir**.

$ (Dolar)

Bir önceki bölümde ^ işaretinin, karakter dizilerinin en başına demir attığını söylemiştik. Yani bu sembol **arama/eşleştirme** işleminin karakter dizisinin en başından başlamasını sağlıyordu. Bu sembol bir bakıma karakter dizilerinin nasıl başlayacağını belirliyordu. İşte şimdi, göreceğimiz dolar işareti de (**$**) **karakter dizilerinin nasıl biteceğini belirliyor**. Bu soyut açıklamaları somut bir örnekle bağlayalım:

```
>>> liste = [“at”, “katkı”, “fakat”, “atkı”, “rahat”,
... “mat”, “yat”, “sat”, “satılık”, “katılım”]
```

Gördüğünüz gibi elimizde **10 ögelik bir liste** var. Diyelim ki biz bu listeden “**at**” hecesiyle biten kelimeleri ayıklamak istiyoruz:

```
>>> for i in liste:
...     if re.search(“at$”, i):
...         print(i)
...
at
fakat
rahat
mat
yat
sat
```

Burada **$** metakarakteri sayesinde, aradığımız karakter dizisinin **nasıl bitmesi gerektiğini belirleyebildik**.

Eğer "**at**" ile başlayan bütün ögeleri ayıklamak isteseydik, ne yapmamız gerektiğini biliyorsunuz:

```
>>> for i in liste:
...     if re.search("^at", i):
...         print i
...
at
atkı
```

Gördüğünüz gibi **^** işareti bir karakter dizisinin nasıl başlayacağını belirlerken **$** işareti **aynı karakter dizisinin nasıl biteceğini belirliyor**. Hatta istersek bu metakarakterleri birlikte de kullanabiliriz:

```
>>> for i in liste:
...     if re.search("^at$", i):
...         print(i)
...
at
```

Sonuç tam da beklediğimiz gibi oldu. Verdiğimiz düzenli ifade kalıbı ile "**at**" ile başlayan ve aynı şekilde biten karakter dizilerini ayıkladık. Bu da bize "**at**" çıktısını verdi.

\ (Ters Bölü)

Bu işaret **bildiğimiz kaçış dizisidir**. *Peki, burada ne işi var?* Şimdiye kadar öğrendiğimiz konulardan gördüğünüz gibi Python'daki düzenli ifadeler açısından **özel anlam taşıyan bir takım semboller/metakarakterler var**. Bunlar **kendileriyle eşleşmiyorlar**. Yani bir karakter dizisi içinde bu sembolleri arıyorsak bunların taşıdıkları özel anlam yüzünden bu sembolleri ayıklamak hemencecik mümkün olmayacaktır. Yani biz "**$**" sembolünü arıyor olsak bunu Python'a nasıl anlatacağız? Çünkü bu sembolü yazdığımız zaman, **Python bunu farklı algılıyor**. Lafı dolandırmadan hemen bir örnek verelim...

Diyelim ki elimizde şöyle bir liste var:

```
>>> liste = ["10$", "25€", "20$", "10TL", "25£"]
```

Amacımız bu listedeki dolarlı değerleri ayıklamaksa ne yapacağız? **Şunu deneyelim önce**:

```
>>> for i in liste:
...     if re.match("[0-9]+$", i):
...         print(i)
```

Python, **$** işaretinin özel anlamından dolayı, bizim sayıyla biten bir karakter dizisi aradığımızı zannedecek, dolayısıyla da herhangi bir **çıktı vermeyecektir**. Çünkü listemizde sayıyla biten bir karakter dizisi yok. *Peki, biz ne yapacağız?* İşte bu noktada \ metakarakteri devreye girecek... Hemen bakalım:

```
>>> for i in liste:
...     if re.match("[0-9]+\$", i):
...         print(i)
...
10$
20$
```

Gördüğünüz gibi \ sembolünü kullanarak **$** işaretinin **özel anlamından kaçtık**. Bu metakarakteri de kısaca anlattığımıza göre *yeni bir metakarakterle yolumuza devam edebiliriz...*

| (Dik Çizgi)

Bu metakarakter, **birden fazla düzenli ifade kalıbını birlikte eşleştirmemizi sağlar**. *Bu ne demek?* Hemen görelim:

```
>>> liste = ["at", "katkı", "fakat", "atkı", "rahat",
... "mat", "yat", "sat", "satılık", "katılım"]
>>> for i in liste:
...     if re.search("^at|at$", i):
...         print(i)
...
at
fakat
atkı
rahat
mat
yat
sat
```

Gördüğünüz gibi **|** metakarakterini kullanarak başta ve sonda "**at**" hecesini içeren kelimeleri **ayıkladık**. Aynı şekilde bu metakarakter yardımıyla, mesela renkleri içeren bir listeden **belli renkleri de ayıklayabiliriz**...

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

Sırada son metakarakterimiz olan **( )** var...

( ) (Parantez)

Bu metakarakter yardımıyla **düzenli ifade kalıplarını gruplayacağız**. Bu metakarakter bizim **bir karakter dizisinin istediğimiz kısımlarını ayıklamamızda çok büyük kolaylıklar sağlayacak**. Ama bu metakarakteri şimdi değil, birazdan **`group()`** metodunu incelerken ele alacağız.

Eşleşme Nesnelerinin Metotları

group() Metodu

Bu bölümde doğrudan düzenli ifadelerin değil, ama **düzenli ifadeler kullanılarak üretilen eşleşme nesnelerinin bir metodu olan `group()`** metodundan **bahsedeceğiz**. Esasında biz bu metodu önceki bölümlerde de kullanmıştık. Ama burada bu metoda biraz *daha ayrıntılı* olarak bakacağız.

Daha önceki bölümlerden hatırlayacağınız gibi bu metot, düzenli ifadeleri kullanarak eşleştirdiğimiz karakter dizilerini görme imkânı sağlıyordu. Bu bölümde bu metodu **( )** metakarakteri yardımıyla **daha verimli** bir şekilde kullanacağız. İsterseniz ilk olarak şöyle basit bir **örnek** verelim:

```
>>> a = "python bir programlama dilidir"
>>> nesne = re.search("(python) (bir) (programlama) (dilidir)", a)
>>> print(nesne.group())
python bir programlama dilidir
```

Burada düzenli ifade kalıbımızı **nasıl grupladığımıza dikkat edin**. **`print(nesne.group())`** komutunu verdiğimizde, eşleşen karakter dizileri ekrana döküldü. Şimdi, bu grupladığımız bölümlere tek tek erişelim:

```
>>> print(nesne.group(0))
python bir programlama dilidir
```

Gördüğünüz gibi "**`0`**" indeksi, **eşleşen karakter dizisinin tamamını** veriyor.

Bir de şuna bakalım:

```
>>> print(nesne.group(1))
python
```

Burada, **1 numaralı grubun ögesi olan** "`python`"ı **aldık**. Gerisinin nasıl olacağını tahmin edebilirsiniz:

```
>>> print(nesne.group(2))
bir

>>> print(nesne.group(3))
programlama

>>> print(nesne.group(4))
dilidir
```

Gelin, isterseniz bununla ilgili **daha kapsamlı** ve **daha anlamlı** bir **örnek** verelim.

Diyelim ki elimizde şöyle bir liste var:

```
li  = ['a href="kitap_hakkinda.html">Bu Kitap Hakkında</a>',
       'a href="python_hakkinda.html">Python Hakkında</a>',
       'a href="python_hakkinda.html">Python Nedir?</a>',
       'a href="kurulum.html">Python Nasıl Kurulur?</a>',
       'a href="surum.html">Kurulu Python Sürümü</a>']
```

Biz bu listedeki ögeleri şu görünümle almak istiyoruz:

```
Başlık: Bu Kitap Hakkında          Bağlantı: kitap_hakkinda.html
Başlık: Python Hakkında            Bağlantı: python_hakkinda.html
Başlık: Python Nedir?              Bağlantı: python_hakkinda.html
Başlık: Python Nasıl Kurulur?      Bağlantı: kurulum.html
Başlık: Kurulu Python Sürümü       Bağlantı: surum.html
```

İşte bunun için **()** metakarakterinden ve **group()** metodundan yararlanabiliriz.

Dikkatlice inceleyin:

```
import re

li  = ['a href="kitap_hakkinda.html">Bu Kitap Hakkında</a>',
       'a href="python_hakkinda.html">Python Hakkında</a>',
       'a href="python_hakkinda.html">Python Nedir?</a>',
       'a href="kurulum.html">Python Nasıl Kurulur?</a>',
       'a href="surum.html">Kurulu Python Sürümü</a>']

for öğe in li:
    # burada düzenli ifadenin .+html ve .+ kısımlarını ()
    # metakarakteri yardımıyla grupluyoruz.
    nesne = re.search('a href="(.+html)">(.+)</a>', öğe)
    if nesne:
        # grupladığımız ögeleri group() metoduyla alıp f'li
        # karakter dizisinin uygun yerlerine yerleştiriyoruz
        print(f"Başlık: {nesne.group(2).ljust(25)} Bağlantı: {nesne.
group(1)}")
```

Burada liste ögelerinin istediğimiz kısımlarını **()** metakarakteri yardımıyla **nasıl grupladığımıza özellikle dikkat edin**. Bu metodun bize ileride ne büyük kolaylıklar sağlayacağını az çok tahmin ediyorsunuzdur. İsterseniz kullanabileceğimiz metotları tekrar listeleyelim:

```
>>> dir(nesne)
```

Bu listede **group()** dışında bir de **groups()** adlı bir metodun olduğunu görüyoruz. Şimdi, bunun ne iş yaptığına bakalım.

GROUPS() METODU

Bu metot, bize kullanabileceğimiz bütün grupları **bir demet hâlinde** sunar:

```
>>> print(nesne.groups())
('python', 'bir', 'programlama', 'dilidir')
```

Bu metodu yukarıda yaptığımız örneğe de uygulayalım:

```
import re

li  = ['a href="kitap_hakkinda.html">Bu Kitap Hakkında</a>',
       'a href="python_hakkinda.html">Python Hakkında</a>',
       'a href="python_hakkinda.html">Python Nedir?</a>',
       'a href="kurulum.html">Python Nasıl Kurulur?</a>',
       'a href="surum.html">Kurulu Python Sürümü</a>']

for öğe in li:
    nesne = re.search('a href="(.+html)">(.+)</a>', öğe)
    print(nesne.groups())
```

Böylece eşleştirme nesnelerinin en sık kullanılan iki metodunu görmüş olduk. Bunları daha sonraki örneklerimizde de **bol bol kullanacağız**. *O yüzden şimdilik bu konuya ara verelim*.

Özel Diziler

Düzenli ifadeler içinde metakarakterler dışında, özel anlamlar taşıyan bazı başka ifadeler de vardır. **Bu bölümde bu özel dizileri inceleyeceğiz**: Boşluk karakterinin yerini tutan özel dizi: **\s**

Bu sembol, **bir karakter dizisi içinde geçen boşlukları yakalamak** için kullanılır. Örneğin:

```
>>> a = [“5 Ocak”, “27Mart”, “4 Ekim”, “Nisan 3”]
>>> for i in a:
...     nesne = re.search(“[0-9]\s[A-Za-z]+”, i)
...     if nesne:
...         print(nesne.group())
...
5 Ocak
4 Ekim
```

Yukarıdaki örnekte, bir sayı ile başlayan ardından bir adet boşluk karakteri içeren, sonra da bir **büyük** veya **küçük harfle devam eden karakter dizilerini ayıkladık**. Burada boşluk karakterini **\s** simgesi ile gösterdiğimize dikkat edin.

Ondalık Sayıların Yerini Tutan Özel Dizi: \d

Bu sembol, **bir karakter dizisi içinde geçen ondalık sayıları eşleştirmek için** kullanılır. Buraya kadar olan örneklerde bu işlevi yerine getirmek için `[0-9]` ifadesinden yararlanıyorduk. Şimdi, artık aynı işlevi daha kısa yoldan, **\d** dizisi ile yerine getirebiliriz. İsterseniz yine yukarıdaki örnekten gidelim:

```
>>> a = [“5 Ocak”, “27Mart”, “4 Ekim”, “Nisan 3”]
>>> for i in a:
...     nesne = re.search(“\d\s[A-Za-z]+”,i)
...     if nesne:
...         print(nesne.group())
...
5 Ocak
4 Ekim
```

Burada `[0-9]` yerine **\d** yerleştirerek daha kısa yoldan sonuca vardık.

Alfanümerik Karakterlerin Yerini Tutan Özel Dizi: \w

Bu sembol, bir karakter dizisi içinde geçen **alfanümerik karakterleri** ve **buna ek olarak** "_" **karakterini bulmak için** kullanılır. Şu örneğe bakalım:

```
>>> a = "abc123_$%+"
>>> print(re.search("\w*", a).group())
abc123_
```

`\w` özel dizisinin hangi karakterleri eşlediğine dikkat edin. **Bu özel dizi, şu ifadeyle aynı anlama gelir**:

```
[A-Za-z0-9_]
```

Düzenli ifadeler içindeki özel diziler genel olarak bunlardan ibarettir. Ama bir de bunların büyük harfli versiyonları vardır ki önemli oldukları için onları da inceleyeceğiz.

Gördüğünüz gibi;

1. `\s` özel dizisi **boşluk karakterlerini** avlıyor.
2. `\d` özel dizisi **ondalık sayıları** avlıyor.
3. `\w` özel dizisi **alfanümerik karakterleri** ve "_" karakterini avlıyor.

Dedik ki bir de bunların **büyük harfli versiyonları** vardır. İşte bu büyük harfli versiyonlar da yukarıdaki dizilerin yaptığı işin **tam tersini** yapar. Yani:

1. `\S` özel dizisi **boşluk olmayan** karakterleri avlar.
2. `\D` özel dizisi **ondalık sayı olmayan** karakterleri avlar. Yani `[^0-9]` ile eş değerdir.
3. `\W` özel dizisi **alfanümerik olmayan karakterleri** ve "_" olmayan karakterleri avlar. Yani [^A-Za-z0-9_] ile eşdeğerdir.

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

```
>>> a = [“5 Ocak”, “27Mart”, “4 Ekim”, “Nisan 3”]
>>> for i in a:
...     nesne = re.search(“\d+\S\w+”,i)
...     if nesne:
...         print(nesne.group())
...
27Mart
```

Burada **\S** özel dizisinin listede belirtilen konumda **boşluk içermeyen ögeyi nasıl bulduğuna dikkat edin**.

Şimdi, bu özel diziler için genel bir **örnek** verip konuyu kapatalım...

Bilgisayarımızda şu bilgileri içeren **adres.txt** adlı bir dosya olduğunu varsayıyoruz:

```
esra : istinye 05331233445
esma : levent 05322134344
sevgi : dudullu 05354445434
kemal : sanayi 05425455555
osman : tahtakale 02124334444
metin : taksim 02124344332
kezban : caddebostan 02163222122
```

Amacımız bu dosyada yer alan isim ve telefon numaralarını isim **>** telefon numarası şeklinde almak:

```
import re

dosya = open(“adres.txt”, encoding=”utf-8”)

for i in dosya.readlines():
    if nesne := re.search(“(\w+)\s+:\s(\w+)\s+(\d+)”, i):
        print(f”{nesne.group(1).ljust(10)} > {nesne.group(3)}”)
```

Burada formülümüz şu şekilde: “**Bir veya daha fazla karakter**” **+** “**bir veya daha fazla boşluk**” **+** “**’:’ işareti**” **+** “**bir adet boşluk**” **+** “**bir veya daha fazla sayı**”.

Düzenli İfadelerin Derlenmesi

compile() Metodu

En başta da söylediğimiz gibi düzenli ifadeler karakter dizilerine göre *biraz daha yavaş çalışırlar*. Ancak düzenli ifadelerin işleyişini **hızlandırmanın da bazı yolları vardır**. Bu yollardan biri de **`compile()`** metodunu kullanmaktır.

'**`compile`**' kelimesi İngilizcede **derlemek** anlamına gelir. İşte biz de bu **`compile()`** metodu yardımıyla düzenli ifade kalıplarımızı kullanmadan önce **derleyerek daha hızlı çalışmalarını sağlayacağız**. Küçük boyutlu projelerde **`compile()`** metodu pek hissedilir bir fark yapmasa da özellikle büyük çaplı programlarda bu metodu kullanmak oldukça faydalı olacaktır.

Basit bir **örnekle** başlayalım:

```
>>> import re
>>> liste = [“Python3.5”, “Python3.6”, “Python3.7”, “Python3.8”,
“Java”]
>>> derli = re.compile(“[A-Za-z]+[0-9]\.[0-9]”)
>>> for i in liste:
...     if nesne := derli.search(i):
...         print(nesne.group())
...
Python3.5
Python3.6
Python3.7
Python3.8
```

Burada öncelikle **düzenli ifade kalıbımızı derledik**. Derleme işlemini nasıl yaptığımıza dikkat edin. Derlenecek düzenli ifade kalıbını **`compile()`** metodunda **parantez içinde** belirtiyoruz. Daha sonra **`search()`** metodunu kullanırken ise **`re.search()`** demek yerine, **`derli.search()`** şeklinde bir ifade kullanıyoruz. Ayrıca dikkat ederseniz **`derli.search()`** kullanımında parantez içinde **sadece eşleşecek karakter dizisini kullandık** (**`i`**). Eğer derleme işlemi yapmamış olsaydık, hem bu karakter dizisini hem de düzenli ifade kalıbını yan yana kullanmamız gerekecektir. Ama düzenli ifade kalıbımızı yukarıda derleme işlemi esnasında belirttiğimiz için bu kalıbı **ikinci kez yazmamıza gerek kalmadı**.

Ayrıca burada kullandığımız **düzenli ifade kalıbına da dikkat edin**. Nasıl bir şablon oturttuğumuzu anlamaya çalışın. Gördüğünüz gibi liste ögelerinde bulunan "**.**" işaretini eşleştirmek için **düzenli ifade kalıbı içinde** **\.** **ifadesini kullandık**. Çünkü bildiğiniz gibi tek başına "**.**" işaretinin Python açısından özel bir anlamı var. Dolayısıyla bu özel anlamdan kaçmak için \ işaretini de kullanmamız gerekiyor.

compile() İle Derleme Seçenekleri

Bir önceki bölümde **compile()** metodunun ne olduğunu, ne işe yaradığını ve nasıl kullanıldığını görmüştük. Bu bölümde ise **compile** (*derleme*) işlemi sırasında kullanılabilecek seçenekleri anlatacağız.

re.IGNORECASE veya re.I

Bildiğiniz gibi Python'da **büyük-küçük harfler önemlidir**. Yani eğer "**python**" kelimesini arıyorsanız alacağınız çıktılar arasında "**Python**" olmayacaktır. Çünkü "**python**" ve "**Python**" **birbirlerinden farklı iki karakter dizisidir**. İşte **re.IGNORECASE** veya kısaca **re.I** adlı derleme seçenekleri bize büyük-küçük harfe dikkat etmeden arama yapma imkânı sağlar. Hemen bir **örnek** verelim:

```
import re

metin = """Programlama dili, programcının bir bilgisayara ne
yapmasını
istediğini anlatmasının standartlaştırılmış bir yoludur. Programlama
dilleri, programcının bilgisayara hangi veri üzerinde işlem
yapacağını,
verinin nasıl depolanıp iletileceğini, hangi koşullarda hangi
işlemlerin
yapılacağını tam olarak anlatmasını sağlar. Şu ana kadar 2500'den
fazla
programlama dili yapılmıştır. Bunlardan bazıları: Pascal, Basic, C,
C#,
C++, Java, Cobol, Perl, Python, Ada, Fortran, Delphi programlama
dilleridir."""

derli = re.compile("programlama", re.IGNORECASE)
print(derli.findall(metin))
```



Bu metin http://tr.wikipedia.org/wiki/Programlama_dili adresinden alınmıştır.

Bu programı çalıştırdığımızda **şu çıktıyı alıyoruz**:

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

Gördüğünüz gibi metinde geçen hem "**programlama**" kelimesini hem de "**Programlama**" kelimesini **ayıklayabildik**. Bunu yapmamızı sağlayan şey de **re.IGNORECASE** adlı **derleme seçeneği** oldu. Eğer bu seçeneği kullanmasaydık çıktıda yalnızca "**programlama**" kelimesini görürdük. Çünkü aradığımız şey aslında "**programlama**" kelimesi idi. Biz istersek **re.IGNORECASE** yerine kısaca re.I ifadesini de kullanabiliriz. **Aynı anlama gelecektir...**

RE.DOTALL VEYA RE.S

Bildiğiniz gibi metakarakterler arasında yer alan "." sembolü herhangi bir karakterin yerini tutuyordu. Bu metakarakter **bütün karakterlerin yerini tutmak üzere kullanılabilir**. Hatırlarsanız, "." metakarakterini anlatırken bu metakarakterin, yeni satır karakterinin yerini tutmayacağını söylemiştik. Bunu bir örnek yardımıyla görelim. Diyelim ki elimizde şöyle bir **karakter dizisi** var:

```
>>> a = "Ben Python,\nMonty Python"
```

Bu karakter dizisi içinde "**Python**" kelimesini **temel alarak** bir arama yapmak istiyorsak kullanacağımız şu kod istediğimiz şeyi yeterince yerine getiremeyecektir:

```
>>> print(re.search("Python.*", a).group())
```

Bu kod şu **çıktıyı** verecektir:

```
Python,
```

Bunun sebebi, "." metakarakterinin **\n** (*yeni satır*) kaçış dizisini **dikkate almamasıdır**. Bu yüzden bu kaçış dizisinin ötesine geçip orada arama yapmıyor.

Ama şimdi biz ona, **bu yeteneği de kazandıracağız**:

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

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

Düzenli İfadelerle Metin/ Karakter Dizisi Değiştirme İşlemleri

sub() Metodu

Şimdiye kadar hep düzenli ifadeler yoluyla bir karakter dizisini **nasıl eşleştireceğimizi** inceledik. Ama tabii ki düzenli ifadeler, yalnızca bir karakter dizisi bulmakla ilgili değildir. **Bu araç aynı zamanda bir karakter dizisini değiştirmeyi de kapsar**. Bu iş için **temel olarak iki metot** kullanılır. Bunlardan ilki **sub()** metodudur. Bu bölümde **sub()** metodunu inceleyeceğiz. En basit şekliyle **sub()** metodunu şu şekilde kullanabiliriz:

```
>>> a = "Kırmızı başlıklı kız, kırmızı elma dolu sepetiyle \
... anneannesinin evine gidiyormuş!"
>>> derle = re.compile("kırmızı", re.IGNORECASE)
>>> print derle.sub("yeşil", a)
```

Burada karakter dizimiz içinde geçen bütün "**kırmızı**" kelimelerini "**yeşil**" kelimesiyle **değiştirdik**. Bunu yaparken de **re.IGNORECASE** adlı **derleme seçeneğinden** yararlandık.

Elbette **sub()** metoduyla **daha karmaşık işlemler** yapılabilir. Bu noktada şöyle bir hatırlatma yapalım. Bu **sub()** metodu karakter dizilerinin **replace()** metoduna çok benzer. Ama tabii ki **sub()** metodu hem kendi başına **replace()** metodundan çok daha **güçlüdür** hem de **beraber kullanılabilecek derleme seçenekleri sayesinde replace() metodundan çok daha esnektir**. Ama tabii ki eğer yapmak istediğiniz iş **replace()** metoduyla hâlledilebiliyorsa en doğru yol, **replace()** metodunu kullanmaktır.

Şimdi, bu **sub()** metodunu kullanarak biraz daha karmaşık bir işlem yapacağız. Aşağıdaki metne bakalım:

```
metin = """Karadeniz Ereğlisi denince akla ilk olarak kömür ve
demir-çelik
gelir. Kokusu ve tadıyla dünyaya nam salmış meşhur Osmanlı çileği
ise ismini
verdiği festival günleri dışında pek hatırlanmaz. Oysa Çin'den
Arnavutköy'e
oradan da Ereğli'ye getirilen kralların meyvesi çilek, burada
geçirdiği
değişim sonucu tadına doyulmaz bir hal alır. Ereğli'nin havasından
mı
suyundan mı bilinmez, kokusu, tadı bambaşka bir hale dönüşür ve
meşhur
Osmanlı çileği unvanını hak eder. Bu nazik ve aromalı çilekten
yapılan reçel
de likör de bir başka olur. Bu yıl dokuzuncusu düzenlenen
Uluslararası
Osmanlı Çileği Kültür Festivali'nde 36 üretici arasında yetiştirdiği
çileklerle birinci olan Kocaali Köyü'nden Güner Özdemir, yılda bir
ton ürün
alıyor. 60 yaşındaki Özdemir, çileklerinin sırrını yoğun ilgiye ve
içten
duyduğu sevgiye bağlıyor: "Erkekler bahçemize giremez. Koca
ayaklarıyla
ezerler çileklerimizi" Çileği toplamanın zor olduğunu söyleyen Ayşe
Özhan da
çocukluğundan bu yana çilek bahçesinde çalışıyor. Her sabah 04.00'te
kalkan
Özhan, çileklerini özenle suluyor. Kasım başında ektiği çilek
fideleri
haziran başında meyve veriyor."""
```



Bu metin http://www.radikal.com.tr/turkiye/cilegi-osmanli-yapanlar-635005/ adresinden alınmıştır.

Gelin, bu metin içinde geçen "**çilek**" kelimelerini "**erik**" kelimesi ile değiştirelim. Ama bunu yaparken metin içinde "**çilek**" kelimesinin "**Çilek**" şeklinde de geçtiğine dikkat edelim. Ayrıca Türkçe kuralları gereği bu "**çilek**" kelimesinin bazı yerlerde **ünsüz yumuşamasına uğrayarak** "**çileğ-**" **şekline dönüştüğünü de unutmayalım**.

Bu metin içinde geçen "**çilek**" kelimelerini "**erik**"le **değiştirmek için** birkaç yol kullanabilirsiniz. Birinci yolda, her değişiklik için ayrı bir **düzenli ifade** oluşturulabilir. Ancak bu yolun **dezavantajı**, metnin de **birkaç kez kopyalanmasını** gerektirmesidir. Çünkü ilk düzenli ifade oluşturulup buna göre metinde bir değişiklik yapıldıktan sonra, ilk değişiklikleri içeren metnin, farklı bir metin olarak kopyalanması gerekir (*metin2 gibi...*). Ardından ikinci değişiklik yapılacağı zaman, bu değişikliğin metin2 üzerinden yapılması gerekir. Aynı şekilde bu metin de mesela, **metin3** şeklinde **tekrar kopyalanmalıdır**. Bundan sonraki yeni bir değişiklik de bu metin3 üzerinden yapılacaktır... Bu durum bu şekilde uzar gider... **Metni tekrar tekrar kopyalamak yerine, düzenli ifadeleri kullanarak şöyle bir çözüm de üretebiliriz**:

```
import re

derle = re.compile("çile[kğ]", re.IGNORECASE)

def değiştir(nesne):
    a = {"çileğ":"eriğ", "Çileğ":"Eriğ",
         "Çilek":"Erik", "çilek":"erik"}
    b = nesne.group().split()

    for i in b:
        return a[i]

print(derle.sub(değiştir, metin))
```

Gördüğünüz gibi **sub()** metodu, argüman olarak bir fonksiyon da alabiliyor. Yukarıdaki kodlar biraz karışık görünmüş olabilir. **Tek tek açıklayalım**.

Öncelikle şu satıra bakalım:

```
derle = re.compile("çile[kğ]", re.IGNORECASE)
```

Burada amacımız, metin içinde geçen "**çilek**" ve "**çileğ**" kelimelerini bulmak. Neden "**çile[kğ]**"? Çünkü "**çilek**" kelimesi bir sesli harften önce geldiğinde sonundaki "**k**" harfi "**ğ**"ye dönüşüyor. **Bu seçenekli yapıyı**, d**aha önceki bölümlerde gördüğümüz `[]` adlı metakarakter yardımıyla oluşturduk**. Düzenli ifade kalıbımızın hem büyük harfleri hem de küçük harfleri **aynı anda bulması için `re.IGNORECASE` seçeneğinden yararlandık**.

Şimdi de şu satırlara bakalım:

```
def değiştir(nesne):
    a = {"çileğ":"eriğ", "Çileğ":"Eriğ",
         "Çilek":"Erik", "çilek":"erik"}
    b = nesne.group().split()

    for i in b:
        return a[i]
```

Burada, daha sonra **`sub()`** metodu içinde kullanacağımız fonksiyonu yazıyoruz. Fonksiyonu, **`def değiştir(nesne)`** şeklinde tanımladık. Burada **`nesne`** adlı bir argüman kullanmamızın nedeni, fonksiyon içinde **`group()`** metodunu kullanacak olmamız. Bu metodu fonksiyon içinde **`nesne`** adlı argümana bağlayacağız. Bu fonksiyon, daha sonra yazacağımız **`sub()`** metodu tarafından çağrıldığında, yaptığımız arama işlemi sonucunda ortaya çıkan **eşleşme nesnesi fonksiyona atanacaktır** (*eşleşme nesnesinin ne demek olduğunu ilk bölümlerden hatırlıyorsunuz*). İşte **`nesne`** adlı bir **argüman** kullanmamızın nedeni de eşleşme nesnelerinin bir metodu olan **`group()`** metodunu fonksiyon içinde kullanabilmek...

Bir sonraki satırda bir adet **sözlük** görüyoruz:

```
a = {"çileğ":"eriğ", "Çileğ":"Eriğ",
     "Çilek":"Erik", "çilek":"erik"}
```

Bu sözlüğü oluşturmamızın nedeni, **metin içinde geçen bütün** "**çilek**" **kelimelerini tek bir** "**erik**" **kelimesiyle değiştiremeyecek olmamız**. Çünkü "**çilek**" kelimesi metin içinde pek çok farklı biçimde geçiyor. Başta da dediğimiz gibi yukarıdaki yol yerine metni **birkaç kez kopyalayarak** ve **her defasında bir değişiklik yaparak da sorunu çözebilirsiniz**. (*Mesela önce "çilek" kelimelerini bulup bunları* "**erik**" *ile değiştirirsiniz. Daha sonra* "**çileğ**" *kelimelerini arayıp bunları* "**erig**" *ile değiştirirsiniz vb.*) Ama metni tekrar tekrar oluşturmak pek **performanslı bir yöntem olmayacaktır**. Bizim şimdi kullandığımız yöntem, **metin kopyalama zorunluluğunu ortadan kaldırıyor**. Bu sözlük içinde "**çilek**" kelimesinin alacağı şekilleri sözlük içinde birer **anahtar** olarak "**erik**" kelimesinin alacağı şekilleri ise birer **değer** olarak belirliyoruz.

Sonraki satırda **iki metot birden var**:

```
b = nesne.group().split()
```

Burada, fonksiyonumuzun argümanı olarak vazife gören, **eşleşme nesnesine ait metotlardan biri olan** **group()** metodunu kullanıyoruz. Böylece **derle = re.compile("çile[kğ]", re.IGNORECASE)** satırı yardımıyla metin içinde bulduğumuz bütün "**çilek**" ve çeşnilerini alıyoruz. Karakter dizilerinin **split()** metodunu kullanmamızın nedeni ise **group()** metodunun verdiği çıktıyı **liste hâline** getirip daha kolay manipüle etmek. Burada **for i in b: print(i)** komutunu verirseniz **group()** metodu yardımıyla ne bulduğumuzu görebilirsiniz:

```
çileğ
çilek
çileğ
çilek
Çileğ
çilek
çilek
çilek
Çileğ
çilek
çilek
çilek
```

Bu çıktıyı gördükten sonra, kodlarda yapmaya çalıştığımız şey daha anlamlı görünmeye başlamış olmalı. Şimdi, **sonraki satıra geçiyoruz**:

```
for i in b:
    return a[i]
```

Burada, **group()** metodu yardımıyla bulduğumuz eşleşmeler üzerinde bir **for** döngüsü oluşturduk. Ardından da **return a[i] komutunu vererek a** adlı **sözlük içinde yer alan ögeleri yazdırıyoruz**. Bu arada, buradaki "**i**"nin yukarıda verdiğimiz **group()** çıktılarını temsil ettiğine dikkat edin. **a[i]** gibi bir komut verdiğimizde aslında sırasıyla şu komutları vermiş oluyoruz:

```
a["çilek"]
a["çileğ"]
a["çilek"]
a["Çileğ"]
a["çilek"]
a["çilek"]
a["çilek"]
a["Çileğ"]
a["çilek"]
a["çilek"]
```

Bu komutların çıktıları sırasıyla "**erik**", "**erığ**", "**erik**", "**Erığ**", "**erik**", "**erik**", "**erik**", "**Erığ**", "**erik**", "**erik**" olacaktır. İşte bu **return** satırı, bir sonraki kod olan **print(derle.sub(değiştir, metin))** ifadesinde etkinlik kazanacak. **Bu son satırımız sözlük ögelerini tek tek metne uygulayacak** ve mesela **a["çilek"]** komutu sayesinde metin içinde "**çilek**" gördüğü yerde "**erik**" kelimesini yapıştıracak ve böylece bize **istediğimiz şekilde değiştirilmiş bir metin verecektir**.

Bu kodların biraz karışık gibi göründüğünü biliyorum, **ama aslında çok basit bir mantığı var**: **group()** metodu ile metin içinde aradığımız kelimeleri ayıklıyor. Ardından da **a** sözlüğü içinde bunları anahtar olarak kullanarak "**çilek**" ve çeşitleri yerine "**erik**" ve çeşitlerini koyuyor...

Yukarıda verdiğimiz düzenli ifadeyi böyle ufak bir metinde kullanmak çok anlamlı olmayabilir. Ama çok büyük metinler üzerinde çok çeşitli ve karmaşık değişiklikler yapmak istediğinizde bu kodların işinize yarayabileceğini göreceksiniz.

SUBN() METODU

Bu metodu, çok kısa bir şekilde anlatıp geçeceğiz. Çünkü bu metot **`sub()`** metoduyla *neredeyse tamamen aynıdır*. Tek farkı, **`subn()`** metodunun **bir metin içinde yapılan değişiklik sayısını da göstermesidir**. Yani bu metodu kullanarak kullanıcılarınıza "**toplam şu kadar sayıda değişiklik yapılmıştır**" şeklinde bir bilgi verebilirsiniz. Bu metot, **çıktı** olarak **iki ögeli bir demet verir**. Birinci öge **değiştirilen metin**, ikinci öge ise **yapılan değişiklik sayısıdır**. Yani kullanıcıya değişiklik sayısını göstermek için yapmanız gereken şey, **bu demetin ikinci ögesini almaktır**. Mesela **`sub()`** metodunu anlatırken verdiğimiz kodların **son satırını** şöyle değiştirebilirsiniz:

```
ab = derle.subn(degistir, metin)
print("Toplam {ab[1]} değişiklik yapılmıştır.")
```

SONUÇ

Böylelikle düzenli ifadeler konusunu bitirmiş olduk. Buradaki amacımız, size düzenli ifadeler konusunda **genel bir bakış sunabilmekti**. Bu yazıları okuduktan sonra kafanızda düzenli ifadelerle ilgili kabataslak da olsa bir resim oluştuysa **bu yazılar amacına ulaşmış demektir**. Elbette düzenli ifadeler, *burada anlattıklarımızdan ibaret değildir*. Bu konunun üzerine eğildiğinizde aslında düzenli ifadelerin dipsiz bir kuyu gibi olduğunu göreceksiniz. Esasında en başta da dediğimiz gibi düzenli ifadeler **apayrı bir dil gibidir**. Doğrusu şu ki düzenli ifadeler başlı başına **bağımsız bir sistemdir**. Hemen hemen bütün programlama dilleri öyle ya da böyle düzenli ifadeleri destekler. **Python da düzenli ifadeleri bünyesine adapte etmiş dillerden biridir**. Bizim düzenli ifadeler konusundaki yaklaşımımız, her zaman bunları gerektiğinde kullanmak olmalıdır. Dediğimiz gibi eğer yapmak istediğiniz bir işlemi karakter dizilerinin metotları yardımıyla yapabiliyorsanız, düzenli ifadelere girişmemek en iyisidir. Çünkü karakter dizisi metotları hem *daha hızlıdır* hem de *anlaması daha kolaydır*.

SQLITE İLE VERİ TABANI PROGRAMLAMA

26

Giriş

Bu bölümde son derece önemli bir konu olan **veri tabanı programlama**yı inceleyeceğiz.

Python'da veri tabanı programlama yapabilmek için pek çok alternatifimiz var. Biz bu alternatifler içinde, sadeliği, basitliği ve kullanım kolaylığı nedeniyle **Sqlite adlı veri tabanı yönetim sistemini ele alacağız**.

Neden Sqlite?

Dediğimiz gibi Python'da veri tabanı işlemleri için kullanabileceğiniz pek çok alternatif bulunur. Ama biz bütün bu alternatifler içinde Sqlite'ı tercih edeceğiz. *Peki neden Sqlite?*

Sqlite'ın öteki sistemlere göre pek çok **avantajı** bulunur. Gelin, isterseniz Sqlite'ın bazı avantajlarına şöyle bir göz gezdirelim:

- Her şeyden önce Python'ın standart kütüphanesi içinde Sqlite veri tabanlarını yönetebilmemizi sağlayan **`sqlite3`** adlı bir modül bulunur. Bu modülü Python'daki herhangi bir modül gibi **içe aktarabilir** ve böylece Sqlite veri tabanı sistemini kullanmaya başlayabilirsiniz.
- Sqlite herhangi bir **yazılım** veya **sunucu kurulumu gerektirmez**. Bu sayede **`sqlite3`** modülünü kullanabilmek için öncelikle bir sunucu yapılandırmanıza gerek kalmaz.

- Sqlite, öteki pek çok veri tabanı alternatifine göre **basittir**. Bu yüzden Sqlite'ı çok kısa bir sürede kavrayıp kullanmaya başlayabilirsiniz.
- Sqlite **özgür** bir yazılımdır. Bu yazılımın baştan aşağı bütün kodları kamuya açıktır. Dolayısıyla Sqlite kodlarının her zerresini istediğiniz gibi kullanabilir, değişikliğe uğratabilir, satabilir ve ticari olan/olmayan bütün uygulamalarınızda gönül rahatlığıyla kullanabilirsiniz.
- Sqlite'ın sade ve basit olması sizi yanıltmasın. Bu özelliklerine bakarak Sqlite'ın **yeteneksiz bir veri tabanı sistemi olduğunu düşünmeyin**. Bugün Sqlite'ı aktif olarak kullanan pek çok büyük ve tanınmış şirket bulunur. Mesela, Adobe, Apple, Mozilla/Firefox, Google, Symbian ve Sun bu şirketlerden bazılarıdır.

  Yukarıdaki sebeplerden ötürü, veri tabanı konusunu Sqlite üzerinden anlatacağız. O hâlde hemen yola koyulalım.

Sqlite'ın Yapısı

Veri tabanları, verileri sonradan kullanılmak üzere içinde tutan bir sistemdir. Bütün ilişkisel veri tabanlarında olduğu gibi Sqlite da bu verileri **tablo benzeri bir yapı içinde tutar**. Yani aslında bir Sqlite veri tabanı içindeki veriler şöyle bir yapıya sahiptir:

Sütun 1	Sütun 2	Sütun 3	Sütun 4	Sütun 5
Değer 1/1	Değer 2/1	Değer 3/1	Değer 4/1	Değer 5/1
Değer 1/2	Değer 2/2	Değer 3/2	Değer 4/2	Değer 5/2
Değer 1/3	Değer 2/3	Değer 3/3	Değer 4/3	Değer 5/3
Değer 1/4	Değer 2/4	Değer 3/4	Değer 4/4	Değer 5/4

Sqlite içinde oluşturulan yukarıdakine benzer her tablonun bir de **ismi** vardır. Daha doğrusu, Sqlite ile bir tablo oluştururken bu tabloya bir de **ad vermemiz gerekir**. Mesela yukarıdaki tabloya '**değerler**' adını verdiğimizi varsayabilirsiniz.

Sqlite ile çalışırken veriler üzerinde yapacağımız işlemleri, yukarıdaki tablonun adını ve bu tablodaki sütunları kullanarak gerçekleştireceğiz. Bu yüzden Sqlite'ın **yapısını anlamak büyük önem taşır**. Gördüğünüz gibi bu veri tabanı sisteminin yapısını anlamak da öyle zor bir iş değildir.

Örnek Veri Tabanı

Sqlite'ı öğrenirken içinde örnek veriler barındıran bir veri tabanının elimizin altında bulunması **alıştırma yapabilmek açısından** faydalı olacaktır. Bunun için *https://www.dropbox.com/s/d9a0ek2yw9m0i1e/kitaplar.sqlite?dl=1* adresindeki **örnek veri tabanını bilgisayarınıza indirin**. Veri tabanı sorgu çalışmalarımızı, bu örnek veri tabanı üzerinde gerçekleştireceğiz.

Yeni Bir Veri Tabanı Oluşturmak

Bu bölümde **`sqlite3`** adlı **standart kütüphane modülü** aracılığıyla yeni bir veri tabanını nasıl oluşturacağımızı öğreneceğiz.

Bu modülü kullanabilmek için öncelikle modülü **içe aktarmamız** gerektiğini biliyorsunuz:

```
>>> import sqlite3
```

Eğer bu isim size çok uzun geliyorsa veya modül adında sayıların ve harflerin birlikte bulunması nedeniyle hem sayı hem de harf girmeyi bir angarya olarak görüyorsanız elbette **`sqlite3`** modülünü **farklı bir adla da içe aktarabilirsiniz**. Mesela:

```
>>> import sqlite3 as sql
```

Veya:

```
>>> import sqlite3 as lite
```

Böylece **`sqlite3`** modülünü '**`sql`**' veya '**`lite`**' adıyla içe aktarmış olduk. Ancak ben konuyu anlatırken okur açısından *kafa karışıklığına sebep olmamak için* modülü **`import sqlite3`** şeklinde içe aktarmışız gibi davranacağım.

Gelelim bu modül yardımıyla nasıl veri tabanı oluşturulacağına... Bunun için **sqlite3** modülünün **connect()** adlı metodundan yararlanacağız. Bu metodu şu şekilde kullanıyoruz:

```
>>> vt = sqlite3.connect('veri tabanı_adı')
```

connect() metoduna verdiğimiz **varitabanı_adı** adlı **argüman**, **kullanacağımız veri tabanının adıdır**. Eğer belirtilen isimde, bir veri tabanı sistemde **bulunmuyorsa** o adla **yeni bir veri tabanı oluşturulacaktır**. Mesela:

```
>>> vt = sqlite3.connect('deneme.sqlite')
```

Eğer bu komutu verdiğiniz dizin içinde '**deneme.sqlite**' adlı bir veri tabanı yoksa **bu ada sahip bir veri tabanı oluşturulacaktır**.

Bu arada, biz veri tabanı dosyasının uzantısı olarak **.sqlite**'ı seçtik. Ama eğer siz isterseniz kendinize uygun başka bir uzantı da belirleyebilirsiniz. Veri tabanı dosyasının **uzantısının ne olması gerektiği konusunda kesin kurallar bulunmaz**. **.sqlite** uzantısının yerine, **.sqlite3**, **.db** veya **.db3** gibi uzantıları tercih edenler de vardır. Hatta eğer siz isterseniz veri tabanınızın uzantısını **.osman** olarak dahi belirleyebilirsiniz. Bu konuda herhangi bir **kısıtlama bulunmaz**.

Yukarıdaki örnekte '**deneme.sqlite**' adını verdiğimiz bir veri tabanı dosyasına, **connect()** metodu yardımıyla bağlandık. Elbette isteseydik **connect()** metoduna **argüman** olarak tam dosya yolu da verebilirdik:

```
>>> import sqlite3
>>> vt = sqlite3.connect('/home/istihza/test.sqlite') #GNU/Linux
>>> vt = sqlite3.connect('c:/users/fozgul/desktop/test.sqlite')
#Windows
```

Bu komut yardımıyla sabit disk üzerinde bir **Sqlite veri tabanı** dosyası oluşturmuş oluyoruz. Ancak isterseniz **sqlite3** ile geçici bir veri tabanı da oluşturabilirsiniz:

```
>>> vt = sqlite3.connect(':memory:')
```

Oluşturduğunuz bu geçici veri tabanı, sabit disk üzerinde değil **RAM** (**bellek**) üzerinde çalışır. Veri tabanını kapattığınız anda da bu geçici veri tabanı **silinir**. Eğer arzu ederseniz RAM üzerinde değil, disk üzerinde de **geçici veritabanları** oluşturabilirsiniz. Bunun için de şöyle bir komut kullanıyoruz:

```
>>> vt = sqlite3.connect('')
```

Gördüğünüz gibi disk üzerinde, geçici bir veri tabanı oluşturmak için boş bir karakter dizisi kullandık. Tıpkı `:memory:` kullanımında olduğu gibi boş karakter dizisiyle oluşturulan **geçici veritabanları da veri tabanı bağlantısının kesilmesiyle birlikte ortadan kalkacaktır**.

Geçici veri tabanı oluşturmak, özellikle çeşitli testler veya denemeler yaptığınız durumlarda işinize yarar. Sonradan nasıl olsa sileceğiniz, sırf test amaçlı tuttuğunuz bir veri tabanını disk üzerinde oluşturmak yerine RAM üzerinde oluşturmayı tercih edebilirsiniz. Ayrıca, geçici veri tabanları sayesinde, yazdığınız bir kodu test ederken bir hatayla karşılaşırsanız sorunun veri tabanı içinde **varolan verilerden değil**, **yazdığınız koddan kaynaklandığından** da emin olabilirsiniz. Çünkü, dediğimiz gibi programın her yeniden çalışışında veri tabanı **baştan** oluşturulacaktır.

Dikkatinizi çekmek istediğim bir nokta da şudur: Gördüğünüz gibi Sqlite, veri tabanını o anda içinde bulunduğunuz dizin içinde oluşturuyor. Mesela MySQL kullanıyor olsaydınız oluşturulan veri tabanlarının önceden tanımlanmış bir dizin içine atıldığını görecektiniz. Örneğin GNU/Linux sistemlerinde, **MySQL veri tabanları /var/lib/mysql** gibi bir dizinin içinde tutulur.

Varolan Bir Veri tabanıyla Bağlantı Kurmak

Biraz önce, '**deneme.sqlite**' adlı yeni bir Sqlite veri tabanı oluşturmak için **şöyle bir komut kullanmıştık**:

```
>>> vt = sqlite3.connect('deneme.sqlite')
```

Eğer bu komutu verdiğiniz dizin içinde '**deneme.sqlite**' adlı bir veri tabanı yoksa **bu ada sahip bir veri tabanı oluşturulur**. Eğer zaten bu adla bir veri tabanı dosyanız varsa **sqlite3** bu veri tabanına bağlanacaktır. Dolayısıyla Sqlite'ta hem yeni bir veri tabanı oluşturmak hem de mevcut bir veri tabanına bağlanmak için birbiriyle tamamen aynı kodları kullanıyoruz.

Mesela biraz önce *https://www.dropbox.com/s/d9a0ek2yw9m0i1e/kitaplar.sqlite?dl=1* adresinden indirdiğimiz '**kitaplar.sqlite**' adlı veri tabanına bağlanalım.

Bu dosyanın bulunduğu konumda bir Python **etkileşimli kabuk oturumu** açtığımızı varsayarsak:

```
>>> vt = sqlite3.connect('kitaplar.sqlite')
```

komutunu kullanarak '**kitaplar.sqlite**' adlı veri tabanıyla bağlantı kurabiliriz.

İmleç Oluşturma

Yukarıda **connect()** metodunu kullanarak hem Sqlite ile nasıl veri tabanı bağlantısı kuracağımızı hem de nasıl **yeni** bir veri tabanı oluşturacağımızı öğrendik.

connect() metodu, bir veri tabanı üzerinde işlem yapabilmemizin ilk adımıdır. Veri tabanını oluşturduktan veya varolan bir veri tabanı ile bağlantı kurduktan sonra, **veri tabanı üzerinde işlem yapabilmek için sonraki adımda bir imleç oluşturmamız gerekir**.

İmleç oluşturmak için `cursor()` adlı bir metottan yararlanacağız:

```
>>> im = vt.cursor()
```

İmleci oluşturduktan sonra artık **önümüz iyice** açılıyor. Böylece, yukarıda oluşturduğumuz `im` nesnesinin `execute()` metodunu kullanarak SQL komutlarını çalıştırabileceğiz. *Nasıl mı?* **Hemen bakalım.**

Tablo Oluşturma

Önceki bölümün sonunda söylediğimiz gibi bir imleç nesnesi oluşturduktan sonra bunun `execute()` metodunu kullanarak **SQL komutlarını işletebiliyoruz**.

Dilerseniz şimdi, basit bir **örnek** yaparak neyin ne olduğunu anlamaya çalışalım.

Öncelikle gerekli modülü **içe aktaralım**:

```
>>> import sqlite3
```

Şimdi de **yeni bir veri tabanı dosyası oluşturalım** (*veya varolan bir veri tabanı dosyasına bağlanalım*):

```
>>> vt = sqlite3.connect('veritabani.sqlite')
```

Bu veri tabanı üzerinde işlem yapabilmek için **öncelikle imlecimizi oluşturalım**:

```
>>> im = vt.cursor()
```

Şimdi de yukarıda oluşturduğumuz imlecin `execute()` adlı metodunu kullanarak veri tabanı içinde bir **tablo oluşturalım**:

```
>>> im.execute("CREATE TABLE adres_defteri (isim, soyisim)")
```

Hatırlarsanız, Sqlite veri tabanı sisteminin **tablo benzeri bir yapıya sahip olduğunu** ve **bu sistemdeki her tablonun da bir isminin bulunduğunu** söylemiştik. İşte burada yaptığımız şey, '`adres_defteri`' adlı bir tablo oluşturup bu tabloya '`isim`' ve '`soyisim`' adlı iki sütun eklemekten ibarettir. Yani aslında şöyle bir şey oluşturmuş oluyoruz:

isim	soyisim

Ayrıca oluşturduğumuz bu tablonun adının da '`adres_defteri`' olduğunu unutmuyoruz...

Bu işlemleri nasıl yaptığımıza dikkat edin. Burada "`CREATE TABLE adres_defteri (isim, soyisim)`" **tek bir karakter dizisidir**. Bu karakter dizisindeki `CREATE TABLE` kısmı bir SQL komutu olup; **bu komut bir tablo oluşturulmasını sağlar**.

Burada `CREATE TABLE` ifadesini **büyük harflerle** yazdık. Ancak bu ifadeyi siz isterseniz küçük harflerle de yazabilirsiniz. Benim burada büyük harf kullanmaktaki amacım **SQL komutlarının**, '`adres_defteri`', '`isim`' ve '`soyisim`' gibi ögelerden görsel olarak ayırt edilebilmesini sağlamak. Yani `CREATE TABLE` ifadesinin mesela '`adres_defteri`' ögesinden **kolayca ayırt edilebilmesini** istediğim için burada `CREATE TABLE` ifadesini büyük harflerle yazdım.

Karakter dizisinin devamında (`isim, soyisim`) ifadesini görüyoruz. Tahmin edebileceğiniz gibi bunlar tablodaki **sütun başlıklarının adını** gösteriyor. Buna göre oluşturduğumuz tabloda '`isim`' ve '`soyisim`' adlı iki farklı sütun başlığı olacak.

Bu arada, Sqlite tabloları oluştururken tablo adı ve sütun başlıklarında **Türkçe karakter kullanmaktan kaçınmak iyi bir fikirdir**. Ayrıca eğer tablo adı ve sütun başlıklarında **birden fazla** kelimeden oluşan etiketler kullanacaksanız bunları ya birbirine **bitiştirin** ya da **tırnak içine alın**.

Örneğin:

```
import sqlite3

vt = sqlite3.connect('perso.sqlite')
im = vt.cursor()

im.execute("""CREATE TABLE 'personel dosyasi'
('personel ismi', 'personel soyismi', memleket)""")
```

Ayrıca, **execute()** metoduna parametre olarak verilen SQL komutlarının alelade birer karakter dizisi olduğuna da dikkatinizi çekmek isterim. Bunlar, **Python'daki karakter dizilerinin bütün özelliklerini taşır**. Mesela bu karakter dizisini **execute()** metoduna göndermeden önce bir değişkene atayabilirsiniz:

```
import sqlite3

vt = sqlite3.connect('perso.sqlite')
im = vt.cursor()

sql = """CREATE TABLE 'personel dosyasi'
('personel ismi', 'personel soyismi', memleket)"""

im.execute(sql)
```

Bu arada, bu kodları ikinci kez çalıştırdığınızda şöyle bir **hata mesajı** alacaksınız:

```
sqlite3.OperationalError: table 'personel dosyasi' already exists
```

Bu hata mesajını almanız gayet normal. Bunun üstesinden nasıl geleceğinizi öğrenmek için okumaya devam edin...

Şartlı Tablo Oluşturma

CREATE TABLE komutunu kullanarak tablo oluştururken şöyle bir **problemle** karşılaşmış olabilirsiniz. Diyelim ki şu kodları yazdınız:

```
import sqlite3

vt = sqlite3.connect('vt.sqlite')

im = vt.cursor()
im.execute("CREATE TABLE personel (isim, soyisim, memleket)")
```

Bu kodları ilk kez çalıştırdığınızda, mevcut dizin altında **vt.sqlite** adlı bir veri tabanı dosyası oluşturulacak ve bu veri tabanı içinde '**isim**', '**soyisim**' ve '**memleket**' başlıklı sütunlara sahip, '**personel**' adlı bir tablo meydana getirilecektir.

Ancak aynı kodları **ikinci kez** çalıştırdığınızda şöyle bir **hata mesajı** ile karşılaşacaksınız:

```
sqlite3.OperationalError: table personel already exists
```

Buradaki sorun, '**vt.sqlite**' dosyası içinde '**personel**' adlı bir tablonun *zaten bulunuyor olmasıdır*. Bir veri tabanı üzerinde işlem yaparken **aynı ada sahip iki tablo oluşturamayız**. Bu hatayı önlemek için **şartlı tablo oluşturma** yönteminden yararlanacağız. Bunun için kullanacağımız **SQL komutu** şudur: **CREATE TABLE IF NOT EXISTS**.

Örneğimizi bu yeni bilgiye göre **tekrar** yazalım:

```
import sqlite3

vt = sqlite3.connect('vt.sqlite')

im = vt.cursor()

sorgu = """CREATE TABLE IF NOT EXISTS personel
(isim, soyisim, memleket)"""

im.execute(sorgu)
```

Bu kodları kaç kez çalıştırırsanız çalıştırın, programınız **hata vermeden işleyecek**; eğer veri tabanında '**personel**' adlı bir tablo yoksa oluşturacak, bu adla zaten bir tablo varsa da sessizce yoluna devam edecektir.

Tabloya Veri Girme

Buraya kadar, **sqlite3** modülünü kullanarak nasıl bir veri tabanı oluşturacağımızı ve çeşitli sütunlardan oluşan bir tabloyu bu veri tabanına nasıl yerleştireceğimizi öğrendik. Şimdi de oluşturduğumuz bu *sütun başlıklarının altını dolduralım*.

Dikkatlice bakın:

```
import sqlite3

vt = sqlite3.connect('vt.sqlite')
im = vt.cursor()

tablo_yap = """CREATE TABLE IF NOT EXISTS personel
(isim, soyisim, memleket)"""

değer_gir = """INSERT INTO personel VALUES ('Fırat', 'Özgül',
'Adana')"""

im.execute(tablo_yap)
im.execute(değer_gir)
```

Burada **INSERT INTO tablo_adı VALUES** adlı yeni bir **SQL komutu** daha öğreniyoruz. **CREATE TABLE** ifadesi Türkçe'de "**TABLO OLUŞTUR**" anlamına geliyor. **INSERT INTO** ise "**... İÇİNE YERLEŞTİR**" anlamına gelir. Yukarıdaki karakter dizisi içinde görünen **VALUES** ise "**DEĞERLER**" demektir. Yani aslında yukarıdaki karakter dizisi şu anlama gelir: "personel İÇİNE '**Fırat**', '**Özgül**' ve '**Adana**' değerlerini yerleştir. Yani şöyle bir tablo oluştur":

isim	soyisim	memleket
Fırat	Özgül	Adana

Buraya kadar gayet güzel gidiyoruz. İsterseniz, şimdi derin bir nefes alıp şu ana kadar yaptığımız şeyleri bir gözden geçirelim:

- Öncelikle **`sqlite3`** modülünü içe aktardık. Bu modülün nimetlerinden yararlanabilmek için bunu yapmamız gerekiyordu. "**`sqlite3`**" kelimesini her defasında yazmak bize angarya gibi gelebileceği için bu modülü farklı bir adla içe aktarmayı tercih edebiliriz. Mesela **`import sqlite3 as sql`** veya **`import sqlite3 as lite`** gibi...
- **`sqlite3`** modülünü içe aktardıktan sonra bir veri tabanına bağlanmamız veya elimizde bir veri tabanı yoksa **yeni bir veri tabanı** oluşturmamız gerekiyor. Bunun için **`connect()`** adlı bir fonksiyondan yararlanıyoruz. Bu fonksiyonu, **`sqlite3.connect('veri tabanı_adı')`** şeklinde kullanıyoruz. Eğer içinde bulunduğumuz dizinde, "**`veri tabanı_adı`**" adlı bir veri tabanı varsa **`Sqlite`** bu veri tabanına **bağlanır**. Eğer bu adda bir veri tabanı yoksa çalışma dizini altında bu ada sahip yeni bir veri tabanı oluşturulur. Özellikle deneme amaçlı işlemler yapmamız gerektiğinde, sabit disk üzerinde bir veri tabanı oluşturmak yerine **RAM üstünde geçici bir veri tabanı ile çalışmayı da tercih edebiliriz**. Bunun için yukarıdaki komutu şöyle yazıyoruz: **`sqlite3.connect(':memory:')`**. Bu komutla RAM üzerinde oluşturduğumuz veri tabanı, bağlantı kesildiği anda ortadan kalkacaktır.
- Veri tabanımızı oluşturduktan veya varolan bir veri tabanına bağlandıktan sonra yapmamız gereken **şey bir imleç oluşturmak** olacaktır. Daha sonra bu imlece ait **metotlardan yararlanarak önemli işler yapabileceğiz**. Sqlite'ta bir imleç oluşturabilmek için **`db.cursor()`** gibi bir komut kullanıyoruz. Tabii ben burada oluşturduğunuz veri tabanına "**`db`**" adını verdiğinizi varsayıyorum.
- İmlecimizi de oluşturduktan sonra **önümüz iyice açılmış oldu**. Şimdi, **`dir(im)`** gibi bir komut kullanarak imlecin metotlarının ne olduğunu inceleyebilirsiniz. Tabii ben burada imlece "**`im`**" adını verdiğinizi varsaydım. Gördüğünüz gibi listede **`execute()`** adlı bir metot da var. Artık imlecin bu **`execute()`** metodunu kullanarak **SQL komutlarını** işletebiliriz.
- Yukarıda üç adet SQL komutu öğrendik. Bunlardan ilki **`CREATE TABLE`**. Bu komut veri tabanı içinde bir **tablo oluşturmamızı** sağlıyor. İkinci komutumuz **`CREATE TABLE IF NOT EXISTS`**. Bu komut da bir **tabloyu eğer yoksa oluşturmamızı** sağlıyor. Üçüncü komutumuz ise **`INSERT INTO ... VALUES ....`** Bu komut, oluşturduğumuz tabloya **içerik eklememizi** sağlıyor. Bunları şuna benzer bir şekilde kullandığımızı hatırlıyorsunuz:

```
im.execute("CREATE TABLE personel (isim, soyisim, memleket)")
im.execute("INSERT INTO personel VALUES ('Fırat', 'Özgül',
'Adana')")
```

Burada bir şey dikkatinizi çekmiş olmalı. SQL komutlarını yazmaya başlarken **çift tırnakla** başladık. Dolayısıyla karakter dizisini yazarken iç taraftaki **`Fırat`**, **`Özgül`** ve **`Adana`** değerlerini yazmak için tek **tırnak kullanmamız gerekti**. Karakter dizileri içindeki manevra alanınızı genişletmek için SQL komutlarını **üç tırnak içinde yazmayı da** tercih edebilirsiniz. Böylece karakter dizisi içindeki tek ve çift tırnakları daha rahat bir şekilde kullanabilirsiniz. Yani:

```
im.execute("""CREATE TABLE personel (isim, soyisim, memleket)""")
im.execute("""INSERT INTO personel VALUES ("Fırat", "Özgül",
"Adana")""")
```

Ayrıca üç tırnak kullanmanız sayesinde, uzun satırları gerektiğinde *bölerek* çok daha okunaklı kodlar da yazabileceğinizi biliyorsunuz.

Verilerin Veri Tabanına İşlenmesi

Bir önceki bölümde, bir **Sqlite veri tabanına nasıl veri gireceğimizi öğrendik**. Ama aslında iş sadece veri girmeyle bitmiyor. Verileri veri tabanına "**işleyebilmek**" için bir adım daha atmamız gerekiyor. Mesela şu örneğe bir bakalım:

```
import sqlite3

vt = sqlite3.connect("vt.sqlite")

im = vt.cursor()
im.execute("""CREATE TABLE IF NOT EXISTS
    personel (isim, soyisim, sehir, eposta)""")

im.execute("""INSERT INTO personel VALUES
    ("Orçun", "Kunek", "Adana", "okunek@gmail.com")""")
```

Burada öncelikle '**vt.sqlite**' adlı bir **veri tabanı** oluşturduk ve bu veri tabanına bağlandık. Ardından, **vt.cursor()** komutuyla imlecimizi de oluşturduktan sonra, SQL komutlarımızı çalıştırıyoruz. Önce **isim**, **soyisim**, **şehir** ve **eposta** adlı sütunlardan oluşan, "**personel**" adlı bir tablo oluşturduk. Daha sonra "personel" tablosunun içine "**Orçun**", "**Kunek**", "**Adana**" ve "**okunek@gmail.com**" değerlerini yerleştirdik.

Ancak her ne kadar veri tabanına veri işlemiş gibi görünsek de **aslında henüz işlenmiş bir şey yoktur**. Biz henüz sadece verileri girdik. Ama verileri veri tabanına **işlemedik**. Bu girdiğimiz verileri veri tabanına işleyebilmek için **commit()** adlı bir metottan yararlanacağız:

```
>>> vt.commit()
```

Gördüğünüz gibi **commit()** imlecin değil, bağlantı nesnesinin (*yani burada vt değişkeninin*) bir metodudur. Şimdi, bu satırı da betiğimize ekleyelim:

```
import sqlite3

vt = sqlite3.connect("vt.sqlite")

im = vt.cursor()
im.execute("""CREATE TABLE IF NOT EXISTS
    personel (isim, soyisim, sehir, eposta)""")

im.execute("""INSERT INTO personel VALUES
    ("Orçun", "Kunek", "Adana", "okunek@gmail.com")""")

vt.commit()
```

Bu son satırı da ekledikten sonra Sqlite veri tabanı içinde **şöyle bir tablo oluşturmuş olduk**:

isim	soyisim	şehir	eposta
Orçun	Kunek	Adana	okunek@gmail.com

Eğer **vt.commit()** satırını yazmazsak veri tabanı, **tablo** ve **sütun başlıkları** oluşturulur, ama sütunların içeriği veri tabanına işlenmez.

Veri tabanının Kapatılması

Bir veri tabanı üzerinde yapacağımız bütün işlemleri tamamladıktan sonra, **prensip olarak o veri tabanını kapatmamız gerekir**. Mesela şu kodları ele alalım:

```
import sqlite3

vt = sqlite3.connect("vt.sqlite")

im = vt.cursor()
im.execute("""CREATE TABLE IF NOT EXISTS
    personel (isim, soyisim, sehir, eposta)""")

im.execute("""INSERT INTO personel VALUES
    ("Orçun", "Kunek", "Adana", "okunek@gmail.com")""")

vt.commit()
vt.close()
```

Burada bütün işlemleri bitirdikten sonra veri tabanını **kapatmak için `close()`** adlı bir metottan yararlandık:

```
vt.close()
```

Bu şekilde, veri tabanının ilk açıldığı andan itibaren, işletim sisteminin devreye soktuğu kaynakları *serbest bırakmış oluyoruz*. Esasında programımız kapandığında, **açık olan bütün Sqlite veritabanları da otomatik olarak kapanır.** Ama yine de bu işlemi **elle yapmak her zaman iyi bir fikirdir**.

Eğer üzerinde işlem yaptığınız veri tabanının her şey bittikten sonra otomatik olarak kapanmasını garantilemek isterseniz **`with`** deyimini kullanabilirsiniz:

```
import sqlite3

with sqlite3.connect('vt.sqlite') as vt:
    im = vt.cursor()
```

```
    im.execute("""CREATE TABLE IF NOT EXISTS personel
        (isim, soyisim, memleket)""")
    im.execute("""INSERT INTO personel VALUES
        ('Fırat', 'Özgül', 'Adana')""")

    vt.commit()
```

Bu şekilde **with** deyimini kullanarak bir veri tabanı bağlantısı açtığımızda, bütün işler bittikten sonra Python bizim için bağlantıyı **otomatik olarak sonlandıracaktır**.

Parametreli Sorgular

Şu ana kadar verdiğimiz örneklerde, veri tabanına girilecek verileri **tek tek elle yerine koyduk**. Örneğin:

```
im.execute("""INSERT INTO personel VALUES
    ('Fırat', 'Özgül', 'Adana')""")
```

Ancak çoğu durumda veri tabanına girilecek veriler **haricî kaynaklardan** gelecektir. Basit bir **örnek** verelim:

```
import sqlite3

with sqlite3.connect('vt.sqlite') as vt:
    im = vt.cursor()

    veriler = [('Fırat', 'Özgül', 'Adana'),
               ('Ahmet', 'Söz', 'Bolvadin'),
               ('Veli', 'Göz', 'İskenderun'),
               ('Mehmet', 'Öz', 'Kilis')]

    im.execute("""CREATE TABLE IF NOT EXISTS personel
        (isim, soyisim, memleket)""")

    for veri in veriler:
        im.execute("""INSERT INTO personel VALUES
            (?, ?, ?)""", veri)

    vt.commit()
```

Burada veri tabanına işlenecek veriler, **veriler** adlı bir değişkenden geliyor. Bu değişken içindeki verileri, **veri tabanına nasıl yerleştirdiğimize dikkat edin**:

```
for veri in veriler:
    im.execute("""INSERT INTO personel VALUES
        (?, ?, ?)""", veri)
```

Ayrıca her bir sütunun (**'isim', 'soyisim', 'memleket'**) altına gelecek her bir değer için (*mesela sırasıyla 'Fırat', 'Özgül', 'Adana'*) **bir adet** '**?**' **işareti yerleştirdiğimizi de gözden kaçırmayın**.

Tablodaki Verileri Seçmek

Yukarıda, bir veri tabanına nasıl veri gireceğimizi ve **bu verileri veri tabanına nasıl işleyeceğimizi gördük**. İşin asıl önemli kısmı, bu verileri daha sonra veri tabanından (*yani tablodan*) geri alabilmektir. Şimdi, bu işlemi nasıl yapacağımıza bakalım.

Veri tabanından herhangi bir veri alabilmek için ilk olarak **SELECT veri FROM tablo_adı** adlı bir **SQL komutundan yararlanarak** ilgili verileri seçmemiz gerekiyor.

Dilerseniz önce bir **tablo** oluşturalım:

```
import sqlite3

vt = sqlite3.connect('vt.sqlite')

im = vt.cursor()

im.execute("""CREATE TABLE IF NOT EXISTS faturalar
(fatura, miktar, ilk_odeme_tarihi, son_odeme_tarihi)""")
```

Şimdi bu tabloya bazı **veriler ekleyelim**:

```
im.execute("""INSERT INTO faturalar VALUES
("Elektrik", 45, "23 Ocak 2020", "30 Ocak 2020")""")
```

Verileri, **veri tabanına işleyelim**:

```
vt.commit()
```

Yukarıdaki kodlar bize şöyle bir tablo verdi:

fatura	miktar	ilk_odeme_tarihi	son_odeme_tarihi
Elektrik	45	23 Ocak 2020	30 Ocak 2020

Buraya kadar olan kısmı zaten biliyoruz. *Bilmediğimiz ise bu veri tabanından nasıl veri alacağımız*. **Onu da şöyle yapıyoruz**:

```
im.execute("""SELECT * FROM faturalar""")
```

Burada özel bir SQL komutu olan **`SELECT veri FROM tablo_adı`** komutundan faydalandık. Burada joker karakterlerden biri olan "**`*`**" işaretini kullandığımıza dikkat edin. **`SELECT * FROM`** faturalar ifadesi şu anlama gelir: "**faturalar adlı tablodaki bütün ögeleri seç!**"

Burada "**`SELECT`**" kelimesi "**SEÇMEK**" demektir. "**`FROM`**" ise "**...DEN/...DAN**" anlamı verir. Yani "**`*SELECT FROM faturalar*`**" dediğimizde "**faturalardan seç**" demiş oluyoruz. Burada kullandığımız "**`*`**" işareti de "**her şey**" anlamına geldiği için "**`*SELECT * FROM faturalar*`**" ifadesi "**faturalardan her şeyi seç**" gibi bir anlama gelmiş oluyor.

Verileri seçtiğimize göre artık seçtiğimiz bu verileri nasıl alacağımıza bakabiliriz. Bunun için de **`fetchone()`**, **`fetchall()`** veya **`fetchmany()`** adlı metotlardan ya da **`for`** döngüsünden yararlanacağız.

Seçilen Verileri Almak

Bu bölümde, **SELECT** sorgusu ile veri tabanından seçtiğimiz verileri farklı yollarla nasıl **çekebileceğimizi**/**alabileceğimizi** inceleyeceğiz.

fetchall() Metodu

Biraz önce şöyle bir program yazmıştık:

```
import sqlite3

vt = sqlite3.connect('vt.sqlite')

im = vt.cursor()

im.execute("""CREATE TABLE IF NOT EXISTS faturalar
(fatura, miktar, ilk_odeme_tarihi, son_odeme_tarihi)""")

im.execute("""INSERT INTO faturalar VALUES
("Elektrik", 45, "23 Ocak 2020", "30 Ocak 2020")""")

vt.commit()

im.execute("""SELECT * FROM faturalar""")
```

Burada '**vt.sqlite**' adlı bir veri tabanında '**faturalar**' adlı bir tablo oluşturduk ve bu tabloya **bazı veriler girdik**. Daha sonra da **SELECT** adlı **SQL komutu yardımıyla** bu verileri seçtik. Şimdi de seçtiğimiz bu verileri veri tabanından alacağız.

Yukarıdaki programa **şu satırı ekliyoruz**:

```
veriler = im.fetchall()
```

Burada da ilk defa gördüğümüz bir metot var: **fetchall()**. Gördüğünüz gibi **fetchall()** imlecin bir metodudur. Yukarıda gördüğümüz **SELECT * FROM** faturalar komutu '**faturalar**' adlı tablodaki bütün verileri seçiyordu. **fetchall()** metodu ise seçilen bu verileri alma işlevi görüyor.

Yukarıda biz, **`fetchall()`** metoduyla aldığımız bütün verileri **`veriler`** adlı bir değişkene atadık.

Artık bu verileri rahatlıkla yazdırabiliriz:

```
print(veriler)
```

Dilerseniz betiğimizi **topluca görelim**:

```
import sqlite3

vt = sqlite3.connect('vt.sqlite')

im = vt.cursor()

im.execute("""CREATE TABLE IF NOT EXISTS faturalar
(fatura, miktar, ilk_odeme_tarihi, son_odeme_tarihi)""")

im.execute("""INSERT INTO faturalar VALUES
("Elektrik", 45, "23 Ocak 2020", "30 Ocak 2020")""")

vt.commit()

im.execute("""SELECT * FROM faturalar""")

veriler = im.fetchall()

print(veriler)
```

Bu betiği **ilk kez çalıştırdığımızda** şöyle bir **çıktı** alırız:

```
[('Elektrik', 45, '23 Ocak 2020', '30 Ocak 2020')]
```

Gördüğünüz gibi veriler bir liste içinde **demet hâlinde** yer alıyor. Ama tabii siz bu verileri istediğiz gibi biçimlendirecek kadar Python bilgisine sahipsiniz. Ayrıca programı her çalıştırdığınızda **`INSERT INTO`** sorgusu tekrar işletileceği için verilerin tabloya **tekrar tekrar** yazılacağını, bu verileri alırken de **çıktı listesinin büyüyeceğini unutmayın**.

Peki, eğer siz bir veri tabanı dosyasına verilerin yalnızca bir kez yazılmasını istiyorsanız ne yapacaksınız? Yani mesela yukarıdaki kodlarda şu sorgu yalnızca **tek bir kez işletilsin**:

```
im.execute("""INSERT INTO faturalar VALUES
("Elektrik", 45, "23 Ocak 2020", "30 Ocak 2020")""")
```

Böylece veri tabanını her çalıştırdığınızda **`("Elektrik", 45, "23 Ocak 2020", "30 Ocak 2020")`** satırı dosyaya **tekrar tekrar yazdırılmasın**.

Bunu şu kodlarla hâlledebilirsiniz:

```
import sqlite3, os

dosya = 'vt.sqlite'
dosya_mevcut = os.path.exists(dosya)

vt = sqlite3.connect(dosya)
im = vt.cursor()

im.execute("""CREATE TABLE IF NOT EXISTS faturalar
(fatura, miktar, ilk_odeme_tarihi, son_odeme_tarihi)""")

if not dosya_mevcut:
    im.execute("""INSERT INTO faturalar VALUES
    ("Elektrik", 45, "23 Ocak 2020", "30 Ocak 2020")""")
    vt.commit()

im.execute("""SELECT * FROM faturalar""")

veriler = im.fetchall()
print(veriler)
```

Burada kodlarımızın en başında '**`vt.sqlite`**' adlı bir veri tabanının **mevcut olup olmadığını kontrol ediyoruz** (*zira eğer ortada bir veri tabanı dosyası yoksa veri de yok demektir*):

```
dosya_mevcut = os.path.exists(dosya)
```

Eğer böyle bir dosya mevcut değilse (*dolayısıyla veri mevcut değilse*) **`INSERT INTO`** sorgusu işletilerek **gerekli veriler yerine yerleştirilecek**:

```
if not dosya_mevcut:
    im.execute("""INSERT INTO faturalar VALUES
    ("Elektrik", 45, "23 Ocak 2020", "30 Ocak 2020")""")
    vt.commit()
```

Eğer böyle bir dosya zaten mevcutsa bu sorgu işletilemeyecek, onun yerine doğrudan **`SELECT`** sorgusuna geçilecek. Böylece **değerler veri tabanına bir kez işlendikten sonra**, **programımız aynı verileri tekrar tekrar veri tabanına yerleştirmeye çalışmayacak**.

Bu arada, daha önce de belirttiğimiz gibi tablo oluştururken **sütun adlarında boşluk** (*ve Türkçe karakter*) **kullanmak iyi bir fikir değildir**. Mesela **`ilk ödeme tarihi`** yerine **`ilk_odeme_tarihi`** ifadesini **tercih edin**. Eğer kelimeler arasında mutlaka boşluk bırakmak isterseniz **bütün kelimeleri tırnak içine alın**. Mesela: "**`ilk odeme tarihi`**" veya "**`ilk ödeme tarihi`**" gibi.

Yukarıda gördüğünüz gibi **`fetchall()`** metodu, bir veri tabanından **`SELECT`** ile seçtiğimiz bütün verileri **önümüze getiriyor**. Eğer seçilen verilerden **kaç tanesini almak istediğinizi kendiniz belirlemek istiyorsanız** **`fetchall()`** yerine **`fetchone()`** veya **`fetchmany()`** metotlarından o anki amacınıza uygun olanını kullanmayı tercih edebilirsiniz. Birazdan **`fetchone()`** ve **`fetchmany()`** metotlarından da söz edeceğiz.

Gelin, isterseniz şimdi **`fetchall()`** metodunu kullanarak veri tabanlarından *veri çekme konusunda biraz alıştırma yapalım*. Bu alıştırmalar için daha önce söz ettiğimiz ve bilgisayarımıza indirdiğimiz '**`kitaplar.sqlite`**' adlı **örnek veri tabanını** kullanacağız.

Öncelikle **veri tabanına bağlanalım** ve **bir imleç** oluşturalım:

```
>>> import sqlite3
>>> vt = sqlite3.connect('kitaplar.sqlite')
>>> im = vt.cursor()
```

Şimdi bu veri tabanındaki tabloyu seçeceğiz.

Peki, ama seçeceğimiz tablonun adını nereden bileceğiz? Hatırlarsanız, bir tablodaki bütün verileri seçebilmek için şu **SQL sorgusunu kullanıyorduk**:

```
"SELECT * FROM tablo_adı"
```

İşte bu sorguda '**tablo_adı**' kısmına ne geleceğini bulabilmek için şu komutları kullanabilirsiniz:

```
>>> import sqlite3
>>> vt = sqlite3.connect('kitaplar.sqlite')
>>> im = vt.cursor()
>>> im.execute("SELECT name FROM sqlite_master")
>>> im.fetchall()
```

Burada **şu satıra dikkat edin**:

```
>>> im.execute("SELECT name FROM sqlite_master")
```

Bütün **Sqlite** veri tabanlarında, ilgili veri tabanının şemasını gösteren '**sqlite_master**' adlı bir **tablo** bulunur. İşte bu tabloyu sorgulayarak **veri tabanı hakkında bilgi edinebiliriz**. Yukarıdaki örnekte, bu '**sqlite_master**' tablosunun '**name**' (*isim*) niteliğini sorguladık.

Bu sorgu bize **şu cevabı verdi**:

```
>>> im.fetchall()

[('kitaplar',)]
```

Demek ki **kitaplar.sqlite** adlı veri tabanında '**kitaplar**' adlı tek bir tablo varmış.

Gelin şimdi, bu bu **tablodaki bütün verileri alalım**:

```
>>> im.execute("SELECT * FROM kitaplar")
>>> im.fetchall()
```

Bu şekilde tablo içinde ne kadar veri varsa hepsini ekrana yazdırdık. Ancak tabii ki bir veri tabanının tamamını bir anda yazdırmak her zaman iyi bir fikir olmayabilir. Eğer **veri tabanının içinde milyonlarca girdi varsa bütün verilerin seçilip yazdırılması mantıklı olmayacaktır**. Gelin o hâlde, şimdi bizim seçilen verilerin ne kadarını çekeceğimizi belirleyebilmemizi sağlayacak metotları inceleyelim.

FETCHONE() METODU

`fetchone()` metodu, bir veri tabanından seçilen verilerin **tek tek alınabilmesine** izin verir.

Bu metodun nasıl kullanılacağını '**`kitaplar.sqlite`**' adlı örnek veri tabanımız üzerinden inceleyelim:

Önce **veri tabanına bağlanalım**:

```
>>> import sqlite3
>>> vt = sqlite3.connect('kitaplar.sqlite')
>>> im = vt.cursor()
```

Şimdi '**kitaplar**' adlı tablodan bütün verileri seçelim:

```
>>> im.execute("""SELECT * FROM kitaplar""")

<sqlite3.Cursor object at 0x003C2D20>
```

Artık seçtiğimiz **verileri tek tek almaya başlayabiliriz**:

```
>>> im.fetchone()

('UZMANLAR İÇİN PHP', 'Mehmet Şamlı', '33,00 TL')
```

Bir tane daha alalım:

```
>>> im.fetchone()

('ADOBE AİR', 'Engin Yöğen', '28,00 TL')
```

İki tane daha...

```
>>> im.fetchone()

('WEB TASARIM REHBERİ', 'Mustafa Aydemir', '38,50 TL')

>>> im.fetchone()

('ORACLE 11g R2', 'Teoman Dinçel', '34,00 TL')
```

`fetchone()`'ın gayet **faydalı bir metot** olduğu her hâlinden belli...

FETCHMANY() METODU

Bu metot, bir veri tabanından seçtiğiniz verilerin istediğiniz kadarını alabilmenize imkân tanır. **Dikkatlice bakın**:

```
>>> im.fetchmany(5)

[('AS 3.0 İLE SUNUCU PROGRAMLAMA', 'Engin Yöğen', '24,00 TL'),
 ('HACKING INTERFACE', 'Hamza Elbahadır', '25,00 TL'),
 ('JAVA VE JAVA TEKNOLOJİLERİ', 'Tevfik Kızılören', '45,00 TL'),
 ('XML VE İLERİ XML TEKNOLOJİLERİ', 'Musa Çiçek', '24,50 TL'),
 ('GRAFİK&ANİMASYON', 'Anonim', '18,50 TL')]
```

Gördüğünüz gibi **beş ögeden oluşan bir liste** elde ettik.

Böylece bir veri tabanından seçilen verileri almanın **farklı yöntemlerini** öğrenmiş olduk. Bu metotların dışında, eğer arzu ederseniz **`for` döngüsünden yararlanarak da veri çekebilirsiniz**. Bunun için herhangi bir metot kullanmanıza gerek yok:

```
>>> for veri in im:
...     print(veri)
```

Gördüğünüz gibi **`for` döngüsünü doğrudan imleç üzerinde kuruyoruz**.

Eğer amacınız alınacak verilerin sayısını **sınırlamaksa** yine `for` döngüsünden ve **`fetchone()`** metodundan birlikte yararlanabilirsiniz:

```
>>> for i in range(5):
...     print(im.fetchone())
```

Biraz sonra veri süzmeyi öğrendiğimizde, bir veri tabanından **veri seçip almanın daha verimli yollarını göreceğiz**.

Veri Süzme

Daha önce bir **Sqlite veri tabanında**, belli bir tablo içindeki bütün verileri seçmek için şu **SQL komutunu** kullanmamız gerektiğini öğrenmiştik:

```
SELECT * FROM tablo_adi
```

Ancak amacımız **çoğu zaman bir tablo içindeki bütün verileri seçmek olmayacaktır**. Programcılık maceramız boyunca genellikle yalnızca belli ölçütlere uyan verileri seçmek isteyeceğiz. Zira **içinde milyonlarca veri barındırabilecek olan veri tabanlarındaki verilerin tamamını seçmek akıl kârı değildir**.

Verileri süzme işini **`WHERE`** adlı bir SQL komutu yardımıyla gerçekleştireceğiz. Bu SQL komutunun **söz dizimi** şöyle:

```
SELECT * FROM tablo_adı WHERE sütun_başlığı = aranan_veri
```

Gördüğünüz gibi bu sorguyu gerçekleştirebilmek için **tablodaki sütun başlıklarını bilmemiz gerekiyor**.

Önceki sayfalarda, '**kitaplar.sqlite**' adlı veri tabanımızdaki tabloların adını nasıl öğrenebileceğimizi anlatmıştık. Hatırlarsanız **bu iş için şu komutu kullanıyorduk**:

```
>>> im.execute("SELECT name FROM sqlite_master")
```

Bu şekilde, bütün Sqlite veri tabanlarında bulunan '**sqlite_master**' adlı özel bir tablonun '**name**' niteliğini sorgulayarak elimizdeki veri tabanında bulunan tabloların adını elde edebiliyoruz. Adını öğrendiğimiz tablodaki **sütun başlıklarını elde etmek için** yine buna benzer bir komuttan yararlanacağız. **Dikkatlice bakın**:

```
>>> im.execute("SELECT sql FROM sqlite_master").fetchone()

('CREATE TABLE "kitaplar"
(\n\t`KitapAdi`\tTEXT,\n\t`Yazar`\tTEXT,\n\t`Fiyati`\tTEXT\n)',)
```

'**sqlite_master**' adlı tablonun '**sql**' niteliğini sorguladığımızda, ilgili tabloyu oluşturmak için kullanılan **SQL komutunu** görüyoruz. Bu komuta dikkatli bakarsanız tablonun '**KitapAdi**', '**Yazar**' ve '**Fiyati**' olmak üzere **üç sütundan** oluştuğunu göreceksiniz.

Sütun adlarını öğrendiğimize göre gelin şimdi, **yazar adına göre veri tabanında bir sorgu yapalım**:

```
>>> im.execute("SELECT * FROM kitaplar WHERE Yazar = 'Fırat Özgül'")
```

Burada **sorguyu nasıl kurduğumuza dikkat edin**. Bu sorgunun ilk kısmı olan **SELECT * FROM** kitaplar ifadesini zaten daha önce öğrenmiştik. Yeni olan kısım **WHERE Yazar = 'Fırat Özgül'**. Burada da anlaşılmayacak bir şey yok. Bu şekilde, veri tabanındaki '**kitaplar**' tablosunun '**Yazar**' sütununda '**Fırat Özgül**' bulunan **bütün kayıtları seçiyoruz**.

Şimdi de seçtiğimiz bu verileri alalım:

```
>>> im.fetchall()

('HERYÖNÜYLE PYTHON', 'Fırat Özgül', '34,00 TL')
```

Gayet başarılı... Bu arada, verileri **almak** için **`fetchall()`** yerine **`for`** döngüsü kullanabileceğinizi de biliyorsunuz:

```
>>> for s in im:
...     print(s)
...
('HERYÖNÜYLE PYTHON', 'Fırat Özgül', '34,00 TL')
```

İmleç üzerinde **`for`** döngüsü kurabildiğimize göre **yıldızlı parametrelerden de yararlanabileceğimizi** tahmin etmişsinizdir:

```
>>> print(*im)

('HERYÖNÜYLE PYTHON', 'Fırat Özgül', '34,00 TL')
```

Veri Tabanı Güvenliği

Python'da veri tabanları ve Sqlite konusunda daha fazla ilerlemeden önce **çok önemli bir konudan bahsetmemiz gerekiyor**. Tahmin edebileceğiniz gibi veri tabanı denen şey oldukça **hassas** bir konudur. Bilgiyi bir araya toplayan bu sistem, içerideki bilgilerin **değerine** ve **önemine** de bağlı olarak üçüncü şahısların iştahını kabartabilir. Ancak depoladığınız verilerin ne kadar değerli ve önemli olduğundan bağımsız olarak **veri tabanı güvenliğini sağlamak, siz programcıların asli görevidir**.

Peki, veri tabanı yönetim sistemleri acaba hangi tehditlerle karşı karşıya?

SQL komutlarını işleten bütün veritabanları için günümüzdeki en büyük tehditlerden birisi hiç kuşkusuz **kötü niyetli kişilerin veri tabanınıza SQL komutu sızdırma** (*SQL injection*) **girişimleridir**.

Şimdi, şöyle bir şey düşünün: Diyelim ki siz bir alışveriş karşılığı birine 100.000 TL'lik bir çek verdiniz. Ancak çeki verdiğiniz kişi bu çek üzerindeki miktarı tahrif ederek artırdı ve banka da tahrif edilerek artırılan bu miktarı çeki getiren kişiye (hamiline) ödedi. Böyle bir durumda epey başınız ağrıyacaktır.

İşte böyle tatsız bir durumla karşılaşmamak için çek veren kişi çekin üzerindeki miktarı hem *rakamla* hem de *yazıyla* belirtmeye özen gösterir. Ayrıca **rakam** ve **yazılara ekleme yapılmasını da engellemek için rakam ve yazıların sağına soluna** "#" **gibi işaretler de koyar**. Böylece çeki alan kişinin, kendisine izin verilenden daha fazla bir miktarı yazmasını engellemeye çalışır.

Yukarıdakine benzer bir şey, veri tabanı uygulamalarında da karşımıza çıkabilir. **Şimdi şu örneğe bakalım**:

```
import sqlite3

#vt.sqlite adlı bir veri tabanı dosyası oluşturup
#bu veri tabanına bağlanıyoruz.
db = sqlite3.connect("vt.sqlite")

#Veri tabanı üzerinde istediğimiz işlemleri yapabilmek
#için bir imleç oluşturmamız gerekiyor.
im = db.cursor()

#imlecin execute() metodunu kullanarak, veri tabanı içinde
#"kullanicilar" adlı bir tablo oluşturuyoruz. Bu tabloda
#kullanıcı_adi ve parola olmak üzere iki farklı sütun var.
im.execute("""CREATE TABLE IF NOT EXISTS kullanicilar
    (kullanici_adi, parola)""")

#Yukarıda oluşturduğumuz tabloya yerleştireceğimiz verileri
#hazırlıyoruz. Verilerin liste içinde birer demet olarak
#nasıl gösterildiğine özellikle dikkat ediyoruz.
veriler = [
            ("ahmet123", "12345678"),
            ("mehmet321", "87654321"),
            ("selin456", "123123123")
          ]

#veriler adlı liste içindeki bütün verileri kullanicilar adlı
```

```
#tabloya yerleştiriyoruz. Burada tek ögeli bir demet
#tanımladığımıza dikkat edin: (i,)
for i in veriler:
    im.execute("""INSERT INTO kullanicilar VALUES %s""" %(i,))

#Yaptığımız değişikliklerin tabloya işlenebilmesi için
#commit() metodunu kullanıyoruz.
db.commit()

#Kullanıcıdan kullanıcı adı ve parola bilgilerini alıyoruz...
kull = input("Kullanıcı adınız: ")
paro = input("Parolanız: ")

#Burada yine bir SQL komutu işletiyoruz. Bu komut, kullanicilar
#adlı tabloda yer alan kullanici_adi ve parola adlı sütunlardaki
#bilgileri seçiyor.
im.execute("""SELECT * FROM kullanicilar WHERE
kullanici_adi = '%s' AND parola = '%s'"""%(kull, paro))

#Hatırlarsanız daha önce fetchall() adlı bir metottan
#söz etmiştik. İşte bu fetchone() metodu da ona benzer.
#fetchall() bütün verileri alıyordu, fetchone() ise
#verileri tek tek alır.
data = im.fetchone()

#Eğer data adlı değişken False değilse, yani bu
#değişkenin içinde bir değer varsa kullanıcı adı
#ve parola doğru demektir. Kullanıcıyı içeri alıyoruz.
if data:
    print("Programa hoşgeldin {data[0]}!")

#Aksi halde kullanıcıya olumsuz bir mesaj veriyoruz.
else:
    print("Parola veya kullanıcı adı yanlış!")
```

Yukarıdaki kodları çalıştırdığınızda, eğer **kullanıcı adı** ve **parolayı doğru girerseniz** '`Programa hoşgeldin`' çıktısını göreceksiniz. Eğer kullanıcı adınız veya parolanız **yanlışsa** bununla ilgili bir **uyarı alacaksınız**.

Her şey iyi hoş, ama bu kodlarda *çok ciddi bir problem var*.

Dediğimiz gibi bu kodlar çalışırken (*teoride*) eğer kullanıcı, veri tabanında varolan bir **kullanıcı adı** ve **parola yazarsa sisteme kabul edilecektir**. Eğer doğru **kullanıcı adı** ve **parola girilmezse sistem kullanıcıya giriş izni vermeyecektir**. *Ama acaba gerçekten öyle mi?*

Şimdi, yukarıdaki programı **tekrar çalıştırın**. Kullanıcı adı ve parola sorulduğunda da her ikisi için şunu yazın:

```
x' OR '1' = '1
```

O da ne! **Program sizi içeri aldı...** Hem de kullanıcı adı ve parola doğru olmadığı hâlde. Hatta şu kodu sadece kullanıcı adı kısmına girip **parola kısmını boş bırakmanız da sisteme giriş hakkı elde etmenize yetecektir**:

```
x' OR '1' = '1' --
```

İşte, yukarıda gösterdiğimiz bu işleme "**SQL sızdırma**" (*SQL injection*) adı verilir. Kullanıcı, tıpkı en başta verdiğimiz tahrif edilmiş çek örneğinde olduğu gibi sistemin zaaflarından yararlanarak elde etmeye hakkı olandan **daha fazlasına erişim hakkı** elde ediyor.

Burada en basit şekliyle **`bool`** işleçlerinden biri olan **`or`**'dan yararlanıyoruz. **`or`**'un nasıl işlediğini gayet iyi biliyorsunuz ama ben yine de birkaç örnekle **`or`**'un ne olduğunu ve ne yaptığını size hatırlatayım. **Şu örneklere bakın**:

```
>>> a = 21

>>> a == 22

False

>>> b = 13

>>> b == 13

True
```

```
>>> if a == 22 and b == 13:
...     print("Merhaba!")
...

>>> if a == 22 or b == 13:
...     print("Merhaba!")
...
Merhaba!
```

Örneklerden de gördüğünüz gibi **and** işlecinin **True** sonucunu verebilmesi için her iki önermenin de **doğru** olması gerekir. O yüzden **a == 22 and b == 13** gibi bir ifade **False** değeri veriyor. Ancak **or** işlecinin **True** sonucu verebilmesi için iki önermeden **sadece birinin doğru olması yeterlidir**. Bu yüzden, sadece **b == 13** kısmı **True** olduğu hâlde **a == 22 or b == 13** ifadesi **True** sonucu veriyor... İşte biz de yukarıdaki SQL sızdırma girişiminde **or**'un bu özelliğinden faydalanıyoruz.

Dilerseniz neler olup bittiğini daha iyi anlayabilmek için sızdırılan kodu doğrudan **ilgili satıra uygulayalım**:

```
im.execute("""SELECT * FROM kullanicilar WHERE
kullanici_adi = 'x' OR '1' = '1' AND parola = 'x' OR '1' = '1'""")
```

Sanırım bu şekilde neler olup bittiği **daha net** görülüyor. Durumu biraz daha netleştirmek için Python'ı yardıma çağırabiliriz:

```
>>> kullanici_adi = 'ahmet123'
>>> parola = '12345678'
>>> kullanici_adi == 'x'
False

>>> '1' == '1'
True

>>> kullanici_adi == 'x' or '1' == '1'
True

>>> parola == 'x'
False
```

```
>>> (kullanici_adi == 'x' or '1' == '1') and (parola == 'x' or '1'
== '1')
True
```

'1' == '1' ifadesi her zaman **True** değeri verecektir. **Dolayısıyla kullanıcı adının ve parolanın doğru olup olmaması hiçbir önem taşımaz**. Yani her zaman **True** değerini vereceği kesin olan ifadeler yardımıyla yukarıdaki gibi bir **sızdırma girişiminde bulunabilirsiniz**.

Yukarıda yaptığımız şey, **'%s'** ile gösterilen yerlere kötü niyetli bir **SQL komutu sızdırmaktan ibarettir**. Burada zaten başlangıç ve bitiş tırnakları olduğu için sızdırılan kodda başlangıç ve bitiş tırnaklarını yazmıyoruz. O yüzden sızdırılan kod şöyle görünüyor:

```
x' OR '1' = '1
```

Gördüğünüz gibi **x**'in başındaki ve **1**'in sonundaki **tırnak işaretleri koymuyoruz**.

Peki, yukarıda verdiğimiz şu kod nasıl çalışıyor:

```
x' OR '1' = '1' --
```

Python'da yazdığımız kodlara yorum eklemek için "#" işaretinden yararlandığımızı biliyorsunuz. İşte SQL kodlarına yorum eklemek için de "**--**" işaretlerinden yararlanılır. Şimdi, dilerseniz yukarıdaki kodu **doğrudan ilgili satıra uygulayalım ve ne olduğunu görelim**:

```
im.execute("""SELECT * FROM kullanicilar WHERE
kullanici_adi = 'x' OR '1'='1' --AND parola = '%s'""")
```

Burada yazdığımız "**--**" işareti AND **parola = '%s'** kısmının sistem tarafından **yorum olarak** algılanmasını sağlıyor. Bu yüzden kodların bu kısmı işletilmiyor. Dolayısıyla da sisteme giriş yapabilmek için sadece **kullanıcı adını girmemiz yeterli oluyor**. Burada ayrıca kodlarımızın çalışması için **1**'in sonuna **bir adet tırnak yerleştirerek kodu kapattığımıza dikkat edin**. *Çünkü normal bitiş tırnağı yorum tarafında kaldı*.

Dikkat ederseniz SQL sızdırdığımızda "**ahmet123**" adlı kullanıcının hesabını ele geçirmiş olduk. Peki, neden ötekiler değil de "**ahmet123**"? Bunun sebebi, "**ahmet123**" hesabının **tablonun en başında yer alması**. Eğer tablonun başında "**admin**" diye bir hesap olmuş olsaydı veri tabanına **azami düzeyde zarar verme imkânına kavuşacaktınız**.

Peki, SQL sızdırma girişimlerini nasıl önleyeceğiz? Bu girişime karşı alabileceğiniz başlıca önlem "**%s**" işaretlerini kullanmaktan kaçınmak olacaktır. Bu işaret yerine "**?**" işaretini kullanacaksınız. **Yani yukarıdaki programı şöyle yazacağız**:

```
import sqlite3

db = sqlite3.connect("vt.sqlite")

im = db.cursor()

im.execute("""CREATE TABLE IF NOT EXISTS kullanicilar
    (kullanici_adi, parola)""")

veriler = [
            ("ahmet123", "12345678"),
            ("mehmet321", "87654321"),
            ("selin456", "123123123")
          ]

for i in veriler:
    im.execute("""INSERT INTO kullanicilar VALUES (?, ?)""", i)

db.commit()

kull = input("Kullanıcı adınız: ")
paro = input("Parolanız: ")

im.execute("""SELECT * FROM kullanicilar WHERE
kullanici_adi = ? AND parola = ?""", (kull, paro))

data = im.fetchone()

if data:
```

```
    print("Programa hoşgeldin {}!".format(data[0]))
else:
    print("Parola veya kullanıcı adı yanlış!")
```

Dediğimiz gibi, SQL sızdırma girişimlerine karşı alabileceğiniz **başlıca önlem** "**%s**" işaretleri yerine "**?**" **işaretini kullanmak olmalıdır**. Bunun dışında, SQL komutlarını işletmeden önce **bazı süzgeçler uygulamak da güvenlik açısından işinize yarayabilir**. *Örneğin kullanıcıdan alınacak verileri alfanümerik karakterlerle sınırlayabilirsiniz*:

```
if kull.isalnum() and paro.isalnum():
    im.execute("""SELECT * FROM kullanicilar WHERE
    kullanici_adi = '%s' AND parola = '%s'""" %(kull, paro))
```

Böylece kullanıcının bazı "**tehlikeli**" karakterleri girmesini engelleyebilir, onları sadece harf ve sayı girmeye zorlayabilirsiniz.

Her hâalükârda unutmamamız gereken şey, *güvenliğin çok boyutlu bir kavram olduğudur*. Birkaç önlemle pek çok güvenlik açığını engelleyebilirsiniz, ancak bütün güvenlik açıklarını bir çırpıda yamamak pek mümkün değildir. Bir programcı olarak sizin göreviniz, yazdığınız programları **güvenlik açıklarına karşı sürekli taramak ve herhangi bir açık ortaya çıktığında da bunu derhâl kapatmaya çalışmaktır**.

NESNE TABANLI PROGRAMLAMA

27

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 Türkçede **NTP** olarak kısaltalım. Şimdilik bu Nesne Tabanlı Programlama'nın ne olduğu ve tanımı bizi ilgilendirmiyor. Biz şimdilik işin teorisiyle pek uğraşmayıp pratiğine bakacağız. NTP'nin **pratikte nasıl işlediğini anlarsak teorisini araştırıp öğrenmek de daha kolay olacaktır**.

Neden Nesne Tabanlı Programlama?

İsterseniz önce kendimizi biraz yüreklendirip cesaretlendirelim. **Şu soruyu soralım kendimize**: *Nesne Tabanlı Programlama'ya hiç girmesem olmaz mı?* Bu soruyu cevaplandırmadan önce bakış açımızı şöyle belirleyelim. Daha doğrusu bu soruyu iki farklı açıdan inceleyelim. **NTP'yi öğrenmek ve NTP'yi kullanmak...** Eğer yukarıdaki soruya NTP'yi kullanmak penceresinden bakarsak cevabımız **evet** olacaktır. Yani NTP'ye hiç girmeseniz de olur. Bu bakımdan NTP'yle ilgilenmek istemeyebilirsiniz. Çünkü Python başka bazı dillerin aksine **NTP'yi dayatmaz**. İyi bir Python programcısı olmak için NTP'yi **kullanmak zorunda değilsiniz**. NTP'ye bulaşmadan da gayet başarılı programlar yazabilirsiniz. Bu bakımdan önünüzde bir engel yok.

Ama eğer yukarıdaki soruya NTP'yi öğrenmek penceresinden bakarsak cevabımız **hayır** olacaktır. Yani NTP'ye girmeseniz olmaz. Bu bakımdan NTP'yle ilgilenmeniz gerekir, *çünkü siz NTP'yi kullanmasanız da başkaları bunu kullanıyor*. Dolayısıyla NTP'nin bütün erdemlerini bir kenara bıraksak dahi sırf başkalarının yazdığı kodları anlayabilmek için bile olsa **elinizi NTP'ye bulaştırmanız gerekecektir**.

Bir de şöyle düşünün: Gerek İnternet 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 bilindiği anlamda NTP'nin faydalarından, bize getirdiği kolaylıklardan hiç bahsetmiyoruz. Zira şu anda içinde bulunduğumuz noktada bunları bilmenin **bize pek faydası dokunmayacaktır**. Çünkü daha NTP'nin ne olduğunu dahi bilmiyoruz ki güzel cümlelerle bize anlatılacak faydaları özümseyebilelim. NTP'yi öğrenmeye çalışan birine birkaç sayfa boyunca "*NTP şöyle iyidir, NTP böyle hoştur,*" demenin pek yararı olmayacaktır. Çünkü böyle bir çaba, konuyu anlatan kişiyi ister istemez okurun henüz bilmediği kavramları kullanarak bazı şeyleri açıklamaya çalışmaya itecek, bu da okurun zihninde birtakım bulanık cümlelerin uçuşmasından başka bir işe yaramayacaktır. Dolayısıyla NTP'nin faydalarını size burada bir çırpıda saymak yerine, **öğrenme sürecine bırakıyoruz bu özümseme işini**... NTP'yi öğrendikten sonra, bu programlama tekniğinin Python deneyiminize herhangi bir katkı sağlayıp sağlamadığına, size herhangi bir fayda getirip getirmediğine kendiniz karar vereceksiniz.

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

Python'da Nesne Tabanlı Programlama'ya ilişkin olarak **bilmemiz gereken ilk kavram sınıflardır**. Sınıfların görünürde yaptığı **en temel iş**, *fonksiyonları ve değişkenleri gruplandırabilmemizi sağlamaktır*. Sınıf oluşturabilmek için `class` adlı bir kelimeden yararlanacağız:

```
class Sınıf:
    ...
```

Sınıfımızı bu şekilde oluşturduktan sonra, artık bunun altına **değişkenlerimizi** ve **fonksiyonlarımızı** yazabiliriz. Mesela kodlarımız içinde farklı işlevlere ilişkin değişken ve fonksiyonları farklı sınıflar içinde tanımlayarak bunları **kendi aralarında gruplayabiliriz**:

```
class Sınıf1:
    değişken1 = 'değer1'

    def fonksiyon1_1():
        pass

    def fonksiyon1_2():
        pass

class Sınıf2:
    değişken2 = 'değer'

    def fonksiyon2_1():
        pass

    def fonksiyon2_2():
        pass
```

Bu perspektiften baktığımızda, sınıfların birbiriyle ilintili değişken ve fonksiyonları düzenli bir şekilde bir araya toplayabilmemizi sağlayan, böylece kodlarımızdaki dağınıklığı ortadan kaldıran **son derece faydalı bir araç olduğunu söyleyebiliriz**.

Mesela diyelim ki kodlarımız içinde başka işlemlere ilişkin değişken ve fonksiyonların yanı sıra **veri tabanı işlemleri ile ilgili birtakım fonksiyonlar** ve **değişkenler** de var:

```
veritabanı_konumu = '/home/firat/veritabani.db'

def bağlantı_kur():
    print("bağlantı kuruldu")

def bağlantıyı_kapat():
    print("bağlantı kapatıldı")

def sorgu_çalıştır():
    print("sorgu çalıştırıldı")
```

İşte veri tabanı ile ilgili bu fonksiyonları **ayrı bir sınıf hâlinde** kodlayarak bunları **kodların geri kalanından ayırabiliriz**:

```
class Veritabanı:
    veritabanı_konumu = '/home/firat/veritabani.db'

    def bağlantı_kur():
        print("bağlantı kuruldu")

    def bağlantıyı_kapat():
        print("bağlantı kapatıldı")

    def sorgu_çalıştır():
        print("sorgu çalıştırıldı")
```

Bu bakımdan sınıflar kodlarımızın en azından **daha derli toplu** görünmesini sağlayabilir. Ancak elbette sınıfların tek işlevi, **kod metni içindeki fonksiyon ve değişkenleri bir çatı altında toplamak değildir**. Zira eğer tek amacımız birbirinden farklı işlevleri yerine getiren fonksiyon ve değişkenleri kendi içinde gruplandırmak ise birbiriyle ilgili değişken ve fonksiyonları ayrı dosyalar içinde de tutabiliriz. Bunun için NTP'ye bulaşmamıza gerek yok. Ama s**ınıfların daha önemli bazı başka özellikleri ve kabiliyetleri de bulunur**. Birazdan bunlardan söz edeceğiz.

Nesneler

Başından beri Nesne Tabanlı Programlama ile ilgili bir şey mutlaka dikkatinizi çekmiş olmalı. Sürekli "**nesne**" deyip duruyoruz, ama *acaba "nesne" derken neyi kastediyoruz?*

Python programlama dili ile ilgili sağda solda şöyle bir cümle duymuş olabilirsiniz:

Python'da her şey bir nesnedir!

Everything is an object in Python!

Peki, bu cümle tam olarak ne anlama geliyor? Burada "nesne" denirken ne kastediliyor?

Programlama açısından bir şeyin bir '**nesne**' olması demek, o şeyin basit bir değer olmayıp birtakım **metot** ve **niteliklere sahip bir öge olması demektir**. Mesela, **Python'da karakter dizileri birer nesne**dir. Yani biz bir karakter dizisi tanımladığımızda aslında sadece basit bir değer tanımlamış olmayız. Aynı zamanda pek çok metodu ve niteliği olan bir 'nesne' de tanımlamış oluruz. **Örneğin**:

```
>>> kardiz = 'elma'
```

İşte burada yaptığımız şey sadece değeri '`elma`' olan, `kardiz` **adlı basit bir değişken tanımlamaktan ibaret değildir**. Burada tanımladığımız şey bir '**nesnedir**'. Yani birtakım metot ve niteliklere sahip bir ögedir:

```
>>> dir(kardiz)

['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__
mod__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__
repr__',
'__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
```

```
'__subclasshook__', 'capitalize', 'casefold', 'center', 'count',
'encode',
'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index',
'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit',
'isidentifier',
'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle',
'isupper',
'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition',
'replace',
'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
'split',
'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate',
'upper', 'zfill']
```

Dolayısıyla "**nesne**" kavramı öyle dışardan göründüğü gibi çok egzantrik, matah veya anlaşılması zor bir şey değildir. **Bu kavram basit bir adlandırmadan ibarettir**.

İşte Python'daki **class** [sınıf] yapısı da bize yeni nesneler oluşturma imkânı verir.

Hatırlarsanız yukarıda şöyle bir **sınıf** tanımlamıştık:

```
class Veritabanı:
    veritabanı_konumu = '/home/firat/veritabani.db'

    def bağlantı_kur():
        print("bağlantı kuruldu")

    def bağlantıyı_kapat():
        print("bağlantı kapatıldı")

    def sorgu_çalıştır():
        print("sorgu çalıştırıldı")
```

Gelin şimdi, bu sınıftan bir **nesne türetelim**:

```
veritabanı = Veritabanı()
```

Kodlarımızın **son hâli** şöyle görünecek:

```
class Veritabanı:
    veritabanı_konumu = '/home/firat/veritabani.db'

    def bağlantı_kur():
        print("bağlantı kuruldu")

    def bağlantıyı_kapat():
        print("bağlantı kapatıldı")

    def sorgu_çalıştır():
        print("sorgu çalıştırıldı")

veritabanı = Veritabanı()
```

Burada veritabanı, **`Veritabanı`** adlı sınıftan üretilmiş bir nesnedir. Dolayısıyla bu nesnenin de **birtakım nitelik** ve **metotları** vardır. Bu durumu **doğrulamak için `veritabanı = Veritabanı()` satırının hemen altına şu kodu ekleyelim**:

```
print(dir(veritabanı))
```

Bu kodları içeren dosyayı çalıştırdığımızda şuna benzer bir **çıktı** alacağız:

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__
eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__le__', '__lt__', '__module__',
'__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__',
'bağlantı_kur',
'bağlantıyı_kapat', 'sorgu_çalıştır', 'veritabanı_konumu']
```

Gördüğünüz gibi yeni oluşturduğumuz **`veritabanı`** adlı nesnenin de tıpkı **başka nesneler gibi birtakım nitelik ve metotları var**. Bu listede yer alan **`veritabanı_konumu`** bir nitelik iken **`bağlantı_kur()`**, **`bağlantıyı_kapat()`** ve **`sorgu_çalıştır()`** birer metottur.

Dolayısıyla kabaca ifade etmek gerekirse Nesne Tabanlı Programlama'da bir sınıf içinde yer alan *değişkenlere* "**nitelik**", *fonksiyonlara* ise "**metot**" adı veriyoruz.

İsterseniz bu noktaya kadar anlattıklarımızı şöyle bir özetleyelim:

"**Nesne Tabanlı Programlama'**', nesnelerin kullanıldığı bir program yazma tarzıdır. Python'da programlarınızı yazarken bu tarzı kullanmak zorunda değilsiniz. Ama başkalarının yazdığı kodları anlayabilmek için bile olsa **NTP**'yi öğrenmenizde fayda var.

Peki 'nesne' nedir? **Nesne ise birtakım nitelikleri ve metotları olan ögelere verilen basit bir addan ibarettir**. Örneğin Python'daki karakter dizileri, sayılar, listeler, demetler, sözlükler ve hatta fonksiyonlar birer nesnedir. Yani bunların hepsinin birtakım **nitelikleri** ve **metotları** vardır.

Peki acaba Python'daki her şey gerçekten bir nesne midir? Aslında bu ifadenin biraz **abartılı** olduğunu söyleyebiliriz. Zira **Python'da nesne olmayan ögeler de vardır.** Mesela `if`, `for`, `continue`, `break` **gibi ögeler birer nesne değildir**. Yani bunların herhangi bir niteliği veya metodu bulunmaz.

Nesnelerin nitelik ve metotlarını şu şekilde sorgulayabileceğimizi biliyorsunuz:

```
>>> dir('')       #karakter dizileri
>>> dir([])       #listeler
>>> dir(())       #demetler
>>> dir({})       #sözlükler
>>> dir(range)    #fonksiyonlar
```

`if`, `for`, `continue`, `break` gibi ögeler ise **birer nesne olmadığı için `dir()` fonksiyonuna cevap vermez**:

```
>>> dir(if)
  File "<stdin>", line 1
    dir(if)
        ^
SyntaxError: invalid syntax
```

İşte **class** kelimesi de bize **yeni sınıflar** ve **dolayısıyla yeni nesneler üretme imkânı tanır**. Bu kelimeyi kullanarak bir sınıf tanımlayabiliriz:

```
class Sınıf:
    pass
```

Yukarıdaki, **boş bir sınıftır.** Ama istersek bu boş sınıfı birtakım **nitelik** ve **metotlarla** doldurabiliriz:

```
class Sınıf:
    nitelik = 'değer'

    def metot1():
        print("metot1")

    def metot2():
        print("metot2")
```

Gördüğünüz gibi Nesne Tabanlı Programlama bağlamında **değişkenlere** '**nitelik**', **fonksiyonlara** ise '**metot**' adı veriyoruz.

Bu sınıfı tanımladıktan sonra artık dilersek bu sınıftan bir **nesne** üretebiliriz:

```
sınıf = Sınıf()
```

Gelin, isterseniz burada terminoloji ile ilgili ufak bir bilgi verelim.

Esasında biz tanımladığımız **sınıfı** (*class*) çağırdığımız anda bir **nesne** (*object*) üretmiş oluruz:

```
Sınıf()
```

Yani bir nesne üretmek için illa **bunu bir değişkene atamamıza gerek yok**. Bu sınıfı bir değişkene atadığımızda ise o sınıfı '**örneklemiş**' (*instantiation*) oluyoruz. Yani **Sınıf()** adlı sınıfı **sınıf** adlı değişkene atayarak ilgili sınıfın bir **örneğini** (*instance*) veya **suretini** çıkarmış oluyoruz.

Burada bir noktaya özellikle dikkatinizi çekmek istiyorum. *Bu kavramların hiçbiri aslında anlaşılması zor, ekzantrik şeyler değildir*. **Nesne** (*object*), **sınıf** (*class*), **metot** (*method*), **nitelik** (*attribute*), **örnek** (*instance*) ve **örnekleme** (*instantiation*) kavramları tamamen basit birer isimlendirmeden ibarettir ve esasında yaptıkları işler son derece basittir. Dolayısıyla Nesne Tabanlı Programlamayla muhatap olurken yapmanız gereken ilk iş, **buna ilişkin kavramları gözünüzde büyütmemek olacaktır**...

Sınıfların Kullanımı

Gelin, bu noktada sınıfları nasıl **kullanacağımızı** ve **çalıştıracağımızı** anlamak için şimdiye kadar öğrendiklerimizi kullanarak *bazı ufak tefek deneme çalışmaları* yapalım. Mesela yukarıda gösterdiğimiz şu kodları önümüze alalım:

```
class Veritabanı:
    veritabanı_konumu = '/home/firat/veritabani.db'

    def bağlantı_kur():
        print("bağlantı kuruldu")

    def bağlantıyı_kapat():
        print("bağlantı kapatıldı")

    def sorgu_çalıştır():
        print("sorgu çalıştırıldı")
```

Bu kodları mesela **`deneme.py` adlı bir dosyaya kaydedelim**.

Yukarıda tanımladığımız sınıfı **çalıştırmanın** veya **kullanmanın ilk yöntemi bunu bir modül gibi içe aktarmaktır**. Bu **`modülü.py`** uzantılı başka bir kod dosyasının içine aktarabileceğimiz gibi **doğrudan etkileşimli kabuk içine de aktarabiliriz**. *Dilerseniz önce bunu etkileşimli kabuğa nasıl aktaracağımıza bakalım*.

Bunun için önce bu dosyanın bulunduğu konumda bir komut satırı açıp **Python'ın etkileşimli kabuğunu başlatalım**. Daha sonra da sınıfımızı içeren **deneme.py** adlı dosyayı bir **modül olarak içe aktaralım**:

```
>>> import deneme
```

Deneme adlı dosyayı bu şekilde '**deneme**' adıyla **içe aktardıktan sonra** mesela **dir()** fonksiyonuyla **bu modülün içinde hangi ögelerin olduğunu sorgulayabiliriz**:

```
>>> dir(deneme)

['Veritabanı', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__']
```

Listenin en başında **Veritabanı** adlı sınıfımızı görebiliyoruz. Gelin şimdi, bu sınıfımızı '**örnekleyelim**' (*instantiation*), yani bunu bir değişkene atayarak **Veritabanı** adlı **sınıfın bir örneğini** (*instance*) **çıkaralım**:

```
>>> vt = deneme.Veritabanı()
```

İşte burada '**vt**', Veritabanı adlı sınıftan üretilmiş bir **nesnedir** (*object*) ve her **nesne gibi birtakım nitelik ve metotları vardır**:

```
>>> dir(vt)

['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__
eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__le__', '__lt__', '__module__',
'__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__',
'bağlantı_kur',
'bağlantıyı_kapat', 'sorgu_çalıştır', 'veritabanı_konumu']
```

Bu listede sınıf içindeki **tüm nitelik** ve **metotlarımızı** görebiliyoruz. Mesela bunlar arasından **`veritabanı_konumu`** adlı niteliğe erişelim:

```
>>> vt.veritabanı_konumu
‘/home/firat/veritabani.db’
```

Gördüğünüz gibi niteliğimize çok rahat bir şekilde erişebildik. **Bir de metotlarımıza erişmeye çalışalım**. Mesela **`bağlantı_kur()`** metoduna ulaşalım:

```
>>> vt.bağlantı_kur()
```

Ama o da ne! Bu komut bize bir **hata mesajı** gösterdi:

```
Traceback (most recent call last):
  File “<stdin>”, line 1, in <module>
TypeError: bağlantı_kur() takes 0 positional arguments but 1 was
given
```

Birazdan bu durumun **nedenini** öğreneceğiz. Ama dilerseniz öncelikle sınıflarımızı bir de dosya içinde nasıl çalıştıracağımıza bakalım.

Diyelim ki elimizde içeriği şu olan **`deneme.py`** isimli bir **kod dosyası var**:

```
class Veritabanı:
    veritabanı_konumu = ‘/home/firat/veritabani.db’

    def bağlantı_kur():
        print(“bağlantı kuruldu”)

    def bağlantıyı_kapat():
        print(“bağlantı kapatıldı”)

    def sorgu_çalıştır():
        print(“sorgu çalıştırıldı”)
```

Bu sınıfı aynı dosya içinde çalıştırmak için öncelikle **sınıfımızı örnekleyelim**:

```
vt = Veritabanı()
```

`Veritabanı` adlı sınıfı **`Veritabanı()`** şeklinde çağırmak suretiyle, sınıfımızdan bir **nesne üretmiş olduk**. Bu nesneyi **`vt`** adlı bir değişkene atamak suretiyle ise sınıfımızı örneklemiş olduk. Burada **`vt`** adlı değişken, **`Veritabanı`** adlı **sınıfın bir örneğidir**.

Şimdi de **`Veritabanı`** adlı sınıfın bir niteliği olan **`veritabanı_konumu`** **ögesine erişelim**:

```
vt.veritabanı_konumu
```

Kodlarımızın **son hâli** şöyle görünecek:

```
class Veritabanı:
    veritabanı_konumu = '/home/firat/veritabani.db'

    def bağlantı_kur():
        print("bağlantı kuruldu")

    def bağlantıyı_kapat():
        print("bağlantı kapatıldı")

    def sorgu_çalıştır():
        print("sorgu çalıştırıldı")

vt = Veritabanı()
vt.veritabanı_konumu
```

Bu kodları bu şekilde **`deneme.py`** adıyla kaydettikten sonra, **bu dosyayı komut satırından herhangi bir Python programını çalıştırdığımız gibi çalıştırabiliriz**:

```
python deneme.py
```

Tıpkı kodlarımızı etkileşimli kabuktan çalıştırırken şahit olduğumuz gibi şu kodlar burada da **hata** verecektir:

```
vt.bağlantı_kur()
vt.bağlantıyı_kapat()
```

Zira bir sınıfın metotlarına dışarıdan erişebilmek için *bilmemiz gereken önemli bir konu daha var*.

SELF

Python'da bir sınıfın metotlarına dışarıdan erişebilmek için o metodun, kendine ait parametreler dışında, **en az bir adet ilave parametreye sahip olması gerekir**. Bu ilave parametreye de Python programlama camiasında geleneksel olarak **`self`** adı verilir:

```
class Veritabanı:
    veritabanı_konumu = '/home/firat/veritabani.db'

    def bağlantı_kur(self):
        print("bağlantı kuruldu")

    def bağlantıyı_kapat(self):
        print("bağlantı kapatıldı")

    def sorgu_çalıştır(self):
        print("sorgu çalıştırıldı")
```

Bu parametrenin adının **`self`** olması şart değildir. İsterseniz siz buna **`mahmut`** da diyebilirsiniz. Ama bu parametrenin **`self`** olarak adlandırılması camia içinde **son derece yerleşmiş bir gelenektir**. Biz de bu geleneği bozmayalım.

Bu **`self`** parametresi sayesinde **`Veritabanı`** adlı sınıfın metotlarına **dışarıdan erişebileceğiz**:

```
>>> vt.bağlantı_kur()
bağlantı kuruldu

>>> vt.bağlantıyı_kapat()
```

```
bağlantı kapatıldı

>>> vt.sorgu_çalıştır()
sorgu çalıştırıldı
```

Normalde parametre alan bir fonksiyonu *argümansız bir şekilde çağırdığımızda hata vereceğini biliyoruz*. Ama burada sınıf tanımı içinde metotların **`self`** adlı **bir parametresi olmasına rağmen, bunları kullanırken herhangi bir argüman belirtmediğimize dikkat edin**.

`self` parametresi ayrıca bir sınıfın metotlarına, o sınıfın başka metotları içinden erişebilmemizi de sağlar:

```
class Veritabanı:
    veritabanı_konumu = '/home/firat/veritabani.db'

    def bağlantı_kur(self):
        print("bağlantı kuruldu")

    def bağlantıyı_kapat(self):
        self.sorgu_çalıştır()
        print("bağlantı kapatıldı")

    def sorgu_çalıştır(self):
        print("sorgu çalıştırıldı")
```

Gördüğünüz gibi **`bağlantıyı_kapat()`** adlı metot içinde, sınıfın **`sorgu_çalıştır()`** adlı metodunu çalıştırdık. Bunun için **`self`** parametresinden yararlandığımıza dikkat edin.

__INIT__ METODU

Python'da sınıfların **`__init__()`** adlı **özel bir metodu** bulunur. **Bu metodun görevi**, bir sınıf çalıştırıldığında yani o sınıf **örneklendiğinde yapılacak işlemleri tanımlamaktır**. Bu metodun tam olarak ne işe yaradığını anlamak için çok basit bir **örnek** verelim:

```
class Deneme:
    def __init__(self):
        print("sınıf oluşturuldu")

deneme = Deneme()
```

Bu kodları bir dosyaya yazıp çalıştırdığımızda şu **çıktıyı** alırız:

```
sınıf oluşturuldu
```

Dolayısıyla eğer oluşturduğunuz bir sınıf örneklendiğinde **yapılmasını istediğiniz işlemler varsa** bu işlemleri, **`__init__()`** metodu içinde tanımlayabilirsiniz.

BASİT BİR OYUN

Gelin, isterseniz nesne denen kavramı daha iyi anlayabilmek, **bir nesneyi nesne yapan metot ve nitelikler arasındaki ilişkiyi daha net bir şekilde kavrayabilmek** ve **buraya kadar anlattıklarımızın zihnimize iyice yerleşmesini sağlamak için**, komut satırı üzerinde çalışan çok basit bir oyun tasarlayalım. Bu şekilde hem *eski bilgilerimizi tekrar etmiş oluruz* hem *teorik bilgilerimizi uygulama sahasına dökmüş oluruz* hem de *yeni şeyler öğrenmiş oluruz*.

Oyunumuzun kodları şöyle:

```
import time
import random
import sys

class Oyuncu:
```

```
def __init__(self, isim, can=5, enerji=100):
    self.isim = isim
    self.darbe = 0
    self.can = can
    self.enerji = enerji

def mevcut_durumu_görüntüle(self):
    print('darbe: ', self.darbe)
    print('can: ', self.can)
    print('enerji: ', self.enerji)

def saldır(self, rakip):
    print('Bir saldırı gerçekleştirdiniz.')
    print('Saldırı sürüyor. Bekleyiniz.')

    for i in range(10):
        time.sleep(.3)
        print('.', end='', flush=True)

    sonuç = self.saldırı_sonucunu_hesapla()

    if sonuç == 0:
        print('\nSONUÇ: kazanan taraf yok')

    if sonuç == 1:
        print('\nSONUÇ: rakibinizi darbelediniz')
        self.darbele(rakip)

    if sonuç == 2:
        print('\nSONUÇ: rakibinizden darbe aldınız')
        self.darbele(self)

def saldırı_sonucunu_hesapla(self):
     return random.randint(0, 2)

def kaç(self):
    print('Kaçılıyor...')
    for i in range(10):
        time.sleep(.3)
        print('\n', flush=True)
```

```
            print('Rakibiniz sizi yakaladı')

        def darbele(self, darbelenen):
            darbelenen.darbe += 1
            darbelenen.enerji -= 1
            if (darbelenen.darbe % 5) == 0:
                darbelenen.can -= 1
            if darbelenen.can < 1:
                darbelenen.enerji = 0
                print('Oyunu {self.isim} kazandı!')
                self.oyundan_çık()

        def oyundan_çık(self):
            print('Çıkılıyor...')
            sys.exit()

    ################################

    # Oyuncular
    siz = Oyuncu('Ahmet')
    rakip = Oyuncu('Mehmet')

    # Oyun başlangıcı
    while True:
        print('Şu anda rakibinizle karşı karşıyasınız.',
              'Yapmak istediğiniz hamle: ',
              'Saldır:  s',
              'Kaç:     k',
              'Çık:     q', sep='\n')

        hamle = input('\n> ')
        if hamle == 's':
            siz.saldır(rakip)

            print('Rakibinizin durumu')
            rakip.mevcut_durumu_görüntüle()

            print('Sizin durumunuz')
            siz.mevcut_durumu_görüntüle()
```

```
    if hamle == 'k':
        siz.kaç()

    if hamle == 'q':
        siz.oyundan_çık()
```

Komut satırı üzerinde çalışan basit bir oyundur bu. Dilerseniz bu kodları incelemeye başlamadan önce, bir dosyaya **kaydedip çalıştırın**. **Karşınıza şöyle bir ekran gelecek**:

```
Şu anda rakibinizle karşı karşıyasınız.
Yapmak istediğiniz hamle:
Saldır:  s
Kaç:     k
Çık:     q

>
```

Programımız bize burada **üç farklı seçenek** sunuyor. Eğer rakibimize saldırmak istiyorsak klavyedeki '**s**' tuşuna; rakibimizden kaçmak istiyorsak klavyedeki '**k**' tuşuna; yok eğer oyundan çıkmak istiyorsak da klavyedeki '**q**' tuşuna basacağız. Tercihinizi belirleyip neler olduğunu *inceleyin* ve *oyunu iyice tanımaya çalışın*.

Oyunu iyice anlayıp tanıdıktan sonra **oyun kodlarını incelemeye geçebiliriz**.

Yukarıda ilk olarak **`Oyuncu`** adlı bir **sınıf** tanımladık:

```
class Oyuncu:
    def __init__(self, isim, can=5, enerji=100):
        self.isim = isim
        self.darbe = 0
        self.can = can
        self.enerji = enerji
```

'**`class`**' kelimesinin **sınıf tanımlamamızı sağlayan bir araç**, '**`Oyuncu`**' kelimesinin ise t**anımladığımız sınıfın adı** olduğunu biliyoruz. Bu satırın hemen ardından gelen **`__init__()`** fonksiyonu, **sınıfımız örneklendiğinde neler olacağını tanımladığımız yerdir**.

Bu sınıfın, örnekleme sırasında **hangi parametreleri alacağını da `__init__()`** fonksiyonu içinde belirliyoruz. Parametre listesinde gördüğümüz ilk öge yani **`self`**, sınıfın o anki **örneğini** temsil ediyor. Python'ın söz dizimi kuralları gereğince bu kelimeyi oraya yazmamız gerektiğini biliyoruz.

Yukarıdaki fonksiyon, **`self`** dışında toplam **üç parametre alıyor**: **`isim`**, **`can`** ve **`enerji`**. Bunlardan ilki, yani **`isim`** parametresinin öntanımlı bir değeri yok. Dolayısıyla sınıfı çağırırken (*yani örneklerken*) bu **parametrenin değerini belirtmemiz gerekecek**. Öteki iki parametre olan **`can`** ve **`enerji`** ise birtakım öntanımlı değerlere sahip. Dolayısıyla sınıfı örneklerken **bu parametrelere farklı bir değer atamadığımız sürece bu parametreler, listede belirtilen değerleri taşıyacak**.

Parametre olarak belirlediğimiz değerleri sınıf içinde kullanabilmek için bunları, **`__init__()`** fonksiyonunun gövdesinde birer **örnek niteliğine** dönüştürüyoruz:

```
self.isim = isim
self.darbe = 0
self.can = can
self.enerji = enerji
```

Burada ilave olarak bir de değeri **`0`** olan **`self.darbe`** adlı bir **değişken** tanımladık. Bu da sınıfımızın örnek niteliklerinden biri olup **ilgili oyuncu** (*yani sınıfın o anki örneği*) **darbe aldıkça bunun değeri yükselecektir**.

Gelin, bu aşamada sınıfımızı **örnekleyerek** neler olup bittiğini daha net anlamaya çalışalım:

```
class Oyuncu:
    def __init__(self, isim, can=5, enerji=100):
        self.isim = isim
        self.darbe = 0
        self.can = can
        self.enerji = enerji

#Sınıfımızı örnekliyoruz
oyuncu = Oyuncu('Ahmet')
```

Burada **`oyuncu = Oyuncu('Ahmet')`** komutunu verdiğimiz anda **`__init__()`** fonksiyonu **çalışmaya başlıyor** ve **`oyuncu`** **adlı nesne için sırasıyla şu değişkenleri oluşturuyor**:

```
isim = 'Ahmet'
darbe = 0
can = 5
enerji = 100
```

Bu örnek niteliklerine nasıl ulaşabileceğinizi biliyorsunuz:

```
print('İsim: ', oyuncu.isim)
print('Darbe: ', oyuncu.darbe)
print('Can: ', oyuncu.can)
print('Enerji: ', oyuncu.enerji)
```

Başta da söylediğimiz gibi **`Oyuncu()`** sınıfını örnekleyerek meydana getireceğiniz bütün sınıf örnekleri, yani **nesneler**, **`__init__()`** fonksiyonu içinde tanımladığınız **örnek niteliklerini** taşıyacaktır:

```
class Oyuncu():
    def __init__(self, isim, can=5, enerji=100):
        self.isim = isim
        self.darbe = 0
        self.can = can
        self.enerji = enerji
```

```
oyuncu1 = Oyuncu('Ahmet')
oyuncu2 = Oyuncu('Mehmet')
oyuncu3 = Oyuncu('Veli')
oyuncu4 = Oyuncu('Ayşe')
```

Burada **oyuncu1**, **oyuncu2**, **oyuncu3** ve **oyuncu4** olmak üzere **dört farklı nesne oluşturduk**. Bu nesnelerin **hangi niteliklere sahip olacağını** ise **Oyuncu() sınıfının tanımı içinde belirttik**. Yani sınıfımız, tıpkı bir fabrika gibi çalışarak bizim için **aynı nitelikleri taşıyan dört farklı nesne üretti**.

İşte, nesne tabanlı programlamanın **özünü oluşturan** "**nesne**" **budur**. Bir nesnenin **hangi niteliklere sahip olacağını belirleyen veri tipine sınıf** (*class*) derken o sınıfın ortaya çıkardığı ürüne ise **nesne** (*object*) adını veriyoruz. **Bunu şuna benzetebilirsiniz**: Eğer '**İnsan**' bir sınıfsa; '**Mahmut**' bu sınıfın bir örneğidir. Dolayısıyla **Mahmut**, **İnsan** sınıfından üretilmiş bir nesnedir. Aynı şekilde eğer '**Köpek**' bir sınıfsa; '**Karabaş**' da bu sınıfın bir örneğidir. Yani Karabaş, **Köpek** sınıfından üretilmiş bir **nesnedir**. Mahmut'un hangi özelliklere sahip olacağını **İnsan** sınıfının nasıl tanımlandığı, Karabaş'ın hangi özelliklere sahip olacağını ise Köpek sınıfının nasıl tanımlandığı belirler. İşte aynı bunun gibi **Oyuncu()** sınıfından üretilen nesnelerin **hangi özelliklere sahip olacağını** da **Oyuncu() sınıfının nasıl tanımlandığı belirler**.

Kodlarımızı incelemeye devam edelim...

```
def mevcut_durumu_görüntüle(self):
    print('darbe: ', self.darbe)
    print('can: ', self.can)
    print('enerji: ', self.enerji)
```

Burada **mevcut_durumu_görüntüle()** adlı bir **örnek metodu** tanımladık. Örnek metotlarının ilk parametresinin her zaman **self** olması gerektiğini biliyoruz.

Tanımladığımız örnek metodunun görevi, **Oyuncu()** sınıfından oluşturduğumuz nesnelerin (*yani örneklerin*) o anki **darbe**, **can** ve **enerji** durumlarını görüntülemek. Birer örnek niteliği olan **darbe**, **can** ve **enerji** değişkenlerine **self** aracılığıyla eriştiğimize özellikle dikkat ediyoruz.

Gelelim sınıfımızın **önemli örnek metotlarından** biri olan **`saldır()`** fonksiyonunu incelemeye:

```
def saldır(self, rakip):
    print('Bir saldırı gerçekleştirdiniz.')
    print('Saldırı sürüyor. Bekleyiniz.')

    for i in range(10):
        time.sleep(.3)
        print('.', end='', flush=True)

    sonuç = self.saldırı_sonucunu_hesapla()

    if sonuç == 0:
        print('\nSONUÇ: kazanan taraf yok')

    if sonuç == 1:
        print('\nSONUÇ: rakibinizi darbelediniz')
        self.darbele(rakip)

    if sonuç == 2:
        print('\nSONUÇ: rakibinizden darbe aldınız')
        self.darbele(self)
```

Bu fonksiyon, **`self`** dışında **tek bir parametre alıyor**. Fonksiyonu çalıştırırken kullanacağımız **`rakip`** parametresi, saldırının kime karşı (*yani sınıf örneklerinden hangisine karşı*) düzenleneceğini belirleyecek.

Fonksiyon gövdesinde **ilk olarak şöyle bir kısım görüyoruz**:

```
print('Bir saldırı gerçekleştirdiniz.')
print('Saldırı sürüyor. Bekleyiniz.')

for i in range(10):
    time.sleep(.3)
    print('.', end='', flush=True)
```

Burada saldırının gerçekleştiğine dair kullanıcıyı bilgilendirdikten sonra **şöyle bir kod parçası yazdık**:

```
for i in range(10):
    time.sleep(.3)
    print('.', end='', flush=True)
```

Bu kodlarda **`time`** adlı bir standart kütüphane modülünün **`sleep()`** adlı bir metodundan **yararlandığımızı** görüyorsunuz. Elbette bu modülü kullanabilmek için öncelikle bu modülü **içe aktarmış olmamız gerekiyor**. Bu işlemi dosyanın en başında **`import time`** satırı yardımıyla gerçekleştirdiğimizi görebilirsiniz.

Yukarıdaki satırlar, 300'er milisaniye aralıklarla, yan yana nokta işaretleri yerleştirecektir. **Dilerseniz etkileşimli kabukta bu kodları şu şekilde test edebilirsiniz**:

```
>>> import time
>>> for i in range(10):
...     time.sleep(.3)
...     print('.', end='', flush=True)
```

`print()` fonksiyonu içinde kullandığımız **`flush`** parametresi, **`print()`** fonksiyonuyla verdiğimiz çıktıdaki **anlık değişimleri görebilmemizi sağlıyor**. Böylece bu kodlarda noktaların hareketini canlı bir şekilde görebiliyoruz. Bu parametrenin ne işe yaradığını anlamak için bu parametreyi çıkarıp **bu kodları bir de öyle çalıştırın**. Sonucun ne olduğunu takip ederek **`flush`** parametresinin görevini daha iyi anlayabilirsiniz.

Bu kodların ardından şöyle bir satır yazdık:

```
sonuç = self.saldırı_sonucunu_hesapla()
```

Burada, **`saldırı_sonucunu_hesapla()`** adlı bir örnek metodunu çağırdığımızı görüyorsunuz:

```
def saldırı_sonucunu_hesapla(self):
     return random.randint(0, 2)
```

Biraz önce **time** adlı bir standart kütüphane modülünü kullanmıştık. Şimdi ise **random** adlı başka bir **standart kütüphane modülünü** kullanıyoruz. Elbette bu modülü de kullanabilmek için öncelikle bu modülü **import random** komutuyla içe aktarmış olmamız gerekiyor. Bu zorunluluğu da tıpkı **time** modülünde olduğu gibi **dosyanın en başında yerine getirmiştik**.

Yukarıda **random** modülünü, 0 ile 2 arası **rastgele sayılar üretmek için kullandık**. **random.randint(0, 2)** komutu her çalıştığında 0, 1 ve 2 sayılarından birini **rastgele üretecektir**. Buradan elde ettiğimiz sonucu **sonuç** adlı bir değişkene atayarak **saldır()** fonksiyonu içinde şu şekilde kullanıyoruz:

```
sonuç = self.saldırı_sonucunu_hesapla()

if sonuç == 0:
    print('\nSONUÇ: kazanan taraf yok')

if sonuç == 1:
    print('\nSONUÇ: rakibinizi darbelediniz')
    self.darbele(rakip)

if sonuç == 2:
    print('\nSONUÇ: rakibinizden darbe aldınız')
    self.darbele(self)
```

Eğer **randint()** metodu **0 sayısını üretirse** rakibimize karşı gerçekleştirdiğimiz saldırının **sonuçsuz kaldığına hükmediyoruz**:

```
if sonuç == 0:
    print('\nSONUÇ: kazanan taraf yok')
```

Eğer **randint()** metodu 1 sayısını üretirse rakibimizi **başarıyla darbelediğimize**, 2 sayısını üretirse de r**akibimiz tarafından darbelendiğimize hükmediyoruz**:

```
if sonuç == 1:
    print('\nSONUÇ: rakibinizi darbelediniz')
    self.darbele(rakip)
```

```
if sonuç == 2:
    print('\nSONUÇ: rakibinizden darbe aldınız')
    self.darbele(self)
```

Saldırı sonucunda rakibimizi darbelediğimizde ve rakibimizden darbe yediğimizde **`darbele()`** adlı bir başka **örnek metodunu** çağırdığımızı da gözden kaçırmayın.

Bu arada, örnek metotlarına da **`self` öneki ile eriştiğimize dikkatinizi çekmek isterim**. Ayrıca her ne kadar örnek metotlarını tanımlarken parantez listesi içinde '**`self`**' kelimesini belirtsek de bu metotları çağırırken **bunları argüman olarak kullanmadığımıza da özellikle dikkat etmelisiniz**. Yani biz bu metotları şöyle tanımlıyoruz:

```
def saldırı_sonucunu_hesapla(self):
     return random.randint(0, 2)
```

Burada parametre listesinde **`self`**'i görüyoruz. Ama bu fonksiyonları çağırırken **parantez içinde** bu **`self`**'i kullanmıyoruz:

```
self.saldırı_sonucunu_hesapla()
```

`self`'i parantez içinde bir **argüman** olarak kullanmak yerine, bu kelimeyi fonksiyon adının başına bir **önek** olarak takıyoruz.

Ne diyorduk? Evet, **`saldır()`** fonksiyonu içinde **`darbele()`** adlı bir fonksiyona atıfta bulunduk. Yani saldırı sonucunda rakibimizi darbelediğimizde ve rakibimizden darbe yediğimizde **`darbele()` adlı bir başka örnek metodunu çağırdık**:

```
def darbele(self, darbelenen):
    darbelenen.darbe += 1
    darbelenen.enerji -= 1
    if (darbelenen.darbe % 5) == 0:
        darbelenen.can -= 1
    if darbelenen.can < 1:
        darbelenen.enerji = 0
        print('Oyunu {} kazandı!'.format(self.isim))
        self.oyundan_çık()
```

Bu fonksiyon içinde, herhangi bir darbe alma durumunda oyuncunun **`darbe`**, **`can`** ve **`enerji`** miktarlarında meydana gelecek değişiklikleri tanımlıyoruz.

Buna göre herhangi bir darbe alma durumunda aşağıdaki işlemler gerçekleştirilecek:

Darbelenen oyuncunun **darbe değeri 1 birim artacak**:

```
darbelenen.darbe += 1
```

`enerji` değeri 1 birim azalacak:

```
darbelenen.enerji -= 1
```

Darbelenen oyuncu **her 5 darbede 1 can kaybedecek**:

```
if (darbelenen.darbe % 5) == 0:
    darbelenen.can -= 1
```

Burada her 5 darbede 1 can kaybetme kriterini nasıl belirlediğimize dikkat edin. Bildiğiniz gibi oyuncu darbe yedikçe darbe değişkeninin değeri artıyor. Bu değer 5 sayısına ulaştığında, **`5 % 5`** işleminin sonucu **`0`** olacaktır. Yani **bu sayı 5'e bölündüğünde bölme işleminden kalan değer 0 olacaktır**. 5'in

tüm katları için (*5, 10, 15, 20 gibi...*) bu durum geçerlidir. Eğer **darbe** değişkenin ulaştığı değer 5'in katı değilse; **bu sayı 5'e tam bölünmediği için bölmeden kalan değer 0 dışında bir sayı olur**. Dolayısıyla darbe değerinin ulaştığı sayının 5'e bölünmesinden kalan değerin **0** olup olmadığını kontrol ederek oyuncunun 5 darbede 1 can kaybetmesini sağlayabiliyoruz.

Oyuncunun can değeri 1'in altına düştüğünde ise **enerji** değeri **0**'a inecek ve **oyunu kimin kazandığı ilan edildikten sonra oyun kapatılacak**:

```
if darbelenen.can < 1:
    darbelenen.enerji = 0
    print('Oyunu {} kazandı!'.format(self.isim))
    self.oyundan_çık()
```

Burada **oyundan_çık()** adlı bir **örnek** metoduna daha atıfta bulunduk:

```
def oyundan_çık(self):
    print('Çıkılıyor...')
    sys.exit()
```

Gayet basit bir fonksiyon. **Herhangi bir şekilde oyundan çıkmak gerektiğinde sys** modülünün **exit() fonksiyonunu kullanarak oyunu terk ediyoruz**.

İlerlemeden önce, **darbele()** fonksiyonunu kullandığımız kısma tekrar bakalım:

```
sonuç = self.saldırı_sonucunu_hesapla()

if sonuç == 0:
    print('\nSONUÇ: kazanan taraf yok')

if sonuç == 1:
    print('\nSONUÇ: rakibinizi darbelediniz')
    self.darbele(rakip)

if sonuç == 2:
    print('\nSONUÇ: rakibinizden darbe aldınız')
    self.darbele(self)
```

Bildiğiniz gibi **darbele()** fonksiyonu, **self** dışında **1 adet parametre daha** alıyor. Bu parametre, **darbeyi hangi oyuncunun alacağını gösteriyor**. *Yani darbeyi alan oyuncu biz miyiz yoksa rakibimiz mi?* İşte bunu tespit etmek için **darbelenen** adlı bir parametre belirledik. Gördüğünüz gibi **darbele()** fonksiyonu **saldır()** adlı başka bir fonksiyonun içinden çağrılıyor. **saldır()** fonksiyonu da **rakip** adlı bir parametre alıyor. İşte darbe alan oyuncunun **can** ve **enerji** değerlerini **yenilemek istediğimizde** bu parametreyi, **darbele()** fonksiyonuna gönderiyoruz:

```
self.darbele(rakip)
```

Burada darbelenen oyuncu karşı taraf. Yani **rakibimiz darbe yemiş**. Eğer darbelenen kişi kendimizsek **kendimize atıfta bulunmak için de self parametresini kullanıyoruz**:

```
self.darbele(self)
```

Pek çok kez söylediğimiz gibi **self** kelimesi **mevcut sınıf örneğini temsil eder**. Dolayısıyla kendimize atıfta bulunmak istediğimiz durumlarda, yukarıda olduğu gibi **self**'i kullanabiliriz.

Eğer arzu ederseniz **darbele()** fonksiyonunu şöyle de yazabilirsiniz:

```
def darbele(self):
    self.darbe += 1
    self.enerji -= 1
    if (self.darbe % 5) == 0:
        self.can -= 1
    if self.can < 1:
        self.enerji = 0
        print('Oyunu {} kazandı!'.format(self.isim))
        self.oyundan_çık()
```

Burada **darbelenen** parametresini **iptal ettik**. Kimin durumunun yenileneceğini **self**'in kim olduğu belirleyecek:

```
if sonuç == 1:
    print('\nSONUÇ: rakibinizi darbelediniz')
    rakip.darbele()
```

```
if sonuç == 2:
    print('\nSONUÇ: rakibinizden darbe aldınız')
    self.darbele()
```

Gördüğünüz gibi eğer rakibi darbeleyip onun **can** ve **enerji durumunu yenilemek istiyorsak** ilgili fonksiyonu **`rakip.darbele()`** şeklinde çağırıyoruz. Kendimizin durumunu **yenilemek istediğimizde ise `self.darbele()` komutunu kullanıyoruz**.

Sınıfımızı tanımladığımıza göre artık bu sınıfı nasıl kullanacağımızı incelemeye geçebiliriz:

```
siz = Oyuncu('Ahmet')
rakip = Oyuncu('Mehmet')
```

Burada öncelikle **`Oyuncu()`** sınıfı için iki farklı **nesne**/**örnek** oluşturuyoruz:

```
siz = Oyuncu('Ahmet')
rakip = Oyuncu('Mehmet')
```

Bu iki nesne, **`Oyuncu()` sınıfının bütün niteliklerini taşıyor**. Nesneleri oluştururken **zorunlu argüman olan isim değerini mutlaka belirtmemiz gerektiğini unutmuyoruz**.

Daha sonra bir **`while`** döngüsü içinde, **oyunumuzun kullanıcı tarafından görüntülenecek kısmını kodluyoruz**:

```
while True:
    print('Şu anda rakibinizle karşı karşıyasınız.',
          'Yapmak istediğiniz hamle: ',
          'Saldır:  s',
          'Kaç:     k',
          'Çık:     q', sep='\n')

    hamle = input('\n> ')
    if hamle == 's':
        siz.saldır(rakip)
```

```
        print('Rakibinizin durumu')
        rakip.mevcut_durumu_görüntüle()

        print('Sizin durumunuz')
        siz.mevcut_durumu_görüntüle()

    if hamle == 'k':
        siz.kaç()

    if hamle == 'q':
        siz.oyundan_çık()
```

Oyunun nasıl oynanacağı konusunda **kullanıcılarımızı bilgilendiriyoruz**:

```
print('Şu anda rakibinizle karşı karşıyasınız.',
      'Yapmak istediğiniz hamle: ',
      'Saldır:  s',
      'Kaç:     k',
      'Çık:     q', sep='\n')
```

Kullanıcılarımızın klavyede **hangi tuşa bastığını şu şekilde alıyoruz**:

```
hamle = input('\n> ')
```

Eğer kullanıcı '**s**' tuşuna basarsa **rakibimize saldırıyoruz**:

```
if hamle == 's':
    siz.saldır(rakip)
```

Saldırının ardından hem **kendi durumumuzu** hem de **rakibimizin durumunu görüntülüyoruz**:

```
print('Rakibinizin durumu')
rakip.mevcut_durumu_görüntüle()

print('Sizin durumunuz')
siz.mevcut_durumu_görüntüle()
```

Eğer kullanıcı '**k**' tuşuna basarsa:

```
if hamle == 'k':
    ...
```

...sınıf içinde tanımladığımız **kaç() metodunu çalıştırıyoruz**:

```
def kaç(self):
    print('Kaçılıyor...')
    for i in range(10):
        time.sleep(.3)
        print('\n', flush=True)

    print('Rakibiniz sizi yakaladı')
```

Burada 300'er milisaniyelik aralıklarla **\n kaçış dizisini kullanarak bir alt satıra geçiyoruz**.

Kullanıcının '**q**' tuşuna basması hâlinde ise **oyundan derhâl çıkıyoruz**:

```
if hamle == 'q':
    siz.oyundan_çık()
```

Bu örnek kodlar bize **sınıflar** ve **nesneler hakkında epey bilgi verdi**. Ayrıca bu kodlar sayesinde önceki bilgilerimizi de pekiştirmiş olduk.

KISIM II

ÖZEL KONULAR

1 MATH MODÜLÜ

İleri düzey modüllerden biri olan **math** modülü, matematikle uğraşanların işlerini bir hayli kolaylaştıracak **metotlar** ve **fonksiyonlar** barındırır. Python programlarımızda matematiksel fonksiyonlardan yararlanabilmek için **math** modülünü kullanıyoruz.

Dilerseniz öncelikle **math** modülünün içeriğini sorgulayalım. Bunun için ilk olarak modülümüzü **içe aktaralım**:

```
>>> import math
```

Bu komut ile modülümüzü çalışmamıza dâhil etmiş olduk. Şimdi, içerdiği fonksiyonları aşağıdaki komutu vererek görelim:

```
>>> dir(math)
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan',
'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp',
'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',
'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh']
```

Modülümüzün içeriğini de gördüğümüze göre şimdi; **kosinüs**, **sinüs**, **tanjant**, **pi**, **karekök**, **üslü ifadeler** gibi fonksiyonlarla ilgili örneklerle kullanımını anlamaya çalışalım.

Üslü İfadeler (pow)

Üslü ifadeler matematikte hep karşımıza çıkmıştır. Python'da bunun için **pow()** fonksiyonu kullanılır. Fonksiyon adında geçen '**pow**' kelimesi İngilizce "**power**" kelimesinin kısaltması olup Türkçede **kuvvet**, **üs** anlamlarına denk gelmektedir. Örnek verecek olursak **23 ifadesinin Python'daki karşılığı**:

```
>>> math.pow(2, 3)
```

şeklindedir. Yukarıdaki kodu yazdığımızda Python bize **cevap** olarak şunu gösterir:

```
8
```

pow() fonksiyonunun kullanımında parantez içerisindeki **ilk sayı tabanı**, **ikinci sayı** ise **üssü** göstermektedir.

Pi sayısı (pi)

Pi sayısı, hepimizin okul yıllarından bildiği üzere alan, hacim hesaplamalarında bolca kullanılır ve değeri genellikle **3** ya da **3**,**14** olarak alınır. Tabii ki esasen **pi sayısı bu kadar kısa değildir**. Bu pi sayısının değerinin kaç olduğunu, Python'a sorarak hemen öğrenelim. Aşağıdaki komutu yazıyoruz:

```
>>> math.pi
```

Dediğimizde Python bize aşağıdaki **çıktıyı** gösterir:

```
>>> 3.1415926535897931
```

Demek ki gerçek pi sayısı biraz daha **uzunmuş**. Şimdi, küçük bir hacim hesaplaması örneği ile konuyu pekiştirelim. **Kürenin hacmini bulalım**. Küre hacmini şu formülle buluyoruz: **Küre hacmi = 4/3(pi.r3)**

Hemen kodlarımızı yazmaya başlayalım:

```
>>> 4 / 3 * math.pi * math.pow(2, 3)
```

Bu kod ile şunu demiş olduk: "**4 ile 3'ü böl, pi sayısı ile çarp, sonra da sonucu 2'nin 3'üncü kuvveti ile çarp**."

Python bize **cevap** olarak şunu gösterdi:

```
33.510321638291124
```

Tabii ki bu işlemleri bir metin düzenleyiciye de yazıp kaydedebiliriz:

```
import math
r = int(input("Kürenin yarıçapını giriniz:"))
hacim = 4 / 3 * math.pi * math.pow(r, 3)
print(hacim)
```

Böylece **kürenin hacmini veren** küçük bir programımız oldu.

Karekök (sqrt)

Karekök ile ilgili fonksiyonumuz **`sqrt()`**. Kullanımı ise aşağıdaki gibidir:

```
>>> math.sqrt(9)
```

kodunu yazıp **ENTER** tuşuna bastığımızda Python bize aşağıdaki **sonucu** gösterir:

```
3
```

Euler Sabiti (e)

Bu fonksiyon matematikteki **`euler`** sabitini veriyor. Kullanımı ise aşağıdaki gibi:

```
>>> math.e
```

Yukarıdaki kodu yazıp **ENTER** tuşuna bastığımızda karşımıza **`euler`** sabiti **`2.718281828459045`** cevap olarak Python tarafından gösteriliyor. Kullanım açısından aynı pi sayısı gibi:

```
2.718281828459045
```

exp() Fonksiyonu

`exp()` fonksiyonunun kullanımı şu şekilde:

```
>>> math.exp(x)
```

Bu kodu küçük bir **örnekle** açıklamak daha kolay olacak:

```
>>> math.exp(2)
```

kodunu yazdığımızda Python aşağıdaki **cevabı** verir:

```
7.38905609893065
```

Bu sayı da nereden çıktı diyorsanız, **`exp()`** fonksiyonu yukarıda bahsettiğimiz **`euler`** sabitinin kuvvetini alan bir fonksiyon. **`exp()`** ifadesindeki **`x`** parametresi **bizim kuvvetimizdir**. Yani **`exp(2)`** dediğimizde esasen biz Python'a şunu demiş oluyoruz: **`(2.718281828459045)2`**, yani **`euler`** sabitinin karesi.

Logaritma (log)

Logaritma ile ilgili fonksiyonumuzun kullanımı şu şekilde: **`log(x, y)`**. Burada **`x`** sayısı **logaritması** alınacak sayı, **`y`** sayısı ise **taban** sayısını temsil etmektedir. Bir örnek ile pekiştirelim:

```
>>> math.log(2, 2)
```

Python bize aşağıdaki cevabı verir:

```
1.0
```

log10() Fonksiyonu

Bu fonksiyonun yukarıdakinden tek farkı **taban sayısının önceden belirlenmiş** ve **`10` olması**. Bu yüzden fonksiyonun kullanımı şöyle: **`log10(x)`**

Burada **`x`, onluk tabana göre logaritması alınacak sayı**dır:

```
>>> math.log10(10)
```

Dönen cevap:

```
1.0
```

degress() Fonksiyonu

`degrees()` fonksiyonu, girilen açıyı **radyandan dereceye çevirmeye yarar**. Kullanımı şu şekilde:

```
>>> math.degrees(x)
```

`x`, burada radyana çevrilecek açımızdır. Örnek olarak **45 derecelik bir açı verelim** ve **radyan karşılığını** bulalım:

```
>>> math.degrees(45)
2578.3100780887044
```

RADIANS() FONKSİYONU

`radians()` fonksiyonu girilen açıyı **dereceden radyana çevirmeye yarar**. **Kullanımı şu şekilde:**

```
>>> math.radians(x)
```

`x` burada dereceye çevrilecek radyandır. Örnek olarak **`45` radyanlık bir açı** verelim ve derece karşılığını bulalım:

```
>>> math.radians(45)
0.78539816339744828
```

Kosinüs, Sinüs ve Tanjant ile ilgili fonksiyonlara girmeden önce belirtmem gereken önemli bir nokta bulunmaktadır. Bu fonksiyonlarda **açı olarak Python**, **radyan** kullanmaktadır. Bu yüzden aldığımız sonuçlar okulda öğrendiğimiz değerlerden farklı olacaktır. Bunu da **`radians()`** fonksiyonu ile düzelteceğiz.

KOSİNÜS (COS)

Hemen **kosinüs fonksiyonu** ile bir **örnek** yapalım:

```
>>> math.cos(0)
```

Python bunun **sonucu** olarak bize:

```
1.0
```

cevabını verir.

```
>>> math.cos(45)
0.5253219888177297
```

Yukarıda gördüğümüz üzere bizim **beklediğimiz cevap bu değil**. Biz **`0.7071`** gibi bir değer bekliyorduk. Bunu aşağıdaki şekilde düzeltebiliriz:

```
>>> math.cos(math.radians(45))
0.7071067811865476
```

Şimdi, istediğimiz cevabı aldık.

Sinüs (sin)

Hemen **sinüs fonksiyonu** ile bir **örnek** yapalım:

```
>>> math.sin(0)
```

Python bunun **sonucu** olarak bize:

```
0.0
```

cevabını verir.

```
>>> math.sin(45)
0.8509035245341184
```

Evet, yukarıda gördüğümüz üzere **bizim beklediğimiz cevap bu değil**. Biz, **`0.7071`** gibi bir değer bekliyorduk.

Bunu aşağıdaki şekilde düzeltebiliriz:

```
>>> math.sin(math.radians(45))
0.7071067811865476
```

Şimdi, istediğimiz **cevabı** aldık.

Tanjant (tan)

Tanjant fonksiyonu ile bir **örnek** yapalım:

```
>>> math.tan(0)
```

Python bunun **sonucu** olarak bize:

```
0.0
```

cevabını verir.

```
>>> math.tan(45)
1.6197751905438615
```

Yukarıda gördüğümüz üzere bizim **beklediğimiz cevap bu değil**. Biz **~1.000** (*yaklaşık*) gibi bir değer bekliyorduk. Bunu aşağıdaki şekilde düzeltebiliriz:

```
>>> math.tan(math.radians(45))
0.99999999999999989
```

Bu da yaklaşık olarak **1.000** yapar. Şimdi, istediğimiz cevabı aldık.

Yukarıda verdiğimiz fonksiyonlardan bazılarını kullanarak basit bir fizik sorusu çözelim.

3 ve 5 Newton büyüklüğünde olan ve aralarındaki açının **60 derece olduğu** iki kuvvetin bileşkesini bulalım.

Boş bir Kate belgesi açarak içine kodlarımızı yazmaya başlayalım:

```
import math

a = 60 # Açımızı a değişkenine atadık.
f1 = 3 # Birinci kuvveti f1 değişkenine atadık.
f2 = 4 # ikinci kuvveti f2 değişkenine atadık.

#Şimdi formülümüzü yazalım:
```

```
R = math.sqrt(math.pow(f1, 2) +
math.pow(f2, 2) + 2 * f1 * f2 *
math.cos(math.radians(a)))

#...ve ekrana çıktı verelim:
print("Bileşke kuvvet:", R)
```

Kodlarımızı çalıştırdığımızda **sonuç**:

```
6.082762530298219
```

Böylece Python'ın standart kütüphanesinde yer alan **math** modülünü de tanımış olduk. Eğer matematikle yakından ilgileniyorsanız bu fonksiyon işlerinizi epey kolaylaştıracaktır.

2

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

Python'da her nesnenin bir **kimliği** (*identity*) vardır. Kabaca söylemek gerekirse kimlik denen şey, esasında o nesnenin **bellekteki adresini temsil eder**. Python'daki her nesnenin kimliği **eşsiz**, **tek** ve **benzersizdir**. *Peki, bir nesnenin kimliğine nasıl ulaşırız?* **Python'da bu işi yapmamızı sağlayacak basit bir fonksiyon bulunur**. Bu fonksiyonun adı **`id()`**. Yani İngilizce'deki **identity** kelimesinin **kısaltması**. Şimdi, örnek bir nesne üzerinde bu **`id()`** fonksiyonunu nasıl kullanacağımıza bakalım.

Bildiğiniz gibi Python'da **her şey bir nesnedir**. Dolayısıyla örnek nesne bulmakta zorlanmayacağız. Herhangi bir şey, oluşturduğumuz anda Python açısından bir **nesneye dönüşecektir zaten**:

```
>>> a = 100
>>> id(a)
137990748
```

Çıktıda gördüğümüz **`137990748`** sayısı **`a`** değişkeninin tuttuğu **`100` sayısının kimliğini gösteriyor**. Şimdi, **`id()`** fonksiyonunu bir de şu şekilde deneyelim:

```
>>> id(100)
137990748
```

Gördüğünüz gibi Python, **a** değişkenini ve **100** sayısını ayrı ayrı sorgulamamıza rağmen **aynı kimlik numaralarını gösterdi**. Bu demek oluyor ki Python iki adet **100** sayısı için **bellekte iki farklı nesne oluşturmuyor**.

İlk kullanımda önbelleğine aldığı sayıyı, ikinci kez ihtiyaç olduğunda **bellekten alıp kullanıyor**. Ama bir de şu örneklere bakalım:

```
>>> a = 1000

>>> id(a)
138406552

>>> id(1000)
137992088
```

Bu defa Python **a** değişkeninin tuttuğu **1000** sayısı ile öteki **1000** sayısı için **farklı kimlik numaraları** gösterdi. Bu demek oluyor ki Python **a** değişkeninin tuttuğu **1000** sayısı için ve doğrudan girdiğimiz **1000** sayısı için bellekte **iki farklı nesne** oluşturuyor. Yani bu iki **1000** sayısı Python açısından birbirinden farklı. Bu durumu görebileceğimiz başka bir yöntem de Python'daki **is** işlecini kullanmaktır. Deneyelim:

```
>>> a is 1000
False
```



Bu komut size **False** çıktısı ile birlikte bir **uyarı mesajı** da gösterecektir. Çünkü gerçek bir programda böyle bir kod yazmak mantıklı değildir. Ancak biz burada Python'ın özel bir yönüne ilişkin sadece inceleme yaptığımız için bu uyarı mesajını görmezden gelebilirsiniz.

Gördüğünüz gibi Python **False** çıktısını suratımıza bir tokat gibi çarptı...

Peki, bu ne anlama geliyor? Şöyle ki: Python'da **is** işlecini kullanarak **iki nesne arasında karşılaştırma yapmak güvenli değildir**. Yani **is** ve **==** işleçleri birbirleriyle aynı işlevi görmez. Bu iki işleç nesnelerin farklı yönlerini sorgular: **is** işleci nesnelerin kimliklerine bakıp o nesnelerin aynı nesneler olup olmadığını kontrol ederken **==** işleci nesnelerin içeriğine bakarak o nesnelerin aynı değere sahip olup olmadıklarını sorgular.

Yani:

```
>>> a is 1000
False
```

Ama...

```
>>> a == 1000
True
```

Burada **is** işleci **a** değişkeninin tuttuğu veri ile **1000** sayısının aynı **kimlik numarasına sahip olup olmadığını sorgularken ==** işleci **a** değişkeninin tuttuğu verinin **1000** olup olmadığını denetliyor. Yani **is** işlecinin yaptığı şey kabaca şu oluyor:

```
>>> id(a) == id(1000)
False
```

Şimdiye kadar denediğimiz örnekler hep sayıydı. Şimdi, isterseniz bir de **karakter dizilerinin** durumuna bakalım:

```
>>> a = "python"
>>> a is "python"
True
```

Burada **True** çıktısını aldık. Bir de **==** işleci ile bir **karşılaştırma** yapalım:

```
>>> a == "python"
True
```

Bu da normal olarak **True** çıktısı veriyor. Ama şu örneğe bakarsak:

```
>>> a = "python güçlü ve kolay bir programlama dilidir"
>>> a is "python güçlü ve kolay bir programlama dilidir"
False
```

Ama...

```
>>> a == "python güçlü ve kolay bir programlama dilidir"
True
```

is ve **==** işleçlerinin nasıl da **farklı sonuçlar** verdiğini görüyorsunuz. Çünkü bunlardan biri nesnelerin kimliğini sorgularken öbürü, **nesnelerin içeriğini sorguluyor**. Ayrıca burada dikkatimizi çekmesi gereken başka bir nokta da "**python**" karakter dizisinin önbelleğe alınıp, gerektiğinde tekrar tekrar kullanılıyorken "**python güçlü ve kolay bir programlama dilidir**" karakter dizisinin ise **önbelleğe alınmıyor** olmasıdır. Aynı karakter dizisinin tekrar kullanılması gerektiğinde Python bunun için bellekte **yeni bir nesne daha** oluşturuyor.

Peki, neden Python örneğin 100 sayısını ve "**python**" karakter dizisini önbelleklerken; **1000** sayısını ve "**python güçlü ve kolay bir programlama dilidir**" karakter dizisini önbelleğe almıyor? **Sebebi şu**: Python kendi iç mekanizmasının işleyişi gereğince ufak nesneleri önbelleğe alırken büyük nesneler için her defasında yeni bir **depolama** işlemi yapıyor. İsterseniz Python açısından **ufak kavramının sınırının ne olabileceğini şöyle bir kod yardımıyla sorgulayabiliriz**:

```
>>> for k in range(1000):
...     for v in range(1000):
...         if k is v:
...             print(k)
```

Bu kod **1000** aralığındaki iki sayı grubunu karşılaştırıp kimlikleri aynı olan sayıları ekrana döküyor... Yani bir bakıma **Python'ın hangi sayıya kadar önbellekleme yaptığını gösteriyor**. Burada aldığımız sonuca göre şöyle bir **denetleme** işlemi yapalım:

```
>>> a = 256
>>> a is 256
True
>>> a = 257
>>> a is 257
False
```

Dediğimiz gibi **`id()`** fonksiyonu ve dolayısıyla **`is`** işleci, nesnelerin kimliklerini denetler. Örneğin **bir listenin kimliğini** şöyle denetleyebiliriz:

```
>>> liste = [“elma”, “armut”, “kebap”]
>>> id(liste)
3082284940
```

Bunu **başka bir liste** üzerinde daha deneyelim:

```
>>> liste2 = [“elma”, “armut”, “kebap”]
>>> id(liste2)
3082284172
```

Gördüğünüz gibi içerik **aynı olduğu hâlde iki listenin kimliği birbirinden farklı**. Python bu iki listeyi bellekte **iki farklı adreste depoluyor**. Ama bu listelerin ait olduğu veri tipi (*yani list*), bellekte tek bir adreste depolanacaktır. Çünkü **bir nesnenin**, **veri tipinin kendisi de başlıbaşına bir nesnedir**. Nasıl **`liste2`** nesnesinin kimlik numarası bu nesne ortalıkta olduğu sürece aynı kalacaksa; bütün listelerin ait olduğu büyük **`list`** veri tipi nesnesi de **tek bir kimlik numarasına** sahip olacaktır:

```
>>> id(type(liste))
>>> 3085544992

>>> id(type(liste2))
3085544992
```

Bu iki çıktı aynıdır, çünkü **`Python list`** veri tipi nesnesi için bellekte **tek bir adres** kullanıyor. Ama ayrı listeler için ayrı adres kullanıyor. Aynı durum tabii ki öteki veri tipleri için de geçerlidir. Mesela **`dict`** (*sözlük*) **veri tipi**:

```
>>> sozluk1 = {}
>>> id(sozluk1)
3082285236

>>> sozluk2 = {}
>>> id(sozluk2)
3082285916
```

Ama tıpkı **list** veri tipinde olduğu gibi **dict veri tipinin nesne kimliği de hep aynı olacaktır**:

```
>>> id(type(sozluk1))
3085549888

>>> id(type(sozluk2))
3085549888
```

Peki, biz bu bilgiden nasıl yararlanabiliriz? Şöyle: **is** işlecini doğrudan iki nesnenin kendisini karşılaştırmak için kullanamasak da **bu nesnelerin veri tipini karşılaştırmak için kullanabiliriz**.

Mesela şöyle bir fonksiyon yazabiliriz:

```
def karşılaştır(a, b):
    if type(a) is type(b):
        print("Bu iki nesne aynı veri tipine sahiptir")
        print("Nesnelerin tipi:", type(a))
    else:
        print("Bu iki nesne aynı veri tipine sahip DEĞİLDİR!")
        print("ilk argümanın tipi:", type(a))
        print("ikinci argümanın tipi:", type(b))
```

Bu fonksiyonu, mesela şöyle kullanabilirsiniz:

```
a = [1, 2, 3]
b = (3, 4, 5)

karşılaştır(a, b)
```

Bu kodları çalıştırdığınızda şöyle bir **çıktı** alacaksınız:

```
Bu iki nesne aynı veri tipine sahip DEĞİLDİR!
ilk argümanın tipi: <class 'list'>
ikinci argümanın tipi: <class 'tuple'>
```

Yukarıdaki **karşılaştır()** fonksiyonu içinde **if type(a) is type(b):** satırı yerine, tabii ki aynı anlamda **if id(type(a)) == id(type(b)):** da yazabilirsiniz. Çünkü **is** işleci, dediğimiz gibi iki nesnenin kimlikleri üzerinden bir **sorgulama işlemi** yapıyor. Dolayısıyla **is** işlecini kullanmak yerine **==** işlecini **id()** fonksiyonuyla **birlikte** kullanmayı da tercih edebilirsiniz.

is işlecini **veri tipi karşılaştırması yapmak için** isterseniz şu şekilde de kullanabilirsiniz:

```
>>> if type(a) is dict:
...     sonuca göre bir işlem...

>>> if type(b) is list:
...     sonuca göre başka bir işlem...

>>> if type(c) is file:
...     sonuca göre daha başka bir işlem...
```

is işlecini aynı zamanda bir nesnenin **None** değerine **eş olup olmadığını kontrol etmek için de** kullanabilirsiniz. Çünkü **None** değeri bellekte her zaman tek bir adreste depolanacak, dolayısıyla bu değere gönderme yapan bütün nesneler için **aynı bellek adresi** kullanılacaktır:

```
>>> if b is None:
... .......
```

gibi...

Sözün özü, **is** işleci iki nesne arasında içerik karşılaştırması yapmak için **güvenli değildir**. Çünkü Python bazı nesneleri (*özellikle ufak boyutlu nesneleri*) önbelleğine alırken **bazı nesneler için her defasında farklı bir depolama işlemi yapmaktadır**. İçerik karşılaştırması için **==** veya **!=** işleçlerini kullanmak daha doğru bir yaklaşım olacaktır.

İKİLİ DOSYALAR

3

Dosyalar genel olarak ikiye ayrılır: Metin dosyaları ve **ikili** (*binary*) **dosyalar**. Eğer bir dosyayı metin düzenleyici ile açtığınızda herhangi bir dilde yazılmış "**okunabilir**" bir metin görüyorsanız o dosya bir **metin dosyası**dır.

Mesela; *Notepad*, *Gedit*, *Kwrite* veya *benzeri metin düzenleyicileri* kullanarak oluşturduğunuz dosyalar birer **metin dosyası**dır. Şimdiye kadar biz hep metin dosyaları üzerinde çalıştık. Ancak metin dosyalarının dışında bir de **ikili dosyalar** bulunur.

İkili dosyalar, metin dosyalarının aksine, **metin düzenleyicilerle açılamayan, açılmaya çalışıldığında ise çoğunlukla anlamsız karakterler içeren** bir dosya türüdür. *Resim dosyaları, müzik dosyaları, video dosyaları, LibreOffice dosyaları, OpenOffice dosyaları*, vb. ikili dosyalara örnektir.

Her ne kadar dosyaları iki türe ayırsak da işin aslı arka planda sadece **tek bir dosya türü vardır**: **İkili dosyalar**. Yani bilgisayarlar açısından bütün dosyalar, içlerinde ne olursa olsun birer **ikili dosyadır** ve içlerinde sadece bit'leri ve bayt'ları (*0'ları ve 1'leri*) barındırır. İşte bu bit'lerin ve bayt'ların ne anlama geleceğini, bu dosyaları okuyan yazılımlar belirler.

Buraya gelene kadar hep metin dosyalarından söz etmiştik. Şimdi ise **ikili dosyalardan** söz edeceğiz.

Hatırlarsanız **metin dosyalarını açmak için temel olarak şöyle bir komut kullanıyorduk**:

```
f = open(dosya_adı, 'r', encoding="utf-8")
```

Bu şekilde bir metin dosyasını okuma kipinde ve "**utf-8**" dil kodlamasıyla açmış oluyoruz. Bir metin dosyasını değil de **ikili bir dosyayı açmak için ise şu komutu kullanacağız**:

```
f = open(dosya_adı, 'rb')
```

Burada dikkatimizi çeken iki nokta var: Birincisi burada herhangi bir **encoding** argümanı görmüyoruz. Çünkü ikili dosyalar açısından '**dil kodlaması**' diye bir kavram söz konusu değildir. Dil kodlamaları sadece **metin dosyaları için geçerlidir**.

İkincisi ise dosyayı açmak için **rb** şeklinde bir **erişim kipi** kullandık.

Python'da **metin dosyaları** ile **ikili dosyalar için kullanılan erişim kipleri birbirinden farklıdır**. Bildiğiniz gibi metin dosyaları için **x**, **w**, **r** veya **a** kiplerinden birini kullanıyoruz. İkili dosyalar için ise **xb**, **wb**, **rb** **ab** kiplerinden birini kullanacağız.

Daha önce dosyalar üzerinde yaptığımız çalışmalardan da gördüğünüz gibi biz bir dosyayı metin dosyası olarak okuduğumuzda bu dosya bize **karakter dizisi tipinde bir veri verecektir**:

```
>>> dosya = open("dosya.txt", "r", encoding="utf-8")
>>> veri = dosya.read()
>>> type(veri)
<class 'str'>
```

İkili dosya olarak okuduğumuz dosyalar ise bize '**bayt**' tipinde bir veri verir:

```
>>> dosya = open("dosya.bin", "rb")
>>> veri = dosya.read()
>>> type(veri)
<class 'bytes'>
```

Bu arada bir İnternet kaynağından okuduğumuz verilerin **her zaman** '**bayt**' **tipinde geldiğini de** bir kenara not edelim:

```
>>> from urllib.request import urlopen
>>> url = "https://tr.wikipedia.org/wiki/Python_(programlama_dili)"
>>> dosya = urlopen(url)
>>> veri = dosya.read()
>>> print(type(veri))
<class 'bytes'>
```

Burada **urllib.request** adlı bir standart kütüphane modülünün **urlopen()** adlı fonksiyonunu kullanarak *"https://tr.wikipedia.org/wiki/Python_(programlama_dili)"* adresindeki **sayfanın içeriğini** okuduk. İşte okuduğumuz bu içeriğin tipi 'karakter dizisi' değil '**bayt**' olacaktır.

İkili Dosyalarla Örnekler

Gelin, isterseniz bu noktada **ikili dosyalara birkaç örnek** verelim.

PDF Dosyalarından Veri Alma

Tıpkı resim, müzik ve video dosyaları gibi PDF dosyaları da birer **ikili dosyadır**. O hâlde hemen önümüze bir **PDF** dosyası alalım ve **bu dosyayı okuma kipinde açalım**:

```
>>> f = open("falanca.pdf", "rb")
```

Şimdi de bu dosyadan **10 baytlık bir veri okuyalım**:

```
>>> f.read(10)

b'%PDF-1.3\n4'
```

Bu çıktıda gördüğünüz '**b**' işareti bu verinin '**bayt**' tipinde olduğunu gösteriyor. Tıpkı **karakter dizileri**, **listeler**, **demetler**, **sözlükler** ve **kümeler** gibi baytlar da Python'daki veri tiplerinden biridir.

Python'da tırnak işaretleri içinde gösterilen ve başında bir 'b' harfi taşıyan veri tiplerine **bayt adı** veriyoruz. Baytlar ve karakter dizileri hemen hemen aynı metotlara sahiptir. Dolayısıyla **karakter dizilerinden bildiğiniz pek çok metodu aynı şekilde baytlar üzerinde de kullanabilirsiniz**.

Gördüğünüz gibi bir PDF dosyasının ilk birkaç baytını okuyarak hem dosyanın bir **PDF belgesi olduğunu teyit edebiliyoruz** hem de bu **PDF belgesinin hangi PDF sürümü ile oluşturulduğunu anlayabiliyoruz**. Buna göre bu belge PDF talimatnamesinin **1.3** numaralı sürümü ile oluşturulmuş.

Eğer biz bu belgeyi bir ikili dosya olarak değil de **bir metin dosyası olarak açmaya çalışırsak Python bize bir hata mesajı** gösterecektir.

Python'ı bu dosyanın bir ikili dosya olduğu konusunda bilgilendirerek **dosyanın düzgün bir şekilde açılıp okunabilmesini sağlıyoruz**.

Gelin, bu PDF belgesi üzerinde biraz daha çalışalım.

PDF belgelerinde, o belge hakkında bazı önemli bilgiler veren **birtakım özel etiketler bulunur**. Bu etiketler şunlardır:

Etiket	Anlamı
`/Creator`	Belgeyi oluşturan yazılım
`/Producer`	Belgeyi PDF'e çeviren yazılım
`/Title`	Belgenin başlığı
`/Author`	Belgenin yazarı
`/Subject`	Belgenin konusu
`/Keywords`	Belgenin anahtar kelimeleri
`/CreationDate`	Belgenin oluşturulma zamanı
`/ModDate`	Belgenin değiştirilme zamanı

Bu etiketlerin tamamı bütün **PDF dosyalarında tanımlı değildir**. Ama özellikle `/Producer` etiketi hemen hemen her PDF dosyasında bulunur.

Şimdi, **örnek olması bakımından elimize bir PDF dosyası alalım ve bunu güzelce okuyalım**:

```
>>> f = open("falanca.pdf", "rb")
>>> okunan = f.read()
```

Şimdi de **`/Producer`** ifadesinin dosya içinde geçtiği noktanın **sıra numarasını** bulalım. Tıpkı karakter dizilerinden olduğu gibi **baytların da `index()` adlı bir metodu vardır**. Bu metot yardımıyla bir ögenin **`bayt`** veri tipi içinde geçtiği noktayı bulabiliyoruz. Yani:

```
>>> producer_index = okunan.index(b"/Producer")
```

Burada **`/Producer`** ifadesinin başına '**`b`**' harfini yerleştirerek bu veriyi bir **`bayt`** olarak tanımlamayı unutmuyoruz. Çünkü şu anda yaptığımız **işlem ikili bir dosya içinde geçen birtakım baytları arama işlemidir**.

`producer_index` değişkeni, '**`/Producer`**' ifadesinin **ilk baytının dosya içindeki konumunu tutuyor**. Kontrol edelim:

```
>>> producer_index

4077883
```

Bu değerin gerçekten de '**`/Producer`**' ifadesinin **ilk baytını depoladığını teyit edelim**:

```
>>> okunan[producer_index]

47
```

Daha önce de dediğimiz gibi **bilgisayarlar yalnızca sayıları görür**. Bu sayının hangi karaktere karşılık geldiğini bulmak için **`chr()`** fonksiyonundan yararlanabilirsiniz:

```
>>> chr(okunan[producer_index])

'/'
```

Gördüğünüz gibi gerçekten de **`producer_index`** değişkeni '**`/Producer`**' ifadesinin **ilk baytının dosya içindeki konumunu gösteriyor**. Biz bu konumu ve bu konumun **`50-60 bayt`** ögesini **sorgularsak PDF belgesini üreten yazılımın adına ulaşabiliriz**. Dikkatlice bakın:

```
>>> okunan[producer_index:producer_index+50]

b'/Producer (Acrobat Distiller 2.0 for Macintosh)\r/T'
```

Hatta eğer bu **çıktı** üzerine **`split()`** metodunu uygularsak çıktıyı daha **kullanışlı bir hâle** getirebiliriz:

```
>>> producer = okunan[producer_index:producer_index+50].split()
>>> producer

[b'/Producer', b'(Acrobat', b'Distiller', b'2.0', b'for',
b'Macintosh)', b'/T']
```

Bu şekilde, **ihtiyacımız olan bilginin istediğimiz parçasına kolayca ulaşabiliriz**:

```
>>> producer[0]
b'/Producer'
>>> producer[1]
b'(Acrobat'
>>> producer[1:3]
[b'(Acrobat', b'Distiller']
```

Elbette bu yöntem, bir PDF dosyasından gerekli etiketleri almanın en iyi yöntemi değildir. Ancak yine de yukarıda, örnek bir ikili dosyadan **nasıl veri alınacağı konusunda** size iyi bir fikir verecektir.

Dilerseniz yukarıdaki işlemleri bir de düzenli ifadelerle nasıl yapabileceğinizi düşünebilirsiniz. Bu sizin için gayet iyi bir beyin jimnastiği olacaktır.

Resim Dosyalarının Türünü Tespit Etme

Dediğimiz gibi *resim dosyaları, müzik dosyaları, video dosyaları* ve *benzeri dosyalar* birer **ikili dosyadır**. Mesela resim dosyalarını ele alalım. Diyelim ki resimlerin hangi türde olduğunu tespit eden bir program yazmak istiyorsunuz. Yani yazdığınız bu programla bir resim dosyasının **PNG** mi, **JPEG** mi, **TIFF** mi, yoksa **BMP** mi olduğunu anlamak istiyorsunuz.

Peki, bir resim dosyasının hangi türde olduğunu bulmak için uzantısına baksanız olmaz mı? Asla unutmayın **dosya uzantıları ile dosya biçimleri arasında doğrudan bir bağlantı yoktur**. O yüzden dosya uzantıları, dosya biçimini anlamak açısından güvenilir bir yöntem değildir. **Bir resim dosyasının sonuna hangi uzantıyı getirirseniz getirin, o dosya bir resim dosyasıdır**. Yani mesela bir resim dosyasının uzantısı yanlışlıkla veya bilerek .docx olarak değiştirilmişse o dosya bir WORD dosyası hâline gelmez. İşte yazacağınız program, bir resim dosyasının uzantısı ne olursa olsun hatta dosyanın bir uzantısı olmasa bile, o dosyanın hangi türde olduğunu söyleyebilecek.

Bir resim dosyasının hangi türde olduğunu anlayabilmek için **ilgili dosyanın ilk birkaç baytını okumanız yeterlidir**. Bu birkaç bayt içinde o resim dosyasının türüne dair bilgileri bulabilirsiniz.

Resim dosyalarının türlerini birbirinden ayırt etmenizi sağlayacak verilerin ne olduğunu, ilgili resim türünün **teknik şartnamesine** bakarak öğrenebilirsiniz. Ancak teknik şartnameler genellikle **okuması zor metinlerdir**. Bu yüzden, doğrudan şartnameyi okumak yerine, Internet üzerinde kısa bir araştırma yaparak konuyu daha kolay anlamanızı sağlayacak yardımcı belgelerden de yardım alabilirsiniz.

JPEG

JPEG biçimi ile ilgili bilgileri *http://www.jpeg.org* adresinde bulabilirsiniz. JPEG dosya biçimini daha iyi anlamanızı sağlayacak **yardımcı kaynak** ise şudur:

1. *http://www.faqs.org/faqs/jpeg-faq/part1/section-15.html*

Yukarıda verdiğimiz adreslerdeki bilgilere göre bir **JPEG dosyasının en başında şu veriler bulunur**:

```
FF  D8  FF  E0  ?   ?   4A  46  49  46  00
```

Ancak eğer ilgili **JPEG** dosyası bir CANON fotograf makinesi ile oluşturulmuşsa bu veri dizisi şöyle de olabilir:

```
FF  D8  FF  E0  ?   ?   45  78  69  66  00
```

Burada soru işareti ile gösterdiğimiz kısım, yani dosyanın **5. ve 6. baytları farklı JPEG dosyalarında birbirinden farklı olabilir**. Dolayısıyla bir JPEG dosyasını başka resim dosyalarından ayırabilmek için **dosyanın ilk dört baytına bakmamız, sonraki iki baytı atlamamız** ve **bunlardan sonra gelen beş baytı kontrol etmemiz yeterli olacaktır**.

Yukarıda gördükleriniz birer **on altılı** (*hex*) **sayıdır**. Bunlar onlu düzende sırasıyla şu sayılara karşılık gelir:

```
255 216 255 224 ? ? 74 70 73 70 0
255 216 255 224 ? ? 69 120 105 102 0 #canon
```

Bu durumu **şu şekilde teyit edebilirsiniz**:

```
jpeg = ["FF", "D8", "FF", "E0", "4A", "46", "49", "46", "00"]
canon = ["FF", "D8", "FF", "E0", "45", "78", "69", "66", "00"]

print("JPEG".center(10))
for i in jpeg:
    print(i, "=>", int(i, 16))

print("\n--------------------------\n")
```

```
print("CANON".center(10))
for i in canon:
    print(i, "=>", int(i, 16))
```

Bu diziler içinde **özellikle şu dört sayı** bizi yakından ilgilendiriyor:

```
74 70 73 70
69 120 105 102 #canon
```

Bu sayılar **ASCII** kodlamasına göre sırasıyla '**J**', '**F**', '**I**', '**F**' ve '**E**', '**x**', '**i**', '**f**' karakterlerine karşılık gelir. Bu durumu da şu kodlarla **teyit edebilirsiniz**:

```
jpeg = [74, 70, 73, 70]
j_canon = [69, 120, 105, 102]

print("JPEG".center(8))
for i in jpeg:
    print(i, "=>", chr(i))

print("\n---------------------------")

print("CANON".center(8))
for i in j_canon:
    print(i, "=>", chr(i))
```

Yani bir **JPEG dosyasını ayırt edebilmek için** ilgili dosyanın **7-10 arası baytlarının ne olduğuna bakmamız yeterli olacaktır**. Eğer bu aralıkta '**JFIF**' veya '**Exif**' karakterleri varsa o dosya bir JPEG dosyasıdır.

Buna göre şöyle bir kod yazabiliriz:

```
f = open(dosya_adı, 'rb')

data = f.read(10)

if data[6:11] in [b"JFIF", b"Exif"]:
    print("Bu dosya JPEG!")

else:
    print("Bu dosya JPEG değil!")
```

Burada, herhangi bir resim dosyasının **ilk on baytını** okuduk öncelikle:

```
data = f.read(10)
```

Çünkü aradığımız bilgiler ilk on bayt içinde yer alıyor.

Daha sonra okuduğumuz kısmın **7 ila 10. baytları arasında kalan verinin ne olduğuna bakıyoruz**:

```
if data[6:11] in [b"JFIF", b"Exif"]:
    ...
```

Eğer ilgili aralıkta **`b'JFIF'`** veya **`b'Exif'`** baytları yer alıyorsa bu dosyanın bir **JPEG dosyası olduğuna karar veriyoruz**.

Yukarıdaki kodları elinizdeki bir JPEG dosyasına uygulayarak *kendi kendinize pratik yapabilirsiniz*.

Mesela benim elimde **`'d1.jpg'`**, **`'d2.jpg'`** ve **`'d3.jpeg'`** adlı **üç farklı JPEG dosyası var**:

```
dosyalar = ["d1.jpg", "d2.jpg", "d3.jpeg"]
```

Bu dosyaların **ilk onar baytını okuyorum**:

```
for f in dosyalar:
    okunan = open(f, 'rb').read(10)
    print(okunan)
```

Buradan şu **çıktıyı** alıyorum:

```
d1.jpg          b'\xff\xd8\xff\xe0\x00\x10JFIF'
d2.jpg          b'\xff\xd8\xff\xe1T\xaaExif'
d3.jpeg         b'\xff\xd8\xff\xe0\x00\x10JFIF'
```

Gördüğünüz gibi bu çıktılar yukarıda **JPEG dosyalarına ilişkin olarak verdiğimiz bayt dizilimi ile uyuşuyor**.

Mesela ilk dosyayı ele alalım:

```
d1.jpg          b'\xff\xd8\xff\xe0\x00\x10JFIF'
```

Burada şu baytlar var:

```
\xff \xd8 \xff \xe0 \x00 \x10 J F I F
```

Sayıların başındaki \x işaretleri bunların birer **on altılı sayı** olduğunu gösteren bir işarettir. Dolayısıyla yukarıdakileri daha net inceleyebilmek için şöyle de yazabiliriz:

```
ff d8 ff e0 00 10 J F I F
```

Şimdi de ikinci dosyanın **çıktısını** ele alalım:

```
d2.jpg          b'\xff\xd8\xff\xe1T\xaaExif'
```

Burada da şu baytlar var:

```
ff d8 ff e1T aa E x i f
```

İşte dosyaların türünü ayırt etmek için bu çıktılardaki **son dört baytı kontrol etmemiz yeterli olacaktır**:

```
for f in dosyalar:
    okunan = open(f, 'rb').read(10)
    if okunan[6:11] in [b'JFIF', b'Exif']:
        print(f"Evet {f} adlı dosya bir JPEG!")
    else:
        print("{f} JPEG değil!")
```

Bu kodları, elinizde bulunan farklı türdeki dosyalara uygulayarak aldığınız çıktıları inceleyebilirsiniz.

PNG

PNG dosya biçiminin, **teknik şartnamesine** *http://www.libpng.org/pub/png/spec/* **adresinden ulaşabilirsiniz**.

Ayrıca yardımcı kaynak olarak da *http://www.fileformat.info/format/png/egff.htm* **adresindeki belgeyi kullanabilirsiniz**.

Şartnamade, *http://www.libpng.org/pub/png/spec/1.2/PNG-Rationale.html#R.PNG-file-signature sayfasındaki* bilgiye göre bir PNG dosyasının ilk 8 baytı mutlaka aşağıdaki değerleri içeriyor:

onlu değer	137 80 78 71 13 10 26 10
on altılı değer	89 50 4e 47 0d 0a 1a 0a
karakter değeri	\211 P N G \r \n \032 \n

Şimdi, elimize herhangi bir PNG dosyası alarak bu durumu **teyit edelim**:

```
>>> f = open("falanca.png", "rb")

>>> okunan = f.read(8)
```

Şartnamede de söylendiği gibi bir PNG dosyasını öteki türlerden ayırt edebilmek için dosyanın **ilk 8 baytına bakmamız yeterli olacaktır**. O yüzden biz de yukarıdaki kodlarda sadece ilk 8 baytı okumakla yetindik.

Bakalım **ilk 8 baytta neler varmış**:

```
>>> okunan

 b"\211PNG\r\n\032\n"
```

Bu değerin, şartnamedeki karakter değeri ile **aynı olup olmadığını sorgulayarak** herhangi bir dosyanın PNG olup olmadığına karar verebilirsiniz:

```
>>> okunan == b"\211PNG\r\n\032\n"

True
```

Dolayısıyla şuna benzer bir kod yazarak **farklı resim dosyalarının türünü tespit edebilirsiniz**:

```
for dosya in dosyalar:
    okunan = open(dosya, 'rb').read(10)

    if okunan[6:11] in [b'JFIF', b'Exif']:
        print(f"{dosya} adlı dosya bir JPEG!")

    elif okunan[:8] == b"\211PNG\r\n\032\n":
        print(f"{dosya} adlı dosya bir PNG!")

    else:
        print(f"Türü bilinmeyen dosya: {dosya}")
```

Bu kodlarda bir resim dosyasının ilk 10 baytını okuduk. 7-11 arası baytların içinde **`b'JFIF'`** veya **`b'Exif'`** baytları varsa **o dosyanın bir JPEG olduğuna**; **ilk 8 bayt `b"\211PNG\r\n\032\n"` adlı bayt dizisine eşitse de o dosyanın bir PNG olduğuna karar veriyoruz**.

GIF

GIF **şartnamesine** *http://www.w3.org/Graphics/GIF/spec-gif89a.txt* **adresinden ulaşabilirsiniz**.

Bir dosyanın GIF olup olmadığına karar verebilmek için **ilk 3 baytını okumanız yeterli olacaktır**. Standart bir GIF dosyasının ilk üç baytı '**`G`**', '**`I`**' ve '**`F`**' karakterlerinden oluşur. Dosyanın sonraki 3 baytı ise GIF'in sürüm numarasını verir. 14/07/2020 itibariyle GIF standardının şu sürümleri bulunmaktadır:

1. 87a - Mayıs 1987
2. 89a - Temmuz 1989

Dolayısıyla standart bir GIF dosyasının **ilk 6 baytı şöyledir**:

```
'GIF87a' veya 'GIF89a'
```

Eğer bir dosyanın GIF olup olmadığını anlamak isterseniz dosyanın **ilk 3 veya 6 baytını denetlemeniz yeterli olacaktır**:

```
for dosya in dosyalar:
    okunan = open(dosya, 'rb').read(10)

    if okunan[6:11] in [b'JFIF', b'Exif']:
        print(f"{dosya} adlı dosya bir JPEG!")

    elif okunan[:8] == b"\211PNG\r\n\032\n":
        print(f"{dosya} adlı dosya bir PNG!")

    elif okunan[:3] == b'GIF':
        print(f"{dosya} adlı dosya bir GIF!")

    else:
        print(f"Türü bilinmeyen dosya: {dosya}")
```

TIFF

TIFF şartnamesine *https://www.adobe.io/content/dam/udp/en/open/standards/tiff/TIFF6.pdf* adresinden ulaşabilirsiniz. Bu şartnameye göre **bir TIFF dosyası şunlardan herhangi biri ile başlar**:

1. 'II'
2. 'MM'

Dolayısıyla, bir TIFF dosyasını tespit edebilmek için dosyanın **ilk 2 baytına bakmanız yeterli olacaktır**:

```
for dosya in dosyalar:
    okunan = open(dosya, 'rb').read(10)

    if okunan[6:11] in [b'JFIF', b'Exif']:
        print(f"{dosya} adlı dosya bir JPEG!")

    elif okunan[:8] == b"\211PNG\r\n\032\n":
        print(f"{dosya} adlı dosya bir PNG!")

    elif okunan[:3] == b'GIF':
```

```
        print(f"{dosya} adlı dosya bir GIF!")

    elif okunan[:2] in [b'II', b'MM']:
        print(f"{dosya} adlı dosya bir TIFF!")

    else:
        print(f"Türü bilinmeyen dosya: {dosya}")
```

BMP

BMP türündeki **resim dosyalarına ilişkin bilgi için** *http://www.digitalpreservation.gov/formats/fdd/fdd000189.shtml* adresine başvurabilirsiniz.

Buna göre **BMP dosyaları BM ile başlar**. Yani:

```
for dosya in dosyalar:
    okunan = open(dosya, 'rb').read(10)

    if okunan[6:11] in [b'JFIF', b'Exif']:
        print(f"{dosya} adlı dosya bir JPEG!")

    elif okunan[:8] == b"\211PNG\r\n\032\n":
        print(f"{dosya} adlı dosya bir PNG!")

    elif okunan[:3] == b'GIF':
        print(f"{dosya} adlı dosya bir GIF!")

    elif okunan[:2] in [b'II', b'MM']:
        print(f"{dosya} adlı dosya bir TIFF!")

    elif okunan[:2] in [b'BM']:
        print(f"{dosya} adlı dosya bir BMP!")

    else:
        print(f"Türü bilinmeyen dosya: {dosya}")
```

Gördüğünüz gibi **ikili dosyalar**, **baytların özel bir şekilde dizildiği ve özel bir şekilde yorumlandığı bir dosya türüdür**. Dolayısıyla ikili dosyalarla çalışabilmek için ikili dosyanın bayt dizilimini yakından tanımak gerekiyor.

4

ÖNEMLİ GÖMÜLÜ FONKSİYONLAR

Bu bölümde hem Python'ın bazı önemli **gömülü fonksiyonlarından** bahsedeceğiz hem de bazı fonksiyonlardaki **Türkçe karakter sorununu** çözmeye çalışacağız.

SORTED()

Bu fonksiyon yardımıyla herhangi bir dizinin ögelerini **alfabe** ve **sayı sırasına** dizebiliyoruz. Çok basit bir **örnek** verelim:

```
>>> k = "istihza"
>>> sorted(k)
['a', 'h', 'i', 'i', 's', 't', 'z']
```

Bu komut, "**istihza**" karakter dizisi içinde geçen harfleri **alfabe sırasına dizdi**.

Eğer arzu ederseniz **sıralamayı ters yönde de** yapabilirsiniz:

```
>>> sorted(k, reverse=True)
['z', 't', 's', 'i', 'i', 'h', 'a']
```

Burada, **reverse** adlı bir isimli **argüman** kullandığımıza dikkat edin. Bu argümanın ön tanımlı değeri **False**'tur. Eğer bunun değerini **True** yaparsak **sorted()** fonksiyonu **sıralama işlemini tersten yapacaktır**.

Bu fonksiyonu, **sayıları sıralamak için de** kullanabilirsiniz.

Gelin, isterseniz önce üzerinde çalışabilmek için **rastgele bir sayı listesi** üretelim. Bunun için **`random`** modülünün **`sample()`** fonksiyonunu kullanabiliriz:

```
>>> import random
>>> samp = random.sample(range(100), 20)
>>> samp
[30, 50, 56, 91, 67, 13, 88, 72, 33, 21, 92, 43, 77, 52, 86, 81, 47,
37, 82,
68]
```

Şu satıra dikkat edin:

```
>>> samp = random.sample(range(100), 20)
```

Burada **`random`** modülünün **`sample()`** fonksiyonunu kullanarak 0 ile 100 aralığında 20 tane **rastgele sayı ürettik**. Gördüğünüz gibi bu sayılar **sıralı değil**. Şimdi, biz bu sayıları **sıraya dizeceğiz**:

```
>>> sorted(samp)
[13, 21, 30, 33, 37, 43, 47, 50, 52, 56, 67, 68, 72, 77, 81, 82, 86,
88, 91,
92]
```

Elbette eğer isterseniz bu sayıları **tersten** de sıralayabilirsiniz:

```
>>> sorted(samp, reverse=True)
[92, 91, 88, 86, 82, 81, 77, 72, 68, 67, 56, 52, 50, 47, 43, 37, 33,
30, 21,
13]
```

Bu bölümün başında **`sorted()`** fonksiyonu yardımıyla herhangi bir diziyi **alfabe** ve **sayı** sırasına dizebileceğimizi söylemiştik. Aslında, bu fonksiyon bunların dışında bize dizileri **uzunluk sırasına dizme imkânı da verir**. Bunun için **`sorted()`** fonksiyonunun **`key`** adlı bir **isimli argümanından** yararlanacağız:

```
>>> isimler = [“ahmet”, “ali”, “fırat”, “can”,
... “abdullah”, “batuhan”, “ayşe”]
>>> sorted(isimler, key=len)
```

```
['ali', 'can', 'ayşe', 'ahmet', 'fırat', 'batuhan', 'abdullah']
```

Gördüğünüz gibi **key** argümanı sıralama işleminin **hangi ölçüte göre** yapılacağını belirliyor. Biz burada, sıralama işleminin **len()** fonksiyonuyla belirlenen **uzunluk ölçütüne göre** yapılmasını istedik.

Diyelim ki elimizde şöyle bir **liste** var:

```
li = ["Ahmet", "can", "mehmet", "Salih", "SEVDA", "Murat"]
```

Biz bu listedeki ögeleri **alfabe sırasına dizmek** istiyoruz:

```
>>> sorted(li)
```

Ancak liste içindeki ögelerin kimisi **küçük**, kimisi **büyük harfli** olduğu için **sorted()** fonksiyonu bunları **düzgün sıralayamıyor**. Böyle bir durumda Python'ın **büyük-küçük harf ayrımı yapmadan sıralama işlemi** gerçekleştirmesi için yine **key** argümanından yararlanabilirsiniz:

```
>>> sorted(li, key=str.lower)
['Ahmet', 'can', 'mehmet', 'Murat', 'Salih', 'SEVDA']
```

Gördüğünüz gibi **str.lower** değeri sayesinde liste içindeki ögelerin hepsinin **küçük harfli olduğu varsayılarak** sıralama işlemi yapıldı. Aynı etkiyi elde etmek için karakter dizilerinin **lower()** metodu yerine **upper()** metodunu da kullanabilirsiniz:

```
>>> sorted(li, key=str.upper)
['Ahmet', 'can', 'mehmet', 'Murat', 'Salih', 'SEVDA']
```

Bu defa da tüm ögeler **büyük harfliymiş gibi** işlem yapıldı. Ama netice itibariyle elde edilen etki aynı olmuş oldu.

Hatırlarsanız önceki derslerimizde listelerin **sort()** metodunu anlatırken bu metodun Türkçe karakterleri **düzgün sıralayamadığını** söylemiştik:

```
>>> isimler = ["ahmet", "ışık", "ismail", "çiğdem", "can", "şule"]
>>> isimler.sort()
>>> isimler
['ahmet', 'can', 'ismail', 'çiğdem', 'ışık', 'şule']
```

Aynı problem, **sorted()** fonksiyonu için de geçerlidir:

```
>>> isimler = ["ahmet", "ışık", "ismail", "çiğdem", "can", "şule"]
>>> sorted(isimler)
['ahmet', 'can', 'ismail', 'çiğdem', 'ışık', 'şule']
```

Gelin şimdi, **bu sorunu nasıl aşabileceğimize** bakalım.

Bu sorunu çözmek için de **key** argümanından yararlanacağız. **Dikkatlice bakın**:

```
def ölçüt(kardiz):
    harfler = "abcçdefgğhıijklmnoöprsştuüvyz"

    tablo = {}
    for harf in harfler:
        tablo[harf] = harfler.index(harf)

    sayısal_değerler = []
    for karakter in kardiz:
        sayısal_değerler.append(tablo.get(karakter))

    return sayısal_değerler

isimler = ["ahmet", "ışık", "ismail", "çiğdem", "can", "şule"]

sıralı = sorted(isimler, key=ölçüt)
print(sıralı)
```

Burada **key** argümanı içinde **değer olarak kullanmak üzere `ölçüt()` adlı bir fonksiyon tanımladık**. Bu fonksiyon, **`sorted()`** fonksiyonunun hangi ölçüte göre sıralama yapacağını belirleyecek:

```
def ölçüt(kardiz):
    harfler = "abcçdefgğhıijklmnoöprsştuüvyz"

    tablo = {}
    for harf in harfler:
        tablo[harf] = harfler.index(harf)

    sayısal_değerler = []
    for karakter in kardiz:
        sayısal_değerler.append(tablo.get(karakter))

    return sayısal_değerler
```

Bu fonksiyon içinde öncelikle **Türk alfabesindeki tüm harfleri gösteren `harfler` adlı bir değişken tanımladık**:

```
harfler = "abcçdefgğhıijklmnoöprsştuüvyz"
```

Daha sonra **`tablo`** adlı **boş bir sözlük** tanımladık:

```
tablo = {}
```

Sözlük veri tipindeki bu tablo içinde Türk alfabesindeki **her bir harfi bir sayı ile eşleştireceğiz**:

```
for harf in harfler:
    tablo[harf] = harfler.index(harf)
```

Bu kodlarla aslında şöyle bir **harf-sayı tablosu** oluşturmuş olduk:

```
{'a': 0,    'b': 1,     'c': 2,
 'ç': 3,    'd': 4,     'e': 5,
 'f': 6,    'g': 7,     'ğ': 8,
 'h': 9,    'ı': 10,    'i': 11,
```

```
 'j': 12,   'k': 13,    'l': 14,
 'm': 15,   'n': 16,    'o': 17,
 'ö': 18,   'p': 19,    'r': 20,
 's': 21,   'ş': 22,    't': 23,
 'u': 24,   'ü': 25,    'v': 26,
 'y': 27,   'z': 28}
```

Gördüğünüz gibi bu sözlükte tüm harflere bir sayı veriyoruz: **`a = 0, b = 1, c = 2`** gibi... Böylece Python, bu sözlüğe bakarak **hangi harfin hangi harften sonra gelmesi gerektiğini anlayacak**.

Kodlarımızın geri kalanı şöyle:

```
sayısal_değerler = []
for karakter in kardiz:
    sayısal_değerler.append(tablo.get(karakter))

return sayısal_değerler
```

Burada öncelikle **`sayısal_değerler`** adlı **boş bir liste** tanımladık:

```
sayısal_değerler = []
```

Daha sonra bir **`for`** döngüsü oluşturarak kardiz içindeki her bir karakterin **harf-sayı tablosu** içindeki sayısal değerini **`sayısal_değerler`** adlı tabloya gönderdik.

Son olarak da **`return`** deyimi yardımıyla fonksiyonumuzdan **`sayısal_değerler`** listesini döndürüyoruz:

```
return sayısal_değerler
```

Bu anlattıklarımız size ilk anda çok *soyut* ve *anlaşılmaz* gelmiş olabilir. Ama eğer yukarıdaki kodları şöyle yazıp çalıştırırsanız bazı şeyler **netleşecektir**:

```
def ölçüt(kardiz):
    harfler = "abcçdefgğhıijklmnoöprsştuüvyz"
```

```
    tablo = {}
    for harf in harfler:
        tablo[harf] = harfler.index(harf)

    sayısal_değerler = []
    for karakter in kardiz:
        sayısal_değerler.append(tablo.get(karakter))

    print(sayısal_değerler)
    return sayısal_değerler

ölçüt("ışık")
```

Gördüğünüz gibi aslında fonksiyonumuzun yaptığı iş çok basit: Kendisine argüman olarak verilen bir karakter dizisi içindeki her bir harfin **sayısal değerini** gösteriyor. Mesela yukarıda bu fonksiyonu "**`ışık`**" karakter dizisi ile çalıştırdık ve şu cevabı aldık:

```
[10, 22, 10, 13]
```

Bu çıktı Python'ın harfleri kolayca sıralayabilmesini sağlayacak. Çünkü farklı dillerin alfabelerindeki karakterleri sıralamanın birbirinden farklı yolları vardır, ama *sayıları sıralamanın tek bir yolu bulunur*. Dolayısıyla **sayıları sıralamak karakterleri sıralamaya kıyasla çok daha kolaydır**.

Şimdi, tanımladığımız bu fonksiyonu **`sorted()`** fonksiyonunun **`key`** argümanına verdiğimizde, **Python hiçbir sorun yaşamadan harfleri sıralayabilecektir**:

```
sıralı = sorted(isimler, key=ölçüt)
print(sıralı)
```

Yukarıda tanımladığımız **`ölçüt()`** fonksiyonu sayesinde, **`sorted()`** fonksiyonu hem tek tek **karakter dizilerini** hem de **liste** gibi veri tipleri içinde geçen karakter dizilerini **sorunsuz** bir şekilde sıralayabilecektir.

MIN()

Bu fonksiyonu bir dizi içindeki **en küçük ögeyi** bulmak için kullanıyoruz.

Örneğin elimizde şöyle bir liste olsun:

```
sayılar = [681, 845, 625, 54, 265, 188, 730, 146, 320, 229, 279,
492, 493,
950, 215, 841, 99, 575, 723, 254, 477, 327, 597, 795, 535, 962, 583,
85,
946, 87, 827, 120, 875, 282, 769, 72, 352, 708, 881, 183, 489, 936,
480,
393, 482, 373, 818, 159, 759, 43, 970, 365, 494, 679, 904, 395, 211,
539,
692, 874, 880, 819, 456, 943, 757, 853, 405, 521, 784, 971, 443,
241, 847,
857, 986, 650, 721, 419, 496, 735, 41, 425, 283, 895, 610, 185, 77,
88, 463,
899, 791, 825, 16, 483, 481, 298, 357, 729]
```

Biz bu liste içindeki **en küçük sayıyı** bulmak istiyoruz. İşte bunun için **`min()`** fonksiyonunu kullanabiliriz:

```
print(min(sayılar))
```

`min()` fonksiyonunu **sayılar dışında karakterler üzerinde de uygulayabilirsiniz**. Diyelim ki elimizde şöyle bir liste var:

```
harfler = ['o', 'l', 'p', 'f', 'v', 'v', 'q', 'e', 'n', 'i', 'g',
'y', 'z',
'b', 'w', 'p', 'j', 'h', 'r', 'l', 'r', 'r', 'd', 't', 'z', 'z',
'p', 'b',
'a', 'd', 'y', 'h', 'j', 'x', 'i', 'o', 'l', 'b', 'p', 'v', 'e',
'f', 'k',
'a', 'm', 'v', 'r', 'n', 'w', 'd', 'o', 'j', 'p', 'c', 'u', 'c',
'e', 'u',
'y', 'l', 'k', 'g', 'h', 'q', 'e', 'd', 'l', 'e', 'q', 'v', 'u',
'm', 'j',
'l', 'a', 's', 'i', 'i', 'c', 'l', 'p', 'k', 'w', 'r', 'g', 'n',
'x', 'n',
'p', 'b', 'm', 'l', 'k', 'k', 'p', 'o', 'm', 'e', 'l', 'z']
```

Eğer **`min()`** fonksiyonunu bu liste üzerine uygularsanız **alfabetik olarak en başta olan harf verilecektir:**

```
print(min(harfler))
```

MAX()

`max()` fonksiyonu, biraz önce öğrendiğimiz **`min()`** fonksiyonunun yaptığı **işin tam tersini yapar:**

```
print(max(sayılar))
```

veya:

```
print(max(harfler))
```

Bu komutlar, sayılar içindeki en büyük sayıyı ve harfler içindeki **alfabetik olarak en sonda yer alan harfi gösterecektir**.

Yalnız hem **`min()`** hem de **`max()`** fonksiyonunu kullanırken dikkate almamız gereken bazı kurallar var:

Sıralamada her zaman **büyük harfler küçük harflerden önce gelir:**

```
>>> z = “abcDEF”
>>> min(z)
‘D’

>>> max(z)
‘c’
```

Eğer sıralamada **büyük-küçük** harfe dikkat edilmemesini isterseniz yukarıda **`sorted()`** fonksiyonunda yaptığımız gibi bu fonksiyonların **`key`** isimli argümanını kullanabilirsiniz:

```
>>> min(z, key=str.lower)
‘a’
>>> max(z, key=str.lower)
```

Bildiğiniz gibi **key** adlı argüman, en büyük veya en küçük değeri belirleme işleminin **hangi ölçüte göre** yapılacağını gösterir.

Bu argümanın değerini **str.lower** yaptığımızda **min()** ve **max()** fonksiyonları bütün harflerin **küçük olduğunu** varsayacaktır. Yani **z** değişkeninin değerini '**abcdef**' olarak alacaktır.

key argümanına **len** değerini vererek sıralamanın **boyuta göre** yapılmasını da sağlayabilirsiniz:

```
>>> l = ["ahmet", "mehmet", "ali", "ayşe", "abdullah"]
>>> min(l, key=len)
'ali'

>>> max(l, key=len)
'abdullah'
```

Tıpkı **sorted()** fonksiyonunda olduğu gibi **min()** ve **max()** fonksiyonlarının da **Türkçe karakterlerle sorunu vardır:**

```
>>> z = "şçöğüıdef"
>>> min(z)
'd'
>>> max(z)
'ş'
>>>
```

Gördüğünüz gibi her iki fonksiyon da **yanlış harfleri gösterdi**. Bu durumu düzeltmek için **sorted()** fonksiyonunda kendi tanımladığımız **ölçüt()** fonksiyonunu kullanabiliriz. **Dikkatlice bakın:**

```
def ölçüt(kardiz):
    harfler = "abcçdefgğhıijklmnoöprsştuüvyz"

    tablo = {}
    for harf in harfler:
        tablo[harf] = harfler.index(harf)
```

```
    sayısal_değerler = []
    for karakter in kardiz:
        sayısal_değerler.append(tablo.get(karakter))

    return sayısal_değerler

z = "şçöğüıdef"

print(min(z, key=ölçüt))
print(max(z, key=ölçüt))
```

Burada da `ölçüt()` fonksiyonu yardımıyla Python'a **Türkçe harfleri nasıl sıralayacağını gösterdiğimiz için** `min()` ve `max()` fonksiyonları işlevlerini **doğru** bir şekilde yerine getirebildi.

SUM()

Python'ın son derece faydalı **hazır/gömülü** fonksiyonlarından biri de `sum()` fonksiyonudur. Bu fonksiyonu kullanarak diziler içindeki sayıların **toplamını** bulabiliyoruz. Mesela **0 ile 20 aralığındaki sayıların toplamını bulalım**:

```
>>> sum(range(20))
190
```

Eğer bu toplam değere bir başka değer daha eklemek istiyorsanız bu fonksiyonu **iki argümanla** çağırabilirsiniz:

```
>>> sum(range(20), 10)
200
```

ENUMERATE()

`enumerate()` fonksiyonu **bir dizi içinde yer alan ögeleri numaralandırmamızı sağlayan** bir fonksiyondur. Elimizde şöyle bir liste olduğunu varsayalım:

```
>>> isimler = ["ahmet", "ışık", "ismail", "çiğdem", "can", "şule"]
```

Biz bu listeyi kullanarak şöyle bir **çıktı** elde etmek istiyoruz:

```
0 ahmet
1 ışık
2 ismail
3 çiğdem
4 can
5 şule
```

Yani liste içindeki her bir ögeyi tek tek numaralandırmak istiyoruz. İşte bunun için **`enumerate()`** fonksiyonundan yararlanabiliriz.

`enumerate()` fonksiyonu, sıra ve öge değerlerinden oluşan **bir dizi demet** verir. *Peki bu ne demek?* **Dikkatlice bakın:**

```
>>> print(*enumerate(isimler))
(0, 'ahmet') (1, 'ışık') (2, 'ismail') (3, 'çiğdem') (4, 'can') (5,
'şule')
```

Burada her biri iki ögeden oluşan birtakım **demetler** görüyoruz. **`enumerate()`** fonksiyonunun böyle bir **çıktı** ürettiğini öğrendikten sonra **artık bu çıktıyı istediğimiz şekilde biçimlendirebiliriz**. Mesela bunları bir **`for`** döngüsü içinde kullanıcıya sunabiliriz:

```
>>> for sıra, öğe in enumerate(isimler):
...     print(sıra, öğe)
...
0 ahmet
1 ışık
2 ismail
3 çiğdem
```

```
4 can
5 şule
```

Eğer **`enumerate()`** fonksiyonunun, ögeleri saymaya **`0`**'dan değil de **`1`**'den başlamasını isterseniz **`enumerate()`** fonksiyonunun **ikinci** argümanını kullanabilirsiniz:

```
>>> for sıra, öğe in enumerate(isimler, 1):
...     print(sıra, öğe)
...
1 ahmet
2 ışık
3 ismail
4 çiğdem
5 can
6 şule
```

ZIP()

Bu fonksiyonu, **farklı dizilerin ögelerini birbiriyle eşleştirmek** için kullanıyoruz:

```
>>> l1 = [1, 2, 3, 4, 5]
>>> l2 = ["bir", "iki", "üç", "dört", "beş"]

>>> print(*zip(l1, l2))
(1, 'bir') (2, 'iki') (3, 'üç') (4, 'dört') (5, 'beş')
```

Gördüğünüz gibi **`zip()`** fonksiyonu da tıpkı **`enumerate()`** gibi **demet veri** tipinde bir **çıktı** veriyor. Bu çıktıyı gördükten sonra artık bunu i**stediğiniz gibi şekillendirebilirsiniz**:

```
>>> for a, b in zip(l1, l2):
...     print(a, b)
...
1 bir
2 iki
3 üç
```

```
4 dört
5 beş
```

Böylece Python'daki en önemli **gömülü/hazır** fonksiyonlardan bazılarını incelemiş olduk. Bunun yanı sıra, ögeleri sıralamak için kullanılan metot ve fonksiyonlardaki **Türkçe karakter sorununun da nasıl üstesinden gelinebileceğini öğrendik**.

KISIM III

ARAYÜZ

GRAFİK ARAYÜZ TASARIMI TEMEL BİLGİLER

1

Bu noktaya kadar Python'la yazdığımız tüm programlar komut arayüzü üzerinde çalışıyordu. Bu bölümde **grafik arayüz üzerinde çalışan programları nasıl yazacağımızı** öğreneceğiz.

Python'da grafik arayüz tasarımı için birden fazla seçeneğimiz var. Bunları bir çırpıda sıralayacak olursak seçeneklerimizin şunlar olduğunu görürüz:

- **PyGI** (*https://pygobject.readthedocs.io/en/latest/*)
- **PyQt** (*https://www.riverbankcomputing.com/static/Docs/PyQt5/*)
- **WxPython** (*https://www.wxpython.org/*)
- **Tkinter** (*https://wiki.python.org/moin/TkInter*)

Arayüz tasarlamak üzere geliştirilmiş yukarıdaki alternatifler içinde biz, Python tarafından resmî olarak desteklenen **Tkinter**'i kullanacağız. *Peki, neden ötekiler değil de Tkinter?*

Öncelikle Tkinter'in öteki seçeneklere karşı en büyük üstünlüğü, **Python programlama dili ile birlikte gelmesidir**. Dolayısıyla Tkinter Python tarafından **resmî olarak desteklenen** bir arayüz tasarım takımıdır. Tkinter hem Windows'ta hem de GNU/Linux üzerinde rahatlıkla çalışabilir. Aslında artık günümüzde öteki arayüz takımları da birden fazla platform üzerinde çalışabilmektedir. Ama Python'la birlikte geldiği için Tkinter, ötekilere göre **çok daha kolay** ve **rahat** bir şekilde kurulup kullanıma hazır hâle getirilebilir.

Eğer Windows kullanıyorsanız Tkinter, zaten Python'la beraber sisteminize kurulmuştur.

Eğer GNU/Linux kullanıyorsanız kullandığınız dağıtımın **paket yöneticisini kullanarak** Tkinter paketini sisteminize kurabilirsiniz. Farklı GNU/Linux dağıtımlarında bu paketin adı farklı olabilir. Paket yöneticiniz içinde '**tk**' şeklinde arama yaparsanız muhtemelen **doğru paketi** bulabilirsiniz. Paket adı '**python3-tk**' veya '**python3.x-tk**' gibi bir şey olacaktır.

Tkinter oldukça **sağlam**, **kararlı** ve **oturmuş** bir arayüz takımıdır. Üstelik kullanımı da oldukça **basittir**. Öteki seçeneklere göre çok daha temiz ve sade bir söz dizimi vardır. Tkinter'in tek eksiği, bu arayüz takımıyla yapılan programların biraz *çirkin* görünmesidir. Ama çeşitli tekniklerden yararlanılarak Tkinter'in görünümünü düzeltmek o kadar da zor bir iş değildir. Ben şahsen, Python programcılarının Tkinter'i kullanmayacak bile olsalar en azından **temelini bilmeleri gerektiğini düşünüyorum**. Tabii ki sizler Tkinter'i öğrendikten sonra öteki seçenekleri de incelemek isteyebilirsiniz.

Dilerseniz lafı daha fazla uzatmadan işe koyulalım.

Pencere Oluşturmak

Grafik arayüz denilen şey tabii ki **penceresiz olmaz**. Dolayısıyla grafik arayüz tasarlamanın ilk adımı **çalışan bir pencere hazırlamak** olacaktır.

Başta söylediğimiz gibi grafik arayüz tasarlarken Tkinter'i kullanacağız. Aslında Tkinter, **Python içinde bir modüldür**. Dolayısıyla başka Python modüllerini nasıl kullanıyorsak `tkinter` modülünü de öyle kullanacağız. Yani öncelikle bu modülü **içe aktaracağız**. Daha önceki yazılardan, Python'da modüllerin nasıl içe aktarıldığını hatırlıyorsunuz.

Şimdi hemen Python'ın **etkileşimli kabuğunda** şu komutu veriyoruz:

```
>>> from tkinter import *
```

Eğer **hiçbir şey olmadan alt satıra geçildiyse sorun yok**. Demek ki sizde `tkinter` modülü kurulu. Ama eğer bu komutu verdiğinizde alt satıra düşmek yerine bir **hata mesajı** alıyorsanız sebebi, gerekli modülün yani tkinter'in sizde **kurulu** olmamasıdır.

Eğer hâl böyleyse yapacağımız işlem çok basit: Kullandığımız GNU/Linux dağıtımının paket yöneticisinden **`tkinter`** modülünü içeren paketi gidip kuracağız. Ubuntu'da bu paketin adı '**python3-tk**'. Sizde de paketin adı buna benzer bir şey olmalı.

Dediğim gibi eğer Windows kullanıyorsanız **`tkinter`** modülü sizde zaten kuruludur. Windows kullanıcılarının **`tkinter`** modülünü kullanabilmesi için ayrı bir şey kurmasına gerek yok.

Modülle ilgili kısmı sağ salim atlattıysanız **artık şu komutu verebilirsiniz**:

```
>>> Tk()
```

Eğer buraya kadar herhangi bir hata yapmadıysak ekrana pırıl pırıl bir arayüz penceresi zıpladığını görmemiz gerekiyor. *Çok güzel, değil mi?*

Burada komutumuzu sadece **`Tk()`** şeklinde kullanabilmemiz, yukarıda **`tkinter`** modülünü **içe aktarma şeklimizden kaynaklanıyor**. Yani **`tkinter`** modülünü **`from tkinter import *`** şeklinde içe aktardığımız için komutu **`Tk()`** şeklinde yazmamız yeterli oluyor.

Eğer **`tkinter`** modülünü **`import tkinter`** şeklinde içe aktarsaydık sadece **`Tk()`** dememiz yeterli olmayacaktı. O zaman **`Tk()`** yerine **`tkinter.Tk()`** dememiz gerekirdi. İsterseniz modülü **`import tkinter`** şeklinde içe aktarıp sadece **`Tk()`** yazdığınızda ne tür bir **hata** aldığınızı kontrol edebilirsiniz. Neyse, konuyu daha fazla dağıtmadan yolumuza devam edelim.

Gördüğünüz gibi **`Tk()`** komutuyla ortaya çıkan pencere, bir pencerenin sahip olması gereken **bütün temel özelliklere sahip**. Yani pencerenin sağ üst köşesinde pencereyi kapatmaya yarayan bir **çarpı işareti**, onun solunda pencereyi büyütüp küçültmemizi sağlayan **karecik** ve en solda da pencereyi **görev çubuğuna indirmemizi sağlayan işaret** bütün işlevselliğiyle karşımızda duruyor. Ayrıca fark edeceğiniz gibi bu pencereyi istediğimiz gibi boyutlandırmamız da mümkün. Bu komutları hangi platformda verdiğinize bağlı olarak **pencerenin görünüşü farklı olacaktır**. Yani bu komutları Windows'ta verdiyseniz Windows'un renk ve şekil şemasına uygun bir pencere oluşacaktır. Eğer **Gnome** kullanıyorsanız pencerenin şekli şemali, Gnome'un renk ve şekil şemasına uygun olarak **KDE**'dekinden **farklı olacaktır**.

Yukarıda verdiğimiz **`Tk()`** komutuyla aslında **`tkinter`** modülü içindeki **`Tk`** adlı sınıfı çağırmış olduk. Bu arada herhangi bir hayal kırıklığı yaşamak istemiyorsanız yukarıdaki **`from tkinter import *`** ve **`Tk()`** komutlarını verirken **büyük-küçük harfe dikkat etmeniz gerekiyor**. Çünkü "**`tkinter`**" ile "**`Tkinter`**" aynı şeyler değildir.

Eğer bir arayüz oluşturacaksak sürekli komut satırında **çalışamayız**. Mutlaka yukarıda verdiğimiz komutları bir yere **kaydetmemiz gerekir**. Bunun için hemen **boş bir belge açıp içine şu satırları ekleyelim**:

```
from tkinter import *
Tk()
```

Eğer bu dosyayı bu şekilde kaydeder ve çalıştırmayı denersek açılmasını beklediğimiz pencere açılmayacaktır. **Burada**, **komut satırından farklı olarak ilave bir satır daha eklememiz gerekiyor**. Yani kodumuzun görünümü şu şekilde olmalı:

```
from tkinter import *
Tk()
mainloop()
```

Belgemizin içine yukarıdaki kodları yazdıktan sonra bunu **.py** uzantılı bir dosya olarak kaydediyoruz ve **normal Python dosyalarını nasıl çalıştırıyorsak öyle çalıştırıyoruz**.

İsterseniz yukarıdaki kodu biraz derleyip toparlayalım. Her ne kadar bu kodlar çalışıyor da olsa **gerçek bir program içinde bu kodlar bu şekilde görünmeyecektir**. Bir defa, **`Tk()`** sınıfını bir değişkene atasak (*yani bu sınıfı 'örneklendirsek' [instantiation]*) hiç fena olmaz:

```
pencere = Tk()
```

Böylece bu sınıfın **fonksiyon** ve **niteliklerine daha kolay erişebiliriz**. Mesela yukarıda tek başına kullandığımız **`mainloop()`** fonksiyonu aslında **`Tk()`** sınıfının metotlarından biridir.

Dolayısıyla bu fonksiyonu şu şekilde belirtmemiz daha **uygun** olacaktır:

```
pencere.mainloop()
```

Yani kodlarımız şöyle görünecek:

```
from tkinter import *

pencere = Tk()
pencere.mainloop()
```

Gördüğünüz gibi bu kodların etkileşimli kabuğa yazdığımız kodlardan temel olarak **tek farkı**; en sonra bir **`pencere.mainloop()`** satırı içeriyor olmasıdır. Bu satır, program içinde bir **döngü oluşturarak** kendisinden önce gelen kodlarla belirlenen özelliklere **göre bir pencere oluşturulmasını sağlıyor**. Bu satırla oluşturulan **döngü sayesinde de** oluşturulan pencere (*aksi belirtilmedikçe veya kullanıcı pencereyi kapatmadıkça*) **ekranda hep açık kalabiliyor**. Eğer bu son satırı yazmazsak penceremiz yine de oluşur, ama biz pencereyi ekranda göremeyiz.

Bu arada buraya küçük bir not düşelim. Eğer yazdığınız programı Python'la birlikte gelen bir metin düzenleyici olan **IDLE içinden çalıştırıyorsanız** yazdığınız programın son satırına **`mainloop()`** ifadesini **yerleştirmeseniz bile programınız hatasız bir şekilde çalışacaktır**. *Neden?* Çünkü IDLE da **`tkinter`** ile yazılmış bir uygulamadır. Dolayısıyla IDLE'nin de kendisine ait bir **`mainloop()`** satırı vardır. Yani siz kendi programınız içine **`mainloop()`**'u **koymasanız bile** IDLE'nin kendi **`mainloop()`**'u sizin **programınızın da çalışmasını sağlayacaktır**. Ancak **`mainloop()`** satırını koymadığınız hâlde programınız IDLE içinden çalışsa da **başka yerde çalışmayacaktır**. O yüzden bu **`mainloop()`** satırını mutlaka yerine koymalıyız...

Dilerseniz kodlarımızın **son hâline** tekrar bir bakalım:

```
from tkinter import *

pencere = Tk()
pencere.mainloop()
```

İlk satırda **`tkinter`** modülünü **programımızın içine aktarıyoruz**. Yani '**`import`**' ediyoruz.

Sonraki satırda, daha önce komut satırında kullandığımız **`Tk()`** sınıfını bir **değişkene atadık**. Nesne Tabanlı Programlama konusundan hatırlayacağınız gibi bir sınıfı değişkene atama işlemine Python'da '**instantiation**', yani '**örnekleme**' adı veriliyordu. İşte biz de burada aslında **`tkinter`** modülü içindeki **`Tk()`** sınıfını örneklemiş olduk.

Son satırdaki **`mainloop()`** ise ana döngüyü başlatarak yazdığımız pencerenin ekranda **sürekli olarak** görünmesini sağlıyor.

Şimdiye kadar Python'da Tkinter yardımıyla **boş bir pencere** oluşturmayı öğrendik. Bu heyecan içinde bu kodları pek çok kez çalıştırıp pek çok defa boş bir pencere oluşmasını izlemiş olabilirsiniz. Ama bir süre sonra boş pencere **sizi sıkmaya başlayacaktır**. O yüzden gelin, isterseniz şimdi de **bu boş pencereyi nasıl dolduracağımızı öğrenelim**. Hemen yeni bir dosya açarak içine şu satırları ekliyoruz:

```
from tkinter import *

pencere = Tk()
```

İlk satırda bildiğiniz gibi **`tkinter`** modülünü çağırdık. Bir alt satırda ise yaptığımız şey **`tkinter`** modülünün **`Tk()`** sınıfına bir ad vermekten, yani daha teknik bir dille ifade etmek gerekirse **`Tk()`** sınıfını **örneklemekten ibaret**. Biz örnekleme (*instantiation*) işlemi için burada '**pencere**' adını tercih ettik. Tabii ki siz isterseniz **başka bir isim de kullanabilirsiniz**.

Şimdi, kodlamaya kaldığımız yerden devam ediyoruz:

```
etiket = Label(text="Elveda Zalim Dünya!")
etiket.pack()

pencere.mainloop()
```

Burada **etiket** adlı yeni bir değişken oluşturduk. Her zamanki gibi bu '**etiket**' adı **zorunlu bir isimlendirme değil**. İstediğiniz ismi kullanmakta serbestsiniz. Ancak değişkenin ismi önemli olmasa da **içeriği önemlidir**.

pencere = Tk() komutu yardımıyla en başta oluşturduğumuz pencereyi bir kutu olarak düşünürsek etiket adıyla tanımladığımız değişken de bu kutu üzerine yerleştirilecek bir **etiket** olarak düşünülebilir. Bahsettiğimiz bu kutunun üzerine etiketini yapıştırmak için **Label()** adlı **özel bir sınıftan faydalanıyoruz**. Daha sonra bu **Label** sınıfına **text** adlı bir parametre vererek etiketin **içeriğinin ne olacağını belirliyoruz**.

Bir alt satırda **etiket.pack()** ifadesini görüyoruz. Bu satır, **bir üstteki satırın görünür hâle gelmesini sağlıyor**. Eğer bu satırı yazmazsak etiketimiz oluşur, ancak biz onu göremeyiz.

En sondaki pencere.mainloop() satırını ise zaten tanıyorsunuz: Yukarıda yazdığımız bütün kodların döngü hâlinde işletilerek bir pencere şeklinde sunulmasını sağlıyor.

Kodlarımızın **son hâli** şöyle olmalı:

```
from tkinter import *

pencere = Tk()

etiket = Label(text="Elveda Zalim Dünya!")
etiket.pack()

pencere.mainloop()
```

Gördüğünüz gibi yine gayet şık, içinde "**Elveda Zalim Dünya!**" yazan bir pencere elde ettik. Alıştırma olsun diye, yukarıda yazdığımız kodlara çok benzeyen başka bir **örnek** daha yapalım:

```
from tkinter import *

pencere = Tk()

etiket_metni = """
Resimler klasörünü silmek
istediğinize emin misiniz?
"""

etiket = Label(text = etiket_metni)
etiket.pack()

mainloop()
```

Bu derste **tkinter** modülünü kullanarak **basit bir pencereyi nasıl oluşturabileceğimizi öğrendik**. Artık bu pencereye ilişkin bazı ayrıntıları incelemeye başlayabiliriz.

Pencere Başlığı

Bildiğiniz gibi gündelik yaşamda karşımıza çıkan pencerelerde genellikle bir **başlık** olur. Mesela bilgisayarınız size bir **hata mesajı** gösterirken pencerenin tepesinde "**Hata**" ya da "**Error**" yazdığını görürsünüz. Eğer istersek biz de oluşturduğumuz **pencerelerin tepesine böyle ifadeler yerleştirebiliriz**. İşte bu işi yapmak, yani penceremizin başlığını değiştirmek için **`Tk()`** sınıfının **`title()`** adlı bir metodundan yararlanmamız gerekiyor. **Yani bunun için şöyle bir kod yazmalıyız**:

```
from tkinter import *

pencere = Tk()

pencere_başlığı = "Hiç bir işe yaramayan \
bir pencereyim ben..."

baslik = pencere.title(pencere_başlığı)

etiket_metni = "...ama en azından bir başlığım var."
etiket= Label(text=etiket_metni)
etiket.pack()

pencere.mainloop()
```

Gördüğünüz gibi **pencereye başlık eklemek son derece kolay**. Tek bir satırla işi hâlledebiliyoruz.

Pencerelerle ilgili daha karmaşık özelliklere geçmeden önce isterseniz burada bir durup şimdiye kadar öğrendiklerimizi **çok kısa bir şekilde özetleyelim**:

`tkinter` modülüyle arayüz tasarımı yapabilmek için öncelikle **`from tkinter import *`** diyerek gerekli modülü **içe aktarmamız** gerekiyor.

Ardından **`Tk()`** sınıfı aracılığıyla boş bir pencere oluşturuyoruz. Kullanım kolaylığı açısından **`Tk()`** ifadesini isimlendirerek **bir değişkene atamayı** (*yani Tk() sınıfını örneklemeyi*) **tercih ediyoruz**.

Eğer penceremizin üzerinde bir metin göstermek istiyorsak bunun için `tkinter` modülünün `Label()` adlı sınıfını kullanacağız. Yani **pencere üzerinde göstermek istediğimiz ifadeyi içeren bir etiket hazırlamamız gerekiyor**. Bu etiketi alıp `Tk()` sınıfı ile oluşturduğumuz pencere üzerine yapıştıracağız. Dediğimiz gibi bu iş için bize `Label()` adlı sınıf yardımcı olacak. Bu sınıfın `text` parametresine vereceğimiz değer, **etiketin içeriğini gösterecek**.

Mesela:

```
oyun = Label(text="Bilgisayarınıza virüs bulaşmış!")
```

Etiketimizi hazırladıktan sonra bu etiketin **görünür hâle gelmesini sağlamamız gerekiyor**. Bunun için `xxx.pack()` gibi bir komut kullanıyoruz. Buradaki '`xxx`' yerine, **bir üst satırda hazırladığımız etiket için kullandığımız değişken adını yazacağız**. Mesela:

```
oyun.pack()
```

Bütün bu yazdığımız kodların **işlevli hâle gelebilmesi için** ise `mainloop()` metodunu kullanarak **ana döngüyü başlatmamız gerekiyor**.

Eğer hazırladığımız pencerenin bir de başlığı olsun istiyorsak `title()` adlı metottan yararlanıyoruz.

Şu ana kadar yapabildiğimiz tek şey, bir pencere oluşturup içine **bir metin eklemek** ve **pencerede basitçe bir başlık oluşturmak**. Ama tabii ki bir süre sonra bu da bizi sıkacaktır. Hem sadece bu özellikleri öğrenerek heyecan verici grafik arayüzler tasarlamamız pek mümkün değil. Bu öğrendiklerimizle en fazla, arkadaşlarımıza ufak tefek şakalar yapabiliriz!

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` modülüyle çalışırken renklerle oynamak oldukça basittir**. Renk işlemleri birtakım ifadeler vasıtasıyla hâllediliyor. **Python dilinde bu ifadelere seçenek** (*option*) **adı veriliyor**. Mesela daha önce öğrendiğimiz ve etiket içine metin yerleştirmemizi sağlayan **`text`** ifadesi de bir seçenektir. Şimdi, öğreneceğimiz seçeneklerin kullanımı da tıpkı bu **`text`** seçeneğinin kullanımına benzer.

fg Seçeneği

Diyelim ki pencere içine yazdığımız bir metni, **daha fazla dikkat çekmesi için kırmızı renkle yazmak istiyoruz**. İşte bu işlem için kullanmamız gereken şey **`fg`** seçeneği. Bu ifade İngilizcedeki **foreground** (*ön plan, ön alan*) kelimesinin **kısaltması** oluyor. Hemen bir **örnek** verelim:

```
from tkinter import *

pencere = Tk()
pencere.title("Hata!")

etiket = Label(text = "Hata: Bilinmeyen Hata 404", fg="red")
etiket.pack()

pencere.mainloop()
```

Gördüğünüz gibi yaptığımız tek şey, **`fg`** seçeneğini **etiketin içine yerleştirmek**. Burada kırmızı renk için kullandığımız kelime, İngilizcede 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**.

Kullanabileceğimiz **İngilizce renk adlarından birkaç tanesini hemen sıralayalım**:

```
red = kırmızı
white = beyaz
black = siyah
yellow = sarı
blue = mavi
brown = kahverengi
green = yeşil
pink = pembe
```

Kullanabileceğimiz renk adları bunlarla sınırlı değil. **Eğer mevcut renk listesini görmek isterseniz, şu adrese bir bakabilirsiniz**: *http://www.tcl.tk/man/tcl8.3/TkCmd/colors.htm*. Tabii o listede sıralanan renkler arasından istediğinizi bulup çıkarmak pek kolay değil. Aynı zamanda renklerin kendisini de gösteren bir liste herhâlde daha kullanışlı olurdu bizim için. Neyse ki çaresiz değiliz. Tkinter bize, renk adlarının yanı sıra **renk kodlarını kullanarak da yazılarımızı renklendirebilme imkânı veriyor**. Renk kodlarını bulmak için şu adresten yararlanabilirsiniz: *http://html-color-codes.info*. Buranın avantajı, **renklerin kendisini de doğrudan görebiliyor olmamız**. Mesela orada beğendiğimiz herhangi bir rengin üzerine tıkladığımızda, **alttaki kutucukta o rengin kodu görünecektir**. Diyelim ki renk paletinden bir renk seçip üzerine tıkladık ve alttaki kutucukta şu kod belirdi: **#610B0B**. Bu kodu kopyalayıp **kendi yazdığımız program içinde gerekli yere yapıştırabiliriz**:

```
from tkinter import *

pencere = Tk()
pencere.title("Hata!")

etiket = Label(text = "Hata: Bilinmeyen Hata 404", fg="#610B0B")
etiket.pack()

pencere.mainloop()
```

Doğrudan renk adları yerine renk kodlarını kullanmak **daha kesin sonuç verecektir**. Ayrıca renk kodları size daha geniş bir renk yelpazesi sunacağı için epey işinize yarayacaktır.

BG Seçeneği

Bu da İngilizcedeki **background** (*arka plan, arka alan*) **kelimesinin kısaltması**. Adından da anlaşılacağı gibi **pencereye yazdığımız metnin arka plan rengini değiştirmeye yarıyor.** Kullanımı şöyle:

```
from tkinter import *

pencere = Tk()
pencere.title("Hata!")

etiket = Label(text = "Hata: Bilinmeyen Hata 404", bg="blue")
etiket.pack()

pencere.mainloop()
```

Yukarıda verdiğimiz renkleri bu seçenek için de kullanabilirsiniz. Ayrıca bu seçenek bir önceki **fg** seçeneğiyle birlikte de kullanılabilir:

```
from tkinter import *

pencere = Tk()
pencere.title("Hata!")

etiket = Label(text = "Hata monitörle sandalyenin tam arasında!",
               fg="white",
               bg="black")
etiket.pack()

pencere.mainloop()
```

Son örnekte renk kodları yerine **renk adlarını** kullandığımıza dikkat edin.

Yazı Tipleri (Fonts)

Tkinter, bize renklerin yanı sıra **yazı tipleriyle de oynama imkânı veriyor**. Hazırladığımız pencerelerdeki yazı tiplerini değiştirmek için tıpkı renklerde olduğu gibi yine bazı seçeneklerden faydalanacağız. Burada kullanacağımız seçeneğin adı **`font`**. **Bunu kullanmak için kodlarımız arasına şuna benzer bir satır 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:

```
from tkinter import *

pencere = Tk()

etiket = Label(text=”Merhaba Dostlar!”, font=(“Times”, 15,
“italic”))
etiket.pack()

pencere.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
```

Örneğin:

```
from tkinter import *

pencere = Tk()

etiket = Label(text="Elveda Zalim Dünya!", font=("Times", 15,
"overstrike"))
etiket.pack()

pencere.mainloop()
```

Yazı tipi olarak neleri kullanabileceğinizi, daha doğrusu kendi sisteminizde hangi yazı tiplerinin **kurulu** olduğunu ise şöyle öğrenebilirsiniz:

```
from tkinter import *
import tkinter.font as font

pencere = Tk()

yazıtipleri = list(font.families())

for yt in yazıtipleri:
    print(yt)

pencere.mainloop()
```

Burada **`tkinter`** modülüyle birlikte bu modülün içindeki ayrı bir modül olan **`font`** adlı modülü de '**`font`**' adıyla **içe aktardığımızı** gözden kaçırmayın. Sistemimizdeki **mevcut yazı tiplerini listelemek için** **`font`** modülünün **`families()`** adlı fonksiyonunu kullandık.

Metin Biçimlendirme

Daha önceki yazılarımızda öğrendiğimiz metin biçimlendirmeye yarayan işaretleri Tkinter'de de kullanabiliyoruz. Yani mesela şöyle bir şey yapabiliyoruz:

```
from tkinter import *

pencere = Tk()

etiket_metni = "Merhaba Dostlar!\n\tNasılsınız?"
yazıtipi = ("Vivaldi", 22, "bold")

etiket = Label(text=etiket_metni, font=yazıtipi, fg="firebrick")
etiket.pack()

pencere.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**:

```
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**: *https://www.tcl.tk/man/tcl8.4/TkCmd/cursors.htm*

Pencere Boyutu

Diyelim ki içinde sadece bir "**etiket**" barındıran bir pencere oluşturduk:

```
from tkinter import *

pencere = Tk()

etiket = Label(text="Hata!")
etiket.pack()

pencere.mainloop()
```

Bu kodları çalıştırdığımızda karşımıza çıkan pencere çok **küçüktür**. O yüzden pencerenin tepesindeki **eksi**, **küçük kare** ve **çarpı işaretleri görünmüyor**. Bunları görebilmek için **pencereyi elle boyutlandırmamız gerekiyor**. Tabii bu her zaman arzu edilecek bir durum değil. Bu gibi durumları engelleyebilmemiz ve penceremizi istediğimiz boyutta oluşturabilmemiz için Tkinter bize **gayet kullanışlı bir araç sunuyor**:

```
from tkinter import *

pencere = Tk()
pencere.geometry("100x100+15+100")

etiket = Label(text="Hata!")
etiket.pack()

pencere.mainloop()
```

Gördüğünüz gibi burada **`pencere.geometry()`** ifadesinin karşısına birtakım sayılar ekleyerek penceremizi boyutlandırıyoruz. İlk iki sayı **`("100x100")`** penceremizin **`100x100`** boyutunda olduğunu, son iki sayı 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 sayıyı çeşitli şekillerde değiştirerek *kendi kendinize denemeler yapmanızı* tavsiye ederim.

Bu arada bu dört sayının aynı anda hepsini birden kullanmak zorunda değilsiniz. İsterseniz **sadece ilk ikisini veya sadece son ikisi** kullanabilirsiniz:

```
from tkinter import *

pencere = Tk()
pencere.geometry("200x200")

etiket = Label(text="Hata!")
etiket.pack()

pencere.mainloop()
```

Burada pencerenin sadece boyunu belirledik. Ama eğer istersek **sadece konumunu da belirleyebiliriz**:

```
from tkinter import *

pencere = Tk()
pencere.geometry("+500+300")

etiket = Label(text="Hata!")
etiket.pack()

pencere.mainloop()
```

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 dâhil. 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:

```
from tkinter import *

pencere = Tk()

pencere.geometry("200x200+100+100")
pencere.resizable(width=FALSE, height=FALSE)

etiket = Label(text="Deneme 1, 2, 3...", cursor="no")
etiket.pack()

pencere.mainloop()
```

Gördüğünüz gibi bu kodlar çalıştırıldığında **ortaya çıkan pencere hiçbir şekilde boyutlandırma kabul etmiyor**.

Böylelikle Grafik Arayüz Tasarımı konusunun **Temel Bilgiler kısmını bitirmiş oluyoruz**. **Bir sonraki bölümde, Tkinter'de Pencere Araçlarını** (*widgets*) **kullanmayı öğreneceğiz**. Ama isterseniz yeni bölüme geçmeden önce şimdiye kadar öğrendiğimiz *hemen hemen bütün konuları kapsayan bir örnek yapalım*.

TEKRAR

Şimdiye kadar Tkinter'in özelliklerine ve nasıl kullanılacağına ilişkin pek çok konu işledik. Yani artık temel olarak Tkinter'in neye benzediğini ve bununla neler yapabileceğimizi az çok biliyoruz. Temel Bilgiler kısmını geride bırakıp yeni bir bölüme geçmeden önce, **şimdiye kadar öğrendiklerimizi bir araya getiren bir örnek yapalım**. Böylece öğrendiklerimizi test etme imkânı bulmuş olacağız. Bu yazıda vereceğim örneği olabildiğince ayrıntılı bir şekilde açıklayacağım.

Şimdi, örneğimize geçelim.

Amacımız **1 ile 100 arasında 6 tane rastgele sayı üretmek**. Yalnız bu 6 sayının her biri **benzersiz olacak**. Yani **6 sayı içinde her sayı tek bir defa geçecek**. Önce kodlarımızın tamamını verelim:

```
from tkinter import *
import random

liste = []

pencere = Tk()
pencere.title("Rastgele Sayılar")

pencere.resizable(width=False, height=False)
pencere.geometry("400x50+600+460")

etiket = Label(fg="white",
               bg="steelblue",
               font=("Helvetica", 22, "bold"))
etiket.pack()

for i in range(6):
    while len(liste) != 6:
        a = random.randint(1, 100)
        if a not in liste:
            liste.append(a)

etiket["text"] = liste

pencere.mainloop()
```

Bu kodları her çalıştırışımızda, **1 ile 100 arasında, birbirinden farklı altı adet rastgele sayı ekranda görünecektir**.

Tkinter'e ilişkin kodlara geçmeden önce, isterseniz yukarıda kullandığımız, rastgele sayı üretmemizi sağlayan Python kodlarını biraz açıklayalım:

Rastgele sayılar üreteceğimiz için öncelikle Python'ın **`random`** adlı modülünü **içe aktarıyoruz**. Bu modülün bu tür **rastgele sayı üretme** işlemlerinde kullanıldığını önceki derslerimizden hatırlıyorsunuz.

`random` modülü içindeki `randint()` metodu bize, örneğin 1'den 100'e kadar rastgele sayılar üretme imkânı sağlıyor. Ancak `random.randint(1, 100)` dediğimiz zaman, **bu kod 1'den 100'e kadar rastgele tek bir sayı üretecektir**. **Bizim amacımız altı adet rastgele sayı üretmek**.

Amacımız altı adet sayı üretmek olduğu için öncelikle `for i in range(6)`: satırı yardımıyla **6 kez işletilecek bir döngü oluşturuyoruz**.

Bu döngü nihai olarak `random.randint()` fonksiyonunu **altı kez çalıştıracaktır**. *Böylelikle elimizde altı adet sayı olmuş olacak*.

Ancak üretilen bu altı sayının hepsi birbirinden **farklı olmayabilir**. Yani bazı sayılar **birden fazla üretilebilir**. Dolayısıyla "**1, 5, 5, 7, 56, 64**" gibi bir diziyle baş başa kalabiliriz. Ama bizim **amacımız altı sayının hepsinin birbirinden farklı olması**. O yüzden bunu sağlayacak bir **önlem almamız gerekiyor**. Bu bahsettiğimiz önlemi `if a not in liste: liste.append(a)` satırlarıyla alıyoruz. Aradaki `while` döngüsü başka bir işe yarıyor. Onu birazdan açıklayacağız.

`if a not in liste: liste.append(a)` satırlarıyla, "Eğer `random.randint(1, 100)` komutu ile üretilen sayı liste içinde yer almıyorsa bu sayıyı listeye ekle!" emrini veriyoruz. Yani `random.randint()` **ile her sayı üretilişinde Python'ın bu sayıyı listeye eklemeden önce, listede bu sayının zaten var olup olmadığını kontrol etmesini istiyoruz**.

Eğer üretilen sayı listede yoksa `liste.append()` komutu sayesinde sayımız listeye eklenecek, yok **eğer üretilen sayı listede zaten varsa o sayı listeye alınmayacaktır**. **Ancak burada bir sorun çıkıyor karşımıza**. Eğer üretilen sayı listede zaten varsa **yukarıda yazdığımız komut gereği o sayı listeye alınmayacak**, bu durumda **listede 6'dan az sayı kalacaktır**. Bizim, üretilen sayıların aynı olması durumunda `random.randint()` komutunun altıyı tamamlayana kadar **tekrar tekrar sayı üretmesini sağlamamız gerekiyor**.

İşte bu işlemi yukarıdaki `while` döngüsü yardımıyla yapacağız. `while len(liste) != 6`: satırı ile liste uzunluğunun **6'ya eşit olup olup olmadığını denetliyoruz**. Yani Python'a şu emri veriyoruz. "liste'nin uzunluğu 6'ya eşit olmadığı sürece `random.randint(1,100)` komutunu işletmeye devam et!".

Böylelikle listedeki tekrar eden sayılar yüzünden liste içeriği 6'dan küçük olduğunda, bu **`while`** döngüsü sayesinde Python listeyi altıya **tamamlayana kadar `random.randint()`** komutunu çalıştırmaya devam edecektir. **`if a not in liste`**: satırı nedeniyle zaten listede var olan sayıları listeye ekleyemeyeceği için, benzersiz bir sayı bulana kadar uğraşacak ve neticede bize 6 adet, **birbirinden farklı sayı verecektir**.

Kodların Tkinter ile ilgili kısmına gelelim. Esasında yukarıdaki kodlar içinde geçen her satırı anlayabilecek durumdayız. Şimdiye kadar görmediğimiz tek biçim, **`etiket["text"] =`** liste satırı. Bu biçimle ilgili olarak şunu söyleyebiliriz: Tkinter'deki etiket, düğme vb. ögelerin niteliklerini, yukarıda görüldüğü şekilde, tıpkı bir sözlüğün ögelerini **değiştirirmiş gibi değiştirebiliriz**. Yukarıdaki yapının, **sözlük ögelerini değiştirme konusunu işlerken gördüğümüz söz diziminin aynısı olduğuna dikkat edin**. Mesela bu yapıyı kullanarak şöyle bir şey yapabiliriz. **Diyelim ki programımız içinde aşağıdaki gibi bir etiket tanımladık**:

```
etiket = Label(fg="white",
               bg="steelblue",
               font=("Helvetica", 22, "bold"))
etiket.pack()
```

Etiketin **`text`** seçeneğinin bulunmadığına dikkat edin. İstersek yukarıdaki satırda **`text`** seçeneğini **`text = ""`** şeklinde **boş olarak da tanımlayabiliriz**. Bu etiketi bu şekilde tanımladıktan sonra, programımız içinde başka bir yerde bu etiketin **`text`** seçeneğini, **öteki özellikleriyle birlikte tek tek değiştirebiliriz**. Şöyle ki:

```
etiket["text"] = "www.istihza.com"
etiket["fg"] = "red"
etiket["bg"] = "black"
etiket["font"] = ("Verdana", 14, "italic")
```

Bu yapının, **Python'daki sözlüklere ne kadar benzediğine bir kez daha dikkatinizi çekmek isterim**. Bu yüzden bu yapıyı öğrenmek sizin için zor olmasa gerek.

Artık örneğimizi verip açıklamalarımızı da yaptığımıza göre, *gönül rahatlığıyla bir sonraki bölüme geçebiliriz*.

PENCERE ARAÇLARI (1. BÖLÜM)

2

Arayüz uygulamaları tabii ki **sadece pencerelerden ibaret değildir**. Arayüzler üzerinde, istediğimiz işlemleri yapabilmemiz için yerleştirilmiş birtakım *menüler, yazılar, düğmeler, kutucuklar* ve *buna benzer araçlar da* bulunur. İşte herhangi bir programın arayüzü üzerinde bulunan düğmeler, etiketler, sağa-sola, yukarı-aşağı kayan çubuklar, kutular, kutucuklar, menüler, vb. **hepsi birden pencere araçlarını oluşturur**.

Aslında şimdiye kadar bu pencere araçlarından bir tanesini gördük. Bildiğimiz bir pencere aracı olarak elimizde şimdilik sadece `Label` bulunuyor. Gelin, bu `Label` **adlı pencere aracına biraz daha yakından bakalım**:

Label Pencere Aracı

Daha önce de söylediğimiz gibi '`label`' kelimesi İngilizcede '**etiket**' anlamına geliyor. Bu araç, anlamına uygun olarak pencerelerin üzerine etiket misali ögeler yerleştirmemizi sağlıyor.

Hatırlayacağınız gibi bu aracı `Label()` şeklinde kullanıyorduk. İsterseniz bununla ilgili olarak hemen **basit bir örnek** verelim:

```
from tkinter import *

pencere = Tk()
pencere.geometry("200x200")
```

```
etiket = Label(text = "Hata: Bellek Read Olamadı!")
etiket.pack()

pencere.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:

```
from tkinter import *

pencere = Tk()
pencere.geometry("200x200")

düğme = Button(text="TAMAM", command = pencere.quit)
düğme.pack()

pencere.mainloop()
```

Dikkat ederseniz **`Button()`** aracının kullanımı, daha önce gördüğümüz **`Label()`** aracının kullanımına **çok benziyor**.

Burada da **parantez içinde bazı parametreler kullandık**. `text` parametresini zaten biliyoruz: Kullanıcıya göstermek istediğimiz metni bu `text` parametresi yardımıyla belirliyoruz. Aynı parantez içinde gördüğümüz **`command`** parametresi ise düğme üzerine tıklandığında **işletilecek komutu** gösteriyor. Biz burada **`pencere.quit()` komutunu vererek düğmeye tıklandığında pencerenin kapatılmasını istedik**. **Dolayısıyla bir satır içinde üç yeni özellik görmüş oluyoruz**:

- **Button:** Kullanacağımız pencere aracı.
- **command:** Oluşturduğumuz düğmeye tıklandığında çalıştırılacak komut.
- **xxx.quit:** Düğmeye tıklandığında pencerenin kapatılmasını sağlayan komut.

Daha sonra gelen satırdaki ifade size tanıdık geliyor olmalı: **`düğme.pack()`**. Tıpkı daha önce **`etiket.pack()`** ifadesinde gördüğümüz gibi bu ifade de **hazırladığımız pencere aracının görünür hâle gelmesini sağlıyor**.

En son satırdaki **`pencere.mainloop()`** ifadesinin ne işe yaradığını ise artık söylemeye bile gerek yok.

Yukarıda bahsettiğimiz **`command`** parametresi, çok faydalı işler yapmanızı sağlayabilir. Mesela diyelim ki "**Oluştur**" düğmesine basınca, bilgisayarda yeni bir dosya oluşturan bir arayüz tasarlamak istiyoruz. **O hâlde hemen bu düşüncemizi tatbik sahasına koyalım**:

```
from tkinter import *

pencere = Tk()
pencere.geometry("300x200")

def oluştur():
    dosya = open("deneme.txt", "w", encoding="utf-8")
    dosya.close()

    etiket["text"] = "dosya oluşturuldu"
    etiket["font"] = ("Verdana", 15)

düğme = Button(text="oluştur", command=oluştur)
düğme.pack()

etiket = Label()
etiket.pack()

pencere.mainloop()
```

Gördüğünüz gibi **`tkinter`** modülünü çağırıp **`pencere = Tk()`** şeklinde penceremizi oluşturup **`geometry()`** metoduyla pencere boyutunu ayarladıktan sonra bir fonksiyon yazdık.

Tkinter dışından bir komut çalıştırmak istediğimizde **bu şekilde bir fonksiyon tanımlamamız gerekir**.

Bu fonksiyon içinde bir **dosya açtık** ve **kapattık**:

```
dosya = open("deneme.txt", "w", encoding="utf-8")
dosya.close()
```

Daha sonra dosyanın oluşturulduğunu **kullanıcılarımıza haber vermek amacıyla** **`Label()`** aracının **`text`** ve **`font`** parametrelerini kullandık:

```
etiket["text"] = "dosya oluşturuldu"
etiket["font"] = ("Verdana", 15)
```

Fonksiyonumuzu tanımladıktan sonra **`Button()`** pencere aracı yardımıyla, pencereye yerleştireceğimiz düğmeyi meydana getirdik.

Burada **`command`** parametresine, biraz önce oluşturduğumuz fonksiyonu atayarak düğmeye basıldığında **yeni bir dosya oluşturulmasına zemin hazırladık**. **`text`** parametresi yardımıyla da düğmemizin adını "**Oluştur**" olarak belirledik.

Son olarak da kullanıcılarımıza dosyanın oluşturulduğunu haber verecek olan etiketimizi meydana getirdik:

```
etiket = Label()
etiket.pack()
```

`Label()` pencere aracının **`text`** ve **`font`** seçeneklerini **`oluştur()`** fonksiyonu içinde değiştirdiğimizi hatırlıyorsunuz.

Eğer bu kodları yazarken yukarıdaki gibi bir fonksiyon oluşturmadan **`düğme = Button(text = "oluştur", command=open("deneme.txt", "w"))`** gibi bir satır yazarsanız '**`deneme.txt`**' **adlı dosya düğmeye henüz basmadan oluşacaktır**.

İsterseniz bu arayüze bir de "**Çıkış**" butonu ekleyebiliriz:

```
from tkinter import *

pencere = Tk()
pencere.geometry("300x200")
```

```
def oluştur():
    dosya = open("deneme.txt", "w", encoding="utf-8")
    dosya.close()

    etiket["text"] = "dosya oluşturuldu"
    etiket["font"] = ("Verdana", 15)

düğme = Button(text = "oluştur", command=oluştur)
düğme.pack()

düğme2 = Button(text = "çıkış", command=pencere.quit)
düğme2.pack()

etiket = Label()
etiket.pack()

pencere.mainloop()
```

Burada yaptığımız şey, **ikinci bir düğme oluşturmaktan ibaret**. Oluşturduğumuz **`düğme2`** adlı değişken için de **`düğme2.pack()`** ifadesini kullanmayı unutmuyoruz.

Bu arada dikkat ettiyseniz yukarıdaki arayüz üzerine yerleştirdiğimiz düğmeler biraz çirkin görünüyor. *Ama eğer isterseniz bu düğmeleri biraz daha cazip bir görünüme kavuşturabilirsiniz*. **Şimdi şu kodlara dikkatlice bakın**:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()
pencere.geometry("300x200")

def oluştur():
    dosya = open("deneme.txt", "w", encoding="utf-8")
    dosya.close()

    etiket["text"] = "dosya oluşturuldu"
    etiket["font"] = ("Verdana", 15)
```

```
düğme = ttk.Button(text = "oluştur", command=oluştur)
düğme.pack()

düğme2 = ttk.Button(text = "çıkış", command=pencere.quit)
düğme2.pack()

etiket = Label()
etiket.pack()

pencere.mainloop()
```

Burada, **ilk olarak şu eklemeyi yaptık**:

```
import tkinter.ttk as ttk
```

Bu kodla **tkinter** içindeki **ttk** adlı ayrı bir modülü **ttk** adıyla programımıza dâhil ettik.

ttk modülü içindeki pencere araçları, **tkinter** modülü içindeki pencere araçlarına göre daha **zariftir**. O yüzden programlarınızda **tkinter** modülünün pencere araçları yerine, **ttk** modülünün pencere araçlarını kullanmayı **tercih edebilirsiniz**.

Mesela biz, yukarıdaki kodlarda **tkinter** modülünün düğmeleri yerine **ttk modülünün düğmelerini kullandık**:

```
düğme = ttk.Button(text = "oluştur", command=oluştur)
düğme.pack()

düğme2 = ttk.Button(text = "çıkış", command=pencere.quit)
düğme2.pack()
```

Buradaki **ttk.Button()** ifadelerine dikkat edin. Bunlar **tkinter** modülünün değil, **ttk modülünün düğmeleri**. Gördüğünüz gibi görünüş olarak bunlar biraz daha şık.

İsterseniz etikette de **tkinter** yerine **ttk** modülünü **tercih edebilirsiniz**:

```
etiket = ttk.Label()
etiket.pack()
```

Ama görünüş olarak iki modülün bize sunduğu **Label()** pencere araçları arasında bir fark olmadığı için bunları **olduğu gibi de bırakabilirsiniz**.

Bu arada eğer **tkinter** yerine **ttk** modülünün pencere araçlarını kullanacaksanız bu ikisi arasındaki **bazı uyumsuzlukları da göz önünde bulundurmanız gerekiyor**.

Örneğin **tkinter** modülünün aksine, **ttk** modülü içindeki pencere araçlarının **fg** ve **bg** seçenekleri bulunmaz. Eğer bir **ttk** pencere aracının **ön plan veya arka plan rengini değiştirecekseniz farklı bir yol izlemeniz gerekir**. Bu durumu bir **örnek** üzerinde gösterelim:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()
pencere.geometry("300x100")

stil = ttk.Style()
stil.configure("TLabel",
               foreground="white",
               background="firebrick",
               font=("arial", 22, "bold"))

etiket = ttk.Label(text="Bu bir ttk etiketidir!")
etiket.pack()

pencere.mainloop()
```

Burada etiketin ön plan ve arka plan renkleri ile birlikte **yazı tipini** **ttk modülünün Style() adlı bir sınıfını kullanarak belirledik**:

```
stil = ttk.Style()
stil.configure("TLabel",
```

```
                foreground="white",
                background="firebrick",
                font=("arial", 22, "bold"))
```

Gördüğünüz gibi **`ttk.Style()`** sınıfını **`stil`** adıyla örneklendirdikten sonra **`configure`** adlı bir metodu kullanarak şöyle bir satır yazdık:

```
stil.configure("TLabel",
                foreground="white",
                background="firebrick",
                font=("arial", 22, "bold"))
```

Burada ilk argüman olan "**`TLabel`**", **ön plan ve arka plan renklerini değiştireceğimiz pencere aracının adını gösteriyor**. Eğer hedefimizdeki pencere aracı **`Label`** ise burada "**`TLabel`**" ifadesini kullanacağız. Eğer hedefimizdeki pencere aracı **`Button`** ise o zaman da "**`TButton`**" ifadesini kullanacağız. **Yani çoğu durumda** (*her zaman olmamakla birlikte*) **pencere aracının adı neyse bu adın sol tarafına bir "T" harfi ekleyeceğiz**.

Pencere aracının adını belirttikten sonra **`foreground`** ve **`background`** parametrelerini kullanarak sırasıyla **ön plan** ve **arka plan** renklerini belirliyoruz.

Son olarak da **`font`** parametresi yardımıyla **yazı tipini** ayarlıyoruz.

Stilimizi bu şekilde tanımladıktan sonra artık oluşturduğumuz tüm etiketler **bu stile sahip olacaktır**:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()

stil = ttk.Style()
stil.configure("TLabel",
                foreground="white",
                background="firebrick")

etiket1 = ttk.Label(text="etiket1")
etiket1.pack()
```

```
buton = ttk.Button(text="buton1")
buton.pack()

etiket2 = ttk.Label(text="etiket2")
etiket2.pack()

pencere.mainloop()
```

Eğer siz bu stilin tüm etiketlere değil de **sadece belli bir etikete uygulanmasını istiyorsanız o zaman kodlarınızı şöyle yazacaksınız**:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()

stil = ttk.Style()
stil.configure("özel.TLabel",
               foreground="white",
               background="firebrick")

etiket1 = ttk.Label(text="etiket1")
etiket1.pack()

buton = ttk.Button(text="buton1")
buton.pack()

etiket2 = ttk.Label(text="etiket2", style="özel.TLabel")
etiket2.pack()

pencere.mainloop()
```

Burada, **özellikle şu kodlara dikkat ediyoruz**:

```
stil = ttk.Style()
stil.configure("özel.TLabel",
               foreground="white",
               background="firebrick")
```

ve:

```
etiket2 = ttk.Label(text="etiket2", style="özel.TLabel")
etiket2.pack()
```

Burada öncelikle **etiket adını belirlerken şu adı kullandık**:

```
"özel.TLabel"
```

Böylece tüm "**`TLabel`**" pencere araçlarını değil, **yalnızca** "**`özel.TLabel`**" **işareti taşıyan `Label` pencere araçlarını hedeflemiş olduk**.

Daha sonra da bu stili hangi etiket üzerine uygulamak istiyorsak stil adını o etiketin **`style`** parametresine yazdık:

```
etiket2 = ttk.Label(text="etiket2", style="özel.TLabel")
```

Böylece bu değişiklikten sadece **`etiket2`** adlı **pencere aracı etkilenmiş oldu**.

Elbette eğer **`ttk`** modülünün **`Label()`** aracını değil de **`tkinter`** modülünün **`Label()`** aracını kullanacaksak hiç **`Style()`** sınıfıyla uğraşmadan **`fg`** ve **`bg`** seçeneklerini kullanmaya devam edebiliriz:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()

etiket1 = ttk.Label(text="etiket1")
etiket1.pack()
```

```
buton = ttk.Button(text="buton1")
buton.pack()

etiket2 = Label(text="etiket2", fg="white", bg="firebrick")
etiket2.pack()

pencere.mainloop()
```

ttk modülünün **Style()** sınıfını da kısaca incelediğimize göre yolumuza devam edebiliriz.

Hatırlarsanız önceki derslerimizden birinde *1-100 arası sayılar içinden rastgele altı farklı sayı üreten bir uygulama yazmıştık*. Artık **Button()** adlı pencere aracını öğrendiğimize göre o uygulamayı birazcık daha geliştirebiliriz. Mesela o uygulamaya bir **düğme ekleyebilir**, **o düğmeye basıldıkça yeni sayıların üretilmesini sağlayabiliriz**:

```
from tkinter import *
import tkinter.ttk as ttk
import random

pencere = Tk()
pencere.geometry("300x100")

def kodlar():
    liste = []
    for i in range(6):
        while len(liste) != 6:
            a = random.randint(1, 100)
            if a not in liste:
                liste.append(a)

    etiket["text"] = liste

etiket = Label(text="Sayı üretmek için düğmeye basın",
               fg="white",
               bg="#61380B",
               font=("Helvetica", 12, "bold"))
etiket.pack()
```

```
düğme = ttk.Button(text="Üret", command=kodlar)
düğme.pack()

pencere.mainloop()
```

Burada, programda ana işlevi gören kodları **`kodlar()`** adlı bir fonksiyon içine aldık. Çünkü **`Button()`** pencere aracı içindeki **`command`** seçeneğine atayabilmemiz için elimizde bir **fonksiyon** olması gerekiyor.

Düğmeye bastığımızda **`ttk`** modülü, fonksiyon içindeki bu kodları işletecek. Düğmeye her basıldığında liste ögelerinin yenilenebilmesi için **`liste=[]`** **satırını fonksiyon içine alıyoruz**. Böylece düğmeye ilk kez basıldığında önce liste ögesinin **içi boşaltılacak**, **ardından altı adet sayı üretilip etikete yazdırılacaktır**. Böylelikle düğmeye her basışımız yeni bir sayı dizisi oluşturacaktır.

Eğer **`liste=[]`** satırını fonksiyon dışında tanımlarsak *düğmeye ilk basışımızda* **altı adet sayı üretilecektir**, *ama ikinci kez düğmeye bastığımızda*, **liste hâlâ eski ögeleri tutuyor olacağı için etiket yenilenmeyecek**, **düğmeye her basışımızda eski sayılar etikete yazdırılmaya devam edecektir**. Bizim istediğimiz şey ise düğmeye her basışta yeni bir sayı setinin etikete yazdırılması. Bunun için **`liste=[]`** satırını fonksiyon içine almamız gerekiyor.

Programımızın işlevini yerine getirebilmesi için kullanıcının bir düğmeye basması gerekiyor. Bu yüzden kullanıcıya ne yapması gerektiğini söylesek iyi olur. Yani kullanıcımızı düğmeye basmaya yönlendirmemiz lazım. Bu amaçla etiketin **`text`** seçeneğine "**Sayı üretmek için düğmeye basın**" şeklinde bir uyarı yerleştirdik. Programımız ilk kez çalıştırıldığında kullanıcı öncelikle bu uyarıyı görecek, böylelikle programı kullanabilmek için ne yapması gerektiğini bilecektir. Belki bu küçük programda çok belirgin olmayabilir, ama büyük programlarda, etiketler üzerine yazdığımız yönergelerin **açık** ve **anlaşılır** olması çok önemlidir. Aksi hâlde kullanıcı daha programı deneyemeden bir kenara atabilir. *Tabii biz böyle olmasını istemeyiz, değil mi?*

ENTRY PENCERE ARACI

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 *
import tkinter.ttk as ttk

pencere = Tk()
pencere.geometry("200x200")

giriş = ttk.Entry()
giriş.pack()

pencere.mainloop()
```

Her zamanki gibi ilk önce **`tkinter`** ve **`ttk`** modüllerini programımıza dâhil ederek 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 '**giriş**' adını verdiğimiz **`Entry()`** aracını rahatlıkla oluşturabiliyoruz.

Burada tıpkı **`Button()`** pencere aracında olduğu gibi yine, **`ttk`** modülü içindeki **`Entry()` aracını kullandığımıza dikkat edin**:

```
giriş = ttk.Entry()
```

Çünkü **`ttk` modülünün sağladığı `Entry()`** aracı, **`tkinter` modülünün sağladığı `Entry()` aracına kıyasla biraz daha şıktır**.

Bundan sonra yapmamız gereken şey tabii ki bu pencere aracını **görünür hâle** getirmek. Onu da **`giriş.pack()`** satırı yardımıyla hallediyoruz. Son darbeyi ise **`mainloop()`** metoduyla vuruyoruz.

Gördüğünüz gibi kullanıcıya tek satırlık metin girme imkânı veren bir arayüz oluşturmuş olduk. **İsterseniz arayüzümüzü**, **bu pencereye birkaç düğme ekleyerek daha işlevli bir hâle getirelim**. Mesela pencere üzerinde programı **kapatmaya yarayan bir düğme ve kullanıcının yazdığı metni silen bir düğme bulunsun**:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()

def sil():
    giriş.delete(0, END)

giriş = ttk.Entry()
giriş.pack()

düğme1 = ttk.Button(text = "KAPAT", command = pencere.quit)
düğme1.pack()

düğme2 = ttk.Button(text = "SİL", command = sil)
düğme2.pack()

pencere.mainloop()
```

Bu kodlar içinde bize yabancı olan tek ifade **`giriş.delete(0, END)`**. Bu komut, görünüşünden de anlaşılacağı gibi **`giriş` 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:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()
```

```
def oluştur():
    dosya = open("deneme.txt", "w")

    metin = giriş.get()
    dosya.write(metin)

giriş = ttk.Entry()
giriş.pack()

düğme = ttk.Button(text = "OLUŞTUR", command = oluştur)
düğme.pack(side=LEFT)

düğme2 = ttk.Button(text = "ÇIK", command = pencere.quit)
düğme2.pack(side=RIGHT)

pencere.mainloop()
```

Bu kodlardaki **giriş.get()** ifadesi **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 bu programı hangi klasör altında çalıştırdıysanız orada oluşmuş olmalı.

Burada ayrıca **side=LEFT** ve **side=RIGHT** şeklinde birtakım ifadeler de görüyoruz. **side** parametresi, **bir pencere aracının pencere üzerinde hangi tarafa yerleştirileceğini belirliyor**. Biz bu kodlarda ilk düğmeyi **sol tarafa** (**LEFT**), ikinci düğmeyi ise **sağ tarafa** (**RIGHT**) yerleştirdik.

Bir örnek daha yapalım...

```
from tkinter import *
import tkinter.ttk as ttk

import random

pencere = Tk()
pencere.geometry("200x200")
```

```
def bas():
    a = random.randint(1, 100)
    giriş.delete(0, END)
    giriş.insert(0, a)

giriş = ttk.Entry()
giriş.pack()

düğme = ttk.Button(text="bas", command=bas)
düğme.pack()

pencere.mainloop()
```

Gördüğünüz gibi bu uygulama **1 ile 100 arasında rastgele sayılar seçiyor**.

Aslında yaptığımız işlem çok basit:

Öncelikle Python'ın **`random`** adlı modülünü **çağırdık**. Daha önce de gördüğümüz gibi rastgele sayılar seçerken bize, **bu modül yardımcı olacak**. Ardından da bu rastgele sayıları oluşturup ekrana yazdırmamızı sağlayacak fonksiyonu oluşturuyoruz. Bu fonksiyonda öncelikle **rastgele sayıların** hangi aralıkta olacağını belirleyip bunu **`a`** adlı **bir değişkene atıyoruz**. Böylece rastgele sayıları ekrana yazdırmak için gereken altyapıyı oluşturmuş olduk.

Şimdi, bu noktada eğer **bir önlem almazsak ekrana basılacak sayılar yan yana sıralanacaktır**. Yani mesela diyelim ki **ilk rastgele sayımız 3** olsun. Bu 3 sayısı ekrana yazıldıktan sonra ikinci rastgele sayı ekrana gelmeden önce bu ilk sayının ekrandan **silinmesi** gerekiyor. **Bütün sayılar yan yana ekrana dizilmemeli. Bunun için kodumuza şu satırı ekliyoruz**:

```
giriş.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 **`ttk.Entry()`** ve **`ttk.Button()`** adlı pencere araçları yardımıyla düğmelerimizi ve metin alanımızı oluşturuyoruz.

Frame Pencere Aracı

`Frame()`, `Tkinter` içinde bulunan sınıflardan biridir. Bu sınıf, aynı zamanda bir pencere aracı (*widget*) olarak da kullanılabilir. Bir pencere aracı olarak kullanıldığında bu sınıfın görevi, **pencere içindeki öbür pencere araçlarını pencere içinde rahatça konumlandırmayı sağlamaktır**. Bu araçtan, ileride geometri yöneticilerini (*geometry managers*) işlerken daha ayrıntılı olarak bahsedeceğiz. Bu derste **`Frame()`** pencere aracının temel olarak nasıl kullanıldığına dair birkaç **örnek** vermekle yetineceğiz. Şu örneğe bir bakalım:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()

etiket = Label(text="Aşağıdaki kutucuğa e.posta adresinizi
yazınız!")
etiket.pack()

giriş = ttk.Entry()
giriş.pack()

düğme = ttk.Button(text="Gönder", command=pencere.destroy)
düğme.pack()

pencere.mainloop()
```

Burada gördüğünüz gibi "**Gönder**" düğmesi hemen üstündeki kutucuğa çok yakın duruyor. Bu arayüzün daha iyi görünmesi için bu iki pencere aracının arasını biraz açmak isteyebiliriz.

İşte **Frame()** aracı, böyle bir ihtiyaç hâlinde işimize yarayabilir:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()

etiket = Label(text="Aşağıdaki kutucuğa e.posta adresinizi
yazınız!")
etiket.pack()

giriş = ttk.Entry()
giriş.pack()

çerçeve = Frame()
çerçeve.pack()

düğme = ttk.Button(text="Gönder", command=pencere.destroy)
düğme.pack()

pencere.mainloop()
```

Gördüğünüz gibi **Frame()** aracını **giriş** ve **düğme** adlı pencere araçlarının arasına yerleştirdik. Bu kodları çalıştırdığımızda, düğmenin azıcık da olsa **aşağıya kaydığını** göreceksiniz. Eğer kayma oranını artırmak isterseniz **pad** seçeneğinden yararlanabilirsiniz. Bu seçenek, bir pencere aracının, öteki pencere araçları ile arasını açmaya yarıyor. **pad** seçeneği iki şekilde kullanılabilir: **padx** ve **pady**. Bu iki seçenekten **padx**, **x** düzlemi üzerinde işlem yaparken **pady** ise **y** düzlemi üzerinde işlem yapıyor. **Yani**:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()

etiket = Label(text="Aşağıdaki kutucuğa e.posta adresinizi
yazınız!")
etiket.pack()
```

```
giriş = ttk.Entry()
giriş.pack()

çerçeve = Frame()
çerçeve.pack(pady=5)

düğme = ttk.Button(text="Gönder", command=pencere.destroy)
düğme.pack()

pencere.mainloop()
```

Burada sadece **pady** seçeneğini kullanmamız yeterli olacaktır. Çünkü bizim bu örnekte istediğimiz şey, iki pencere aracının arasını y düzlemi üzerinde açmak. **Yani yukarıdan aşağıya doğru...**

Eğer x düzlemi üzerinde işlem yapmak isteseydik (*yani soldan sağa doğru*), o zaman **padx** seçeneğini de kullanacaktık. İsterseniz bu **padx** ve **pady** seçeneklerine **bazı sayılar vererek birtakım denemeler yapabilirsiniz**. Bu sayede bu seçeneklerin ne işe yaradığı daha net anlaşılacaktır. Ayrıca dikkat ederseniz bu **padx** ve **pady** seçenekleri, daha önce gördüğümüz **side** seçeneğine kullanım olarak çok benzer. Hem **padx** ve **pady** hem de **side** seçenekleri, **pack()** aracının **birer özelliğidir**.

Şimdilik bu **Frame()** pencere aracını burada bırakıyoruz. *Bir sonraki bölümde bu araçtan ayrıntılı olarak bahsedeceğiz*.

Böylelikle **Pencere Araçları** konusunun ilk bölümünü bitirmiş olduk. Şimdi, başka bir pencere aracı grubuna geçmeden önce çok önemli bir konuya değineceğiz: **Tkinter'de Geometri Yöneticileri** (**Geometry Managers**).

GEOMETRİ YÖNETİCİLERİ

3

Şimdiye kadar yaptığımız örneklerden de gördüğünüz gibi bir pencere aracını kullanıcılara gösterebilmemiz için **bu aracı tanımlamak yeterli olmuyor**. Aracı tanımlamanın yanı sıra **özel bir metodu** da kullanmamız gerekiyor. Şu ana dek Tkinter ile hazırladığımız örneklerde **pencere araçlarını kullanıcıya gösterebilmek için `pack()` adlı özel bir metottan yararlandık**. Bu metodun kullanımına şöyle bir **örnek** verebiliriz:

```
from tkinter import *

pencere = Tk()
pencere.geometry("200x200")

dolar = Label(text="Dolar", fg = "white", bg = "red",
              font=("Verdana", "13", "bold"), width=10)
dolar.pack()

avro = Label(text="Avro", fg = "white", bg = "blue",
             font=("Verdana", "13", "bold"), width=10)
avro.pack()

lira = Label(text="Lira", fg = "white", bg = "green",
            font=("Verdana", "13", "bold"), width=10)
lira.pack()

pencere.mainloop()
```

Burada **fg**, **bg**, **font** ve **width** seçenekleri yardımıyla her bir etiketin sırasıyla **ön plan rengini**, **arka plan rengini**, **yazı tipini** ve **genişliğini** belirledik.

Bu arada eğer yukarıdaki kodlarda **tkinter** modülünün **Label()** pencere aracı yerine **ttk** modülündeki kullanmak isterseniz **yukarıdaki görüntüyü elde etmek için şu kodları yazabilirsiniz**:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()
pencere.geometry("200x200")

stil = ttk.Style()
stil.configure("TLabel",
               foreground="white",
               font=("Verdana", "13", "bold"),
               width=10, anchor="center")

stil.configure("dolar.TLabel",
               background="red")

stil.configure("avro.TLabel",
               background="blue")

stil.configure("lira.TLabel",
               background="green")

dolar = ttk.Label(text="Dolar", style="dolar.TLabel")
dolar.pack()

avro = ttk.Label(text="Avro", style="avro.TLabel")
avro.pack()

lira = ttk.Label(text="Lira", style="lira.TLabel")
lira.pack()

pencere.mainloop()
```

Burada gördüğünüz gibi **`Style()`** sınıfı aracılığıyla önce genel bir "**`TLabel`**" **stili tanımladık**:

```
stil.configure("TLabel",
               foreground="white",
               font=("Verdana", "13", "bold"),
               width=10, anchor="center")
```

Bu stil içinde tüm **`Label()`** araçları için sırasıyla **arka plan**, **yazı tipi**, **genişlik** ve **hizalama** bilgilerini ayarladık. Burada metni etiket içinde **ortalamak için `anchor` adlı bir seçeneğe** "`center`" **değerini verdik**.

Daha sonra her bir etiket için farklı olan özellikleri belirlemek üzere a**yrı stiller kullandık**:

```
stil.configure("dolar.TLabel",
               background="red")

stil.configure("avro.TLabel",
               background="blue")

stil.configure("lira.TLabel",
               background="green")
```

Burada **dolar**, **avro ve lira etiketlerinin arka plan renklerini ayrı ayrı belirledik**. Sonra da bunları ilgili etiketlerin **`style`** parametresi içine yazdık:

```
dolar = ttk.Label(text="Dolar", style="dolar.TLabel")
avro = ttk.Label(text="Avro", style="avro.TLabel")
lira = ttk.Label(text="Lira", style="lira.TLabel")
```

Tercihinize göre **`tkinter`** veya **`ttk`** pencere araçlarını seçmekte özgürsünüz. Hangisi kolayınıza geliyorsa onu seçebilirsiniz. Biz tercihi size bırakarak yolumuza devam edelim...

Yukarıdaki pencere araçlarını gösterebilmek için kullandığımız **`pack()`** adlı metoda Tkinter'de **geometri yöneticisi** (*geometry manager*) adı veriliyor.

Anladığınız gibi geometri yöneticilerinin **temel işlevi**, **Tkinter ile oluşturulan pencere araçlarını** (*yani widget'leri*) **pencere üzerinde görünür hâle getirmektir**. Tkinter'de esasen üç farklı geometri yöneticisi bulunur. **Bu geometri yöneticilerini şöyle sıralayabiliriz**:

- `pack()`
- `grid()`
- `place()`

Geometri yöneticileri bir pencere aracının **konumunu** belirler. Bu geometri 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 ustadır**. O yüzden, pencere araçlarını pencere üzerinde hassas bir şekilde konumlandırmak istediğimiz zaman `place()` geometri yöneticisinden yararlanabiliriz.

Geometri yöneticileri ile ilgili olarak aklımızda tutmamız gereken en önemli şey; **aynı pencere üzerinde birbirinden farklı yöneticileri bir arada kullanmamaktır**. Bunun ne demek olduğunu ileride örneklerle gösterdiğimiz zaman çok daha iyi anlayacaksınız.

Bu geometri yöneticileri içinde **en kolay olanı** `pack()` adlı yöneticidir. Zaten bu sebeple şimdiye kadar gördüğümüz örneklerde de hep bu `pack()` metodunu kullandık. Ancak hangi geometri yöneticisini kullanırsak kullanalım, öteki yöneticilerin nasıl kullanıldığını da öğrenmemiz gerekir. Çünkü bu geometri yöneticilerinin **her birinin iyi olduğu bir alan vardır**. Şimdi, isterseniz bu geometri yöneticilerini *tek tek inceleyelim*.

Pack Geometri Yöneticisi

Bir önceki bölümde de söylediğimiz gibi **`pack()` geometri yöneticisi Tkinter'deki geometri yöneticileri içinde en kolay olanıdır**. Bu geometri yöneticisinin kullanımıyla ilgili basit bir örneği önceki bölümde vermiştik. İsterseniz **bu örneği tekrar önümüze alalım**:

```
from tkinter import *

pencere = Tk()
pencere.geometry("200x200")

dolar = Label(text="Dolar", fg = "white", bg = "red",
              font=("Verdana", "13", "bold"), width=10)
dolar.pack()

avro = Label(text="Avro", fg = "white", bg = "blue",
             font=("Verdana", "13", "bold"), width=10)
avro.pack()

lira = Label(text="Lira", fg = "white", bg = "green",
            font=("Verdana", "13", "bold"), width=10)
lira.pack()

pencere.mainloop()
```

`pack()` metodunu parametresiz olarak kullandığımızda **ögelerin alt alta sıralandığına dikkat edin**. Ancak gördüğünüz gibi **`pack()`** metodunu bu şekilde kullandığımızda **etiketler pencereye orantısız olarak dağılıyor**. Yani görünüş biraz özensiz duruyor. Ama eğer biz istersek bu durumu bir nebze olsun düzeltebiliriz. Bunun için ihtiyacımız olan şey, **`pack()`** geometri yöneticisinin **expand** adlı seçeneğidir.

EXPAND SEÇENEĞİ

expand seçeneği, **pencere araçlarının pencere üzerinde daha orantılı bir şekilde dağılmasını sağlar**. Bu **expand** seçeneği **YES** ve **NO** olmak üzere iki farklı değer alır. Varsayılan değer **NO**'dur. Bunu yukarıdaki örnek üzerinde görelim:

```
from tkinter import *

pencere = Tk()
pencere.geometry("200x200")

dolar = Label(text="Dolar", fg = "white", bg = "red",
              font=("Verdana", "13", "bold"), width=10)
dolar.pack(expand=YES)

avro = Label(text="Avro", fg = "white", bg = "blue",
             font=("Verdana", "13", "bold"), width=10)
avro.pack(expand=YES)

lira = Label(text="Lira", fg = "white", bg = "green",
            font=("Verdana", "13", "bold"), width=10)
lira.pack(expand=YES)

pencere.mainloop()
```

expand kelimesi İngilizcede **genişlet**, **yay** gibi anlamlara gelir. Yukarıdaki örnekte gördüğünüz gibi **pack()** metoduna parametre olarak **expand = YES** seçeneğini vermek yoluyla ögelerin **pencere üzerine dağılmasını** veya **yayılmasını sağlıyoruz**. Eğer **expand = NO** gibi bir biçim kullanırsak ögeler pencere üzerine yayılmayacak, yani **sanki bu seçeneği hiç kullanmamışız gibi davranılacaktır**.

Gördüğünüz gibi ögeler yukarıdaki şekilde birbirinden ayrı duruyor. Biz bu ögelerin birbirine değmesini de isteyebiliriz. **Yani tercihimiz, tıpkı bazı ülkelerin bayraklarında olduğu gibi bu üç rengin şeritler hâlinde birbirine değecek şekilde uzanmasını sağlamak olabilir**. Bu iş için kullanabileceğimiz ayrı bir seçenek var. Bu seçeneğin adı: **fill**.

FILL SEÇENEĞİ

Bu '**fill**' kelimesi, İngilizcede **doldurmak** anlamına gelir. Bu seçenek yardımıyla, araçlarımızın pencereyi doldurmasını sağlayacağız. **fill** seçeneği üç farklı değer alabilir: **X**, **Y** ve **BOTH**.

X seçeneği, **pencere aracımızın, içinde bulunduğu pencereyi x düzlemi boyunca doldurmasını sağlar** (*yani soldan sağa doğru*).

Y seçeneği, **pencere aracımızın, içinde bulunduğu pencereyi y düzlemi boyunca doldurmasını sağlar** (*yani yukarıdan aşağıya doğru*).

BOTH seçeneği, **pencere aracımızın, içinde bulunduğu pencereyi x ve y düzlemi boyunca doldurmasını sağlar** (*both kelimesi İngilizce'de her ikisi de anlamına gelir*).

Şimdi, bu seçenekleri örnekler üzerinde görerek kafamızda somutlaştıralım:

```
from tkinter import *

pencere = Tk()
pencere.geometry("200x200")

dolar = Label(text="Dolar", fg = "white", bg = "red",
              font=("Verdana", "13", "bold"), width=10)
dolar.pack(expand=YES, fill=X)

avro = Label(text="Avro", fg = "white", bg = "blue",
             font=("Verdana", "13", "bold"), width=10)
avro.pack(expand=YES, fill=Y)

lira = Label(text="Lira", fg = "white", bg = "green",
            font=("Verdana", "13", "bold"), width=10)
lira.pack(expand=YES, fill=BOTH)

pencere.mainloop()
```

Burada, **her bir ögede ayrı bir değer kullandık**. Ama eğer siz isterseniz her üçü için tek bir değer kullanarak bu değerlerin tam olarak ne yaptığını daha net görmeyi tercih edebilirsiniz.

SIDE SEÇENEĞİ

side seçeneği bir pencere aracını istediğimiz bir yöne yaslamamızı sağlıyor. Bu seçenek dört farklı değer alır: **TOP**, **BOTTOM**, **LEFT** ve **RIGHT**.

- **TOP:** Pencere aracını üste yaslar.
- **BOTTOM:** Pencere aracını dibe yaslar.
- **LEFT:** Pencere aracını sola yaslar.
- **RIGHT:** Pencere aracını sağa yaslar.

Ancak bu dört değer içinde en sık kullanılanlar **TOP** ve **LEFT**'tir. Eğer pencere araçlarını yan yana sıralamak istersek kısaca **LEFT** değerini kullanabiliriz. Eğer pencere araçlarını üst üste sıralamak istersek de **TOP** değerini kullanabiliriz. **Şu iki örneği dikkatlice inceleyin**:

LEFT ile kullanım:

```
from tkinter import *

pencere = Tk()
pencere.geometry("200x100+50+100")

dolar = Label(text="Dolar", fg = "white", bg = "red",
              font=("Verdana", "13", "bold"))
dolar.pack(side=LEFT)

avro = Label(text="Avro", fg = "white", bg = "blue",
             font=("Verdana", "13", "bold"))
avro.pack(side=LEFT)

lira = Label(text="Lira", fg = "white", bg = "green",
             font=("Verdana", "13", "bold"))
lira.pack(side=LEFT)

pencere.mainloop()
```

TOP ile kullanım:

```
from tkinter import *

pencere = Tk()
pencere.geometry("200x100+50+100")

dolar = Label(text="Dolar", fg = "white", bg = "red",
              font=("Verdana", "13", "bold"))
dolar.pack(side=TOP)

avro = Label(text="Avro", fg = "white", bg = "blue",
             font=("Verdana", "13", "bold"))
avro.pack(side=TOP)

lira = Label(text="Lira", fg = "white", bg = "green",
             font=("Verdana", "13", "bold"))
lira.pack(side=TOP)

pencere.mainloop()
```

Ayrıca istersek **birkaç seçeneği bir arada kullanarak** aşağıdaki şekilde bir şey de yapabiliriz:

```
from tkinter import *

pencere = Tk()
pencere.geometry("200x100+50+100")

dolar = Label(text="Dolar", fg = "white", bg = "red",
              font=("Verdana", "13", "bold"))
dolar.pack(side=TOP, expand=YES, fill=BOTH)

avro = Label(text="Avro", fg = "white", bg = "blue",
             font=("Verdana", "13", "bold"))
avro.pack(side=TOP, expand=YES, fill=BOTH)
```

```
lira = Label(text="Lira", fg = "white", bg = "green",
             font=("Verdana", "13", "bold"))
lira.pack(side=TOP, expand=YES, fill=BOTH)

pencere.mainloop()
```

Bu pencereyi **elle sağa-sola** veya **yukarı-aşağı doğru** boyutlandırın. Gördüğünüz gibi pencereyi ne şekilde boyutlandırırsanız boyutlandırın, **etiket ögeleri pencereyi doldurmaya devam ediyor**.

Elbette **pack()** geometri yöneticisi **yalnızca etiketlerle birlikte kullanılmıyor**. Öğrendiğimiz öteki pencere araçlarını da kullanarak örneğin, şöyle bir şey yapabiliriz:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()

uyarı = Label(pencere, text="Programdan çıkmak istediğinize \
emin misiniz?")
uyarı.pack()

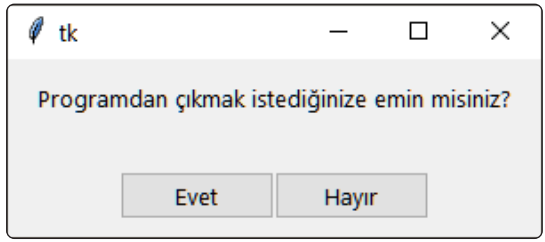
evet = ttk.Button(text="Evet", command=pencere.quit)
evet.pack()

hayır = ttk.Button(text="Hayır")
hayır.pack()

pencere.mainloop()
```

Burada **pack()** metotları içinde değişik seçeneklere farklı değerler uygulayarak istediğinize yakın bir görüntü elde edebilirsiniz.

Ancak aşağıdaki gibi bir görüntü elde etmek için biraz uğraşmak gerekiyor:



Evet-hayır penceresi

Yukarıdaki gibi bir görüntü elde edebilmek için daha önce gördüğümüz **`Frame()` adlı pencere aracından yararlanmanız gerekir**. `Frame()` aracının nasıl kullanıldığını hatırlıyorsunuz:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()
pencere.geometry("300x100")

yeni = Frame()
yeni.pack(side=BOTTOM, pady=10)

uyarı = Label(pencere, text="Programdan çıkmak istediğinize \
emin misiniz?")
uyarı.pack(pady=10)

evet = ttk.Button(yeni, text="Evet", command=pencere.quit)
evet.pack(side=LEFT)

hayır = ttk.Button(yeni, text="Hayır")
hayır.pack()

pencere.mainloop()
```

Burada dikkatimizi çekmesi gereken **bazı çok önemli yenilikler var**.

Mesela yukarıda geçen şu satıra bakın:

```
uyarı = Label(pencere, text="Programdan çıkmak istediğinize emin
misiniz?")
```

Burada **text** parametresinden önce '**pencere**' değerini belirtmemiz, bizim için bir **yenilik** sayılır. Daha önceden yazdığımız pencere araçlarında, bu '**pencere**' ifadesini kullanmamıza gerek yoktu. Çünkü yazdığımız programlarda hep **sadece tek bir taşıyıcı** pencere aracı (*container widget*) yer alıyordu. *Bu taşıyıcı pencere aracı ifadesi de ne oluyor*, **diye sorarsanız hemen açıklayalım**: Görevi başka pencere araçlarını taşımak olan pencere araçlarına Tkinter'de "**taşıyıcı pencere aracı**" (*container widget*) adı veriliyor. Mesela yazdığımız bütün programlarda **pencere = Tk** ifadesi ile oluşturduğumuz ana penceremiz, esasında bir **taşıyıcı pencere** aracıdır. Yani **bu pencerenin görevi pencere araçlarını tutmaktır**. Aynı şekilde **Frame()** de bir taşıyıcı pencere aracıdır. Bu pencere aracı da tıpkı ana penceremiz gibi **başka pencere araçlarını üzerinde taşıyabilir**. Mesela yukarıdaki örnekte **Frame()** pencere aracı bu görevde kullanılıyor. Nasıl **uyarı** adlı etiketi pencere adlı taşıyıcının üstüne yerleştirmişsek **evet** ve **hayır** adlı pencere araçlarını da yeni adlı taşıyıcının üzerine yerleştiriyoruz. Bu yerleştirme işlemini, yukarıda gördüğünüz gibi pencere araçlarının **ilk argümanı olarak** herhangi bir taşıyıcı adı belirtmek suretiyle gerçekleştiriyoruz. Mesela **evet** adlı pencere aracını **yeni adlı taşıyıcı üzerine yerleştirmişiz**. Bunu, **ttk.Button(yeni, text="Evet".....")** komutundaki ilk parametre olan **yeni** ifadesinden anlıyoruz. Aynı şekilde **uyarı** adlı pencere aracını **pencere** adlı taşıyıcının üzerine yerleştirdiğimizi de **Label(pencere, text="Pr...")** komutundaki **ilk parametre olan pencere** ifadesinden anlıyoruz.

Bu kodlarda, ayrıca **pady** seçeneğini kullanarak **yeni** ve **uyarı** adlı pencere araçlarını **öteki pencere araçlarından bir miktar uzaklaştırdığımıza da dikkat edin**.

Peki, **`Frame()`** aracını (*ve* **`pack()`** *yöneticisini*) kullanarak başka neler yapabiliriz? Hemen bir **örnek** daha verelim:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()

etiket = Label(text="Aşağıdaki kutucuğa e.posta \
adresinizi yazınız!")
etiket.pack()

giriş = ttk.Entry()
giriş.pack()

çerçeve = Frame(height=2, bd=1, relief=SUNKEN)
çerçeve.pack(fill=X, pady=5, padx=5)

düğme = ttk.Button(text="Gönder", command=pencere.destroy)
düğme.pack()

pencere.mainloop()
```

Burada birkaç yeni özellik görüyoruz. Daha önce görmediğimiz özellikler **`bd`** ve **`relief`** seçenekleridir. **`bd`** İngilizcede "**kenar, kenarlık, sınır**" anlamına gelen "**border**" kelimesinin kısaltması. Bu seçenek yardımıyla **`Frame()`** aracının **kenarlık ölçülerini belirliyoruz**. Varsayılan olarak **`Frame()`** aracının kenarlık boyutu 0'dır. Biz bunu yukarıda **`1`** olarak değiştirdik. Aynı satırda gördüğümüz **`height`** seçeneği ise bir pencere aracının **yüksekliğini** ayarlamamızı sağlıyor. Yukarıdaki örnekte de **`Frame()`** pencere aracının yüksekliğini **`2`** olarak ayarladık. **`relief`** seçeneği ise İngilizce bir kelime olarak Türkçede "**kabartma**" (*rölyef*) anlamına geliyor.

Dolayısıyla bu seçenek yardımıyla pencere aracının **kabartma tipini belirliyoruz**. Burada **altı farklı stil uygulayabiliriz**:

- **SOLID**
- **FLAT**
- **RIDGE**
- **SUNKEN**
- **GROOVE**
- **RAISED**

Bu stillerin nasıl bir görüntü ortaya çıkardığını daha net görebilmek için **`height`** ve **`bd`** seçeneklerini *daha yüksek değerlere ayarlamayı tercih edebilirsiniz*.

`Frame()` aracını kullanarak mesela şöyle bir şey de yapabiliriz:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()

etiket = Label(text="Aşağıdaki kutucuğa e.posta \
adresinizi yazınız!")
etiket.pack()

giriş = ttk.Entry()
giriş.pack()

çerçeve = Frame()
çerçeve.pack(side=BOTTOM, padx=5, pady=5)

düğme = ttk.Button(çerçeve, text="Gönder", width=8, command=pencere.
destroy)
düğme.pack(side=LEFT, padx=5)

düğme2 = ttk.Button(çerçeve, text="İptal", width=8)
düğme2.pack()

pencere.mainloop()
```

Burada, pencere **aracı seçeneklerini nasıl kullandığımıza dikkat edin**.

pack() metodunu kullanarak **çok karmaşık arayüz görüntüleri** elde edebiliriz. Aslında bu metodu kullanarak *yapamayacağımız şey yok gibidir*. Ama bazı şeyleri bu metotla yapmaya kalkışmak *bir süre sonra işkenceye dönüşebilir*. O yüzden yeri geldiğinde **pack()** yerine başka metotları kullanmayı da bilmeliyiz. Mesela bazı işler için **grid()** metodu daha **uygun** olabilir. **grid()** metodunu anlatmaya geçmeden önce isterseniz **pack()** metodunu **nerelerde kullanabileceğimizi şöyle bir sıralayalım**:

- Pencere araçlarını alt alta dizmek istediğimizde,
- Pencere araçlarını yan yana dizmek istediğimizde,
- Pencere araçlarını alt alta veya yan yana dizip bunların pencereyi (*x veya y düzleminde ya da her ikisinde birden*) tamamen kaplamasını istediğimizde.

Bu metodu nerelerde kullanmamızın uygun olacağını gördüğümüze göre artık *başka bir geometri yöneticisini anlatmaya başlayabiliriz*.

Grid Geometri Yöneticisi

Bu geometri yöneticisi, pencere araçlarının ızgara biçiminde sıralanacağı arayüzler için uygundur. Bununla hemen basit bir örnek yapalım:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()

düğme1 = ttk.Button(text="Ürünler")
düğme1.grid()

düğme2 = ttk.Button(text="Hizmetler")
düğme2.grid()

düğme3 = ttk.Button(text="Ulaşım")
düğme3.grid()

düğme4 = ttk.Button(text="Hakkında")
düğme4.grid()

pencere.mainloop()
```

grid() adlı geometri yöneticisi basitçe bu şekilde kullanılıyor. Ama tabii ki bu hâliyle pek cazip görünmüyor. Buna istediğimiz şekli verebilmek için yine **birtakım seçeneklerden** yararlanmamız gerekecek.

Bu arada yukarıdaki kodları daha rahat anlayabilmeniz için düğmeleri böyle **tek tek** oluşturduk. Ama aslında bu düğmeleri bir **for** döngüsü içinde de oluşturabilirsiniz. **Dikkatlice bakın**:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()

başlıklar = ["Ürünler", "Hizmetler", "Ulaşım", "Hakkında"]

for başlık in başlıklar:
    ttk.Button(text=başlık).grid()

pencere.mainloop()
```

Burada öncelikle tüm başlıkları **tek bir liste içinde topladık**:

```
başlıklar = ["Ürünler", "Hizmetler", "Ulaşım", "Hakkında"]
```

Daha sonra bu liste üzerinde bir **for** döngüsü kurarak **tüm düğmeleri bu döngü içinde tek satırda oluşturduk**.

Bu örneği de verdiğimize göre **grid()** yöneticisinin seçeneklerini incelemeye geçebiliriz.

row Seçeneği

Bu geometri yöneticisinin en önemli iki seçeneğinden birincisi, **row** seçeneğidir. Bu kelime İngilizcede **satır** anlamına gelir. Bu seçenek yardımıyla, **pencere aracının hangi satırda yer alacağını belirleyeceğiz**. Bununla herhangi bir örnek yapmadan önce, isterseniz **grid()** geometri yöneticisinin öteki seçeneğinden de bahsedelim. Çünkü bu seçenek hep **row** seçeneğiyle **birlikte kullanılır**.

COLUMN SEÇENEĞİ

"**Column**" kelimesi İngilizcede **sütun** anlamına gelir. Dolayısıyla bu seçenek yardımıyla **pencere aracımızın pencere üzerinde hangi sütunda yer alacağını belirleyeceğiz**. Artık bu iki seçeneği kullanarak bir örnek verebiliriz.

Aşağıdaki **örneği dikkatle inceleyin**:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()

dolar = Label(text="Dolar")
dolar.grid(row=0, column=0)

avro = Label(text="Avro")
avro.grid(row=1, column=0)

lira = Label(text="Lira")
lira.grid(row=2, column=0)

dolar_g = ttk.Entry()
dolar_g.grid(row=0, column=1)

avro_g = ttk.Entry()
avro_g.grid(row=1, column=1)

lira_g = ttk.Entry()
lira_g.grid(row=2, column=1)

pencere.mainloop()
```

Gördüğünüz gibi bu **`grid()`** metodu yardımıyla **`pack()`** metoduna kıyasla **daha ince ayarlamalar** yapabiliyoruz. **`grid()`** metodu bize, **pencere aracımızın koordinatlarını daha hassas bir şekilde belirtme imkânı sunuyor**. Şimdi, isterseniz ilk verdiğimiz, cazip görünmeyen örneği yeniden ele alalım.

Örneğimiz şuydu:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()

düğme1 = ttk.Button(text="Ürünler")
düğme1.grid()

düğme2 = ttk.Button(text="Hizmetler")
düğme2.grid()

düğme3 = ttk.Button(text="Ulaşım")
düğme3.grid()

düğme4 = ttk.Button(text="Hakkında")
düğme4.grid()

pencere.mainloop()
```

Şimdi yapacağımız şey, pencere üzerinde dağınık bir şekilde duran bu araçların her biri için uygun bir **satır-sütun belirlemek**. Hemen bakalım:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()

düğme1 = ttk.Button(text="Ürünler")
düğme1.grid(row=0, column=0)

düğme2 = ttk.Button(text="Hizmetler")
düğme2.grid(row=0, column=1)

düğme3 = ttk.Button(text="Ulaşım")
düğme3.grid(row=1, column=0)
```

```
düğme4 = ttk.Button(text="Hakkında")
düğme4.grid(row=1, column=1)

pencere.mainloop()
```

İsterseniz daha önceden gördüğümüz **`fg`**, **`bg`** ve **`font`** gibi seçenekler yardımıyla veya **`Style()`** sınıfını kullanarak **bu düğmelerin ve üzerindeki yazıların renklerini gönlünüze göre değiştirebilirsiniz de**...

Dediğimiz gibi **`grid()`** yöneticisinin **en uygun kullanım alanı ızgara biçimli arayüzlerdir**. Mesela hesap makineleri gibi.

Şu örneğe bir bakalım:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()

#bir hesap makinesinde pencerenin boyutlandırılabilir
#olması anlamsız olacaktır. O yüzden pencereleri
#boyutlandırılamaz hale getiriyoruz.
pencere.resizable(width=FALSE, height=FALSE)

#hesap makinemizin tuş takımında yer alacak
#simgeleri ve sayıları oluşturuyoruz.
liste = [\
"9", "8", "7",
"6", "5", "4",
"3", "2", "1",
"0", "+", "-",
"/", "*", "=",
"C"]

#grid() yöneticisinin row (sıra) ve column (sütun)
#seçenekleri için birer başlangıç değeri belirliyoruz.
sıra = 1
sütun = 0
```

```
#birazdan oluşturacağımız düğmelerin genişlik ve rölyef
#seçeneklerini ayarlamak için bir stil oluşturuyoruz.
stil = ttk.Style()
stil.configure("TButton", width=4, relief=GROOVE)

#tuş takımındaki bütün pencere araçlarını tek bir
#for döngüsü ile oluşturuyoruz.
for i in liste:
    ttk.Button(text=i).grid(row=sıra, column=sütun)
    #column seçeneğinin değeri her defasında bir sayı artıyor.
    sütun += 1
  #sütun'un yani "column'un" değeri 2'yi geçtiğinde
  #column'un değerini sıfırlıyoruz. Böylece tuşlar
  #alt alta dizilebiliyor. row'un (sıra) değeri de
  #her defasında bir sayı artıyor.
    if sütun > 2:
        sütun = 0
        sıra += 1

pencere.mainloop()
```

Gördüğünüz gibi yukarıda bir hesap makinesinde olması gereken temel tuşları, basit bir **for** döngüsü yardımıyla bir çırpıda oluşturduk. Yukarıda kodlar içinde verdiğim yorumları dikkatlice incelemenizi öneririm.

Böylelikle bir geometri yöneticisini daha incelemiş olduk. Sıra geldi *sonuncu geometri yöneticimize...*

Place Geometri Yöneticisi

Tkinter'deki geometri yöneticilerinin üçüncüsü **`place()`** adlı geometri yöneticisidir. **`place()`** geometri yöneticisi ile çalışırken bize **temel olarak iki seçenek yardımcı olacak**: **`relx`** ve **`rely`**. Tahmin edebileceğiniz gibi **`relx`**, **x düzlemi üzerindeki işlemler için** **`rely`** ise **y düzlemi üzerindeki işlemler için kullanılacak**.

Hemen bir **örnek** görelim:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()
pencere.geometry("200x200")

giriş1 = ttk.Entry()
giriş1.place(relx=0.0, rely=0.0, relheight=0.15)

düğme1 = ttk.Button(text="Düğme1")
düğme1.place(relx=0.6, rely=0.0, relheight=0.15)

düğme2 = ttk.Button(text="Düğme2")
düğme2.place(relx=0.0, rely=0.2, relwidth=1)

giriş2 = ttk.Entry()
giriş2.place(relx=0.0, rely=0.35, relwidth=1)

pencere.mainloop()
```

Bu örnekte **`relx`** ve **`rely`** seçeneklerinin yanı sıra; **`relheight`** ve **`relwidth`** seçeneklerini de kullandık. Rahatlıkla anlayabileceğiniz gibi bu **`place()`** yöneticisi yardımıyla, **pencerelerimizi istediğimiz şekilde düzenleyebiliyoruz**. Ama tabii ki gördüğünüz gibi koordinatları tam olarak vermeyi gerektirdiği için bu yöneticiyle çalışmak **biraz emek istiyor**. Bu yüzden bazı işler için bu yöneticiyi kullanmak **pek pratik olmayabilir**. Böyle durumlarda öteki yöneticilerin neler yapabileceğini de göz önünde bulundurmakta fayda var. Bir sonraki konumuz olan **`Checkbutton()`** pencere aracında **`place()`** **adlı geometri yöneticisinden bahsetmeye devam edeceğiz**.

PENCERE ARAÇLARI (2.BÖLÜM)

4

Checkbutton Pencere Aracı

Checkbutton denen şey, bildiğimiz **onay kutusudur**. Basitçe şu şekilde oluşturuyoruz bu onay kutusunu:

```
from tkinter import *

pencere = Tk()

onay = Checkbutton()
onay.pack()

pencere.mainloop()
```

Tabii ki muhtemelen bu onay kutumuzun, hatta kutularımızın **birer adı olsun isteyeceğiz**:

```
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()

pencere.mainloop()
```

Gayet güzel. Ama gördüğünüz gibi ögeler alt alta sıralanırken **hizalı değiller**. Dolayısıyla göze pek hoş görünmüyorlar. Eğer istersek **şöyle bir görünüm verebiliriz onay kutularımıza**:

```
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)

pencere.mainloop()
```

Ögeler böyle yan yana dizildiklerinde fena görünmüyorlar ama biz yine de ögelerin düzgün şekilde **alt alta dizilmesini** isteyebiliriz. Burada geometri yöneticilerinden biri olan **`place()`** adlı olanı tercih edebiliriz. Şimdi, bununla ilgili bir **örnek** görelim.

Mesela bir tane **onay kutusu oluşturup bunu en sola yaslayalım**:

```
from tkinter import *

pencere = Tk()

onay = Checkbutton(text="Kubuntu")
onay.place(relx = 0.0, rely = 0.1)

pencere.mainloop()
```

Bir önceki bölümde öğrendiğimiz **`place()`** yöneticisini nasıl kullandığımızı görüyorsunuz. Bu yöneticinin **`relx`** ve **`rely`** seçenekleri yardımıyla **onay kutusunun koordinatlarını** belirliyoruz. Buna göre onay kutumuz **x** düzlemi üzerinde **0.0 konumunda**; **y** düzlemi üzerinde ise **0.1** konumunda yer alıyor. Yani yukarıdan aşağıya **0.0**; soldan sağa **0.1** konumunda bulunuyor pencere aracımız.

Hemen birkaç tane daha **onay kutusu ekleyelim**:

```
from tkinter import *

pencere = Tk()

onay_pa = Checkbutton(text="Kubuntu")
onay_pa.place(relx = 0.0, rely = 0.1)

onay_de = Checkbutton(text="Debian")
onay_de.place(relx = 0.0, rely = 0.2)

onay_ub = Checkbutton(text="Ubuntu")
onay_ub.place(relx = 0.0, rely = 0.3)

onay_wix = Checkbutton(text="Windows XP")
onay_wix.place(relx = 0.0, rely = 0.4)

pencere.mainloop()
```

Dikkat ederseniz yukarıdaki bütün onay kutularının **relx** seçeneği **0.0** iken **rely** seçenekleri **birer birer artmış**. Bunun nedeni, bütün onay kutularını **sola yaslayıp hepsini alt alta dizmek isteyişimiz**. Bu örnek **relx** ve **rely** seçeneklerinin nasıl kullanılacağı konusunda iyi bir fikir vermiş olmalı. Eğer bir kolaylık sağlayacaksa **relx**'i **sütun**, **rely**'yi ise **satır** olarak düşünebilirsiniz.

Şimdi de mesela **Kubuntu**'yu **Debian**'ın yanına, **Ubuntu**'yu da **Windows XP**'nin yanına gelecek şekilde **alt alta** sıralayalım:

```
from tkinter import *

pencere = Tk()

onay_pa = Checkbutton(text="Kubuntu")
onay_pa.place(relx = 0.0, rely = 0.1)

onay_de = Checkbutton(text="Debian")
onay_de.place(relx = 0.4, rely = 0.1)

onay_ub = Checkbutton(text="Ubuntu")
onay_ub.place(relx = 0.0, rely = 0.2)

onay_wix = Checkbutton(text="Windows XP")
onay_wix.place(relx = 0.4, rely = 0.2)

pencere.mainloop()
```

Kubuntu'yu **0.0** no.lu sütunun **0.1** no.lu satırına; **Debian**'ı da **0.4** no.lu sütunun **0.1** no.lu satırına yerleştirdik. Aynı şekilde **Ubuntu**'yu **0.0** no.lu sütunun **0.2** no.lu satırına; **Windows XP**'yi de **0.4** no.lu sütunun **0.2** no.lu satırına yerleştirdik.

Eğer oluşturduğumuz bu onay kutularına biraz **canlılık katmak istersek** yani *kutunun seçili olup olmamasına göre bazı işler yapmak istersek* kullanmamız gereken bazı özel kodlar var. Önce hemen **yazmamız gereken ilk satırları yazalım**:

```
from tkinter import *

pencere = Tk()
```

Bunun hemen ardından şöyle **iki satır** ekleyelim:

```
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 hâlde 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**:

```
from tkinter import *

pencere = Tk()

d = IntVar()
d.set(0)
```

```
btn1 = Checkbutton(text="Kubuntu", variable=d)
btn1.place(relx=0.0, rely=0.1)

pencere.mainloop()
```

Burada dikkatimizi çeken bir farklılık, **`btn1`** adlı onay kutusunun seçenekleri arasına **`variable`** adlı bir seçeneğin girmiş olması. **Bu seçeneğin işlevini az çok anlamış olmalısınız**: Yukarıda belirlediğimiz **`d=IntVar()`** ve **`d.set(0)`** ifadeleri ile **`Checkbutton()`** pencere aracı arasında **bağlantı kuruyor**. Yani bu seçenek yardımıyla pencerece aracına, "**değişken olarak d'yi kullan**" emrini veriyoruz.

Bu kodları çalıştırdığımızda karşımıza, içinde bir onay kutusu bulunan bir pencere açılır. Bu penceredeki onay kutusu **seçili değildir**. İsterseniz yukarıdaki kodlar içinde yer alan **`d.set(0)`** ifadesini **`d.set(1)`** olarak değiştirip **kodlarımızı öyle çalıştıralım**:

```
from tkinter import *

pencere = Tk()

d = IntVar()
d.set(1)

btn1 = Checkbutton(text="Kubuntu", variable=d)
btn1.place(relx=0.0,rely=0.1)

pencere.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**:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()
pencere.geometry("200x200")

# Onay kutusu için bir değişken oluşturuyoruz
v = IntVar()

# Bu değişkene değer olarak "0" atayalım
v.set(0)

# Öbür onay kutusu için başka bir değişken daha oluşturuyoruz
z = IntVar()

# Bu değişkene de "0" değerini atıyoruz.
z.set(0)

# Bu kez bir karakter değişkeni oluşturalım
d = StringVar()

# Bu değişkenin değeri "Kubuntu" olsun.
d.set("Kubuntu")

# Bir tane daha karakter değişkeni oluşturalım
e = StringVar()

# Bunun da değeri "Debian" olsun.
e.set("Debian")
```

```
# Şimdi onay kutularını seçili hale getirdiğimizde çalışmasını
#istediğimiz komut için bir fonksiyon oluşturuyoruz:
def onay():
    giris.delete(0, END)
    # Eğer "v" değişkeninin değeri "1" ise....
    if v.get() == 1:
        # d.get() ile "d" değişkeninin değerini alıyoruz...
         giris.insert(0, f"Merhaba {d.get()} kullanıcısı")

    # Yok eğer "z" değişkeninin değeri "1" ise...
    elif z.get() == 1:
         # e.get() ile "e" değişkeninin değerini alıyoruz...
         giris.insert(0, f"Merhaba {e.get()} kullanıcısı")

    # Değişkenlerin değer "1" değil, başka bir değer ise..."
    else:
        giris.delete(0, END)

#"text" seçeneğinin değerinin "d.get()" olduğuna dikkat edin.
onay_pa = Checkbutton(text=d.get(), variable=v, command=onay)
onay_pa.place(relx = 0.0, rely = 0.1)

#"text" seçeneğinin değerinin "e.get()" olduğuna dikkat edin.
onay_de = Checkbutton(text=e.get(), variable=z, command=onay)
onay_de.place(relx= 0.0, rely= 0.2)

giris = ttk.Entry(width=200)
giris.place(relx = 0.0, rely=0.0)

pencere.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**:

```
from tkinter import *

pencere1 = Tk()

pencere2 = Toplevel()

pencere1.mainloop()
```

Bu kodları çalıştırdığımızda ekranda **ikinci bir pencerenin daha açıldığını** görürüz. Diyelim ki bu ikinci pencerenin, **bir düğmeye basıldığında açılmasını istiyoruz**:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()

def ekle():
    pencere2 = Toplevel()

btn_pen2 = ttk.Button(text="ekle", command=ekle)
btn_pen2.pack()

pencere.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**:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()

def ekle():
    toplevel = Toplevel()

    toplevel = ttk.Button(toplevel, text="çıkış",
                    command=toplevel.destroy)
    toplevel.pack()

btn_pen2 = ttk.Button(pencere, text="ekle", command=ekle)
btn_pen2.pack()

pencere.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 "**`toplevel`**" ifadesini görüyoruz. **Bunu yapmamızın nedeni şu**: Elimizde **`pencere`** ve **`toplevel`** 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 fazla pencere var ve *siz oluşturulan düğmenin hangi pencere üzerinde görüntüleneceğini belirtmezseniz* örneğin bizim durumumuzda, Tkinter **`ekle`** adlı düğmeye her basışta **otomatik** olarak ana pencere üzerinde **`çıkış`** adlı bir düğme oluşturacaktır. Anladığınız gibi eğer aksi belirtilmemişse pencere araçları **otomatik** olarak ana pencere üzerinde açılacaktır.

Diyelim ki **a**, **b**, **c** ve **d** adlarında **dört adet penceremiz var**. İşte, **hangi düğmenin hangi pencerede görüntüleneceğini belirlemek için** bu özellikten faydalanacağız:

```
btn1 = ttk.Button(a, text="btn1")
btn2 = ttk.Button(b, text="btn2")
btn3 = ttk.Button(c, text="btn3")
btn4 = ttk.Button(d, text="btn4")
```

Yukarıdaki kodlarda yeni bir özellik olarak bir de **command=toplevel.destroy** ifadesini görüyoruz. Aslında **destroy** komutu biraz **quit** komutuna benziyor; görevi, **mevcut pencereyi kapatmak**. Peki, burada **toplevel.destroy** komutu yerine **toplevel.quit** komutunu kullanamaz mıyız? Tabii ki **kullanabiliriz**, ancak kullanırsak **çıkış** düğmesine bastığımızda sadece **toplevel** değil, doğrudan **ana pencerenin kendisi de kapanacaktır**.

Eğer ikinci pencerenin kapanıp ana pencerenin açık kalmasını istiyorsanız kullanmanız gereken komut **destroy**; yok eğer ikinci pencere kapandığında **ana pencere de kapansın istiyorsanız** kullanmanız gereken komut **quit** olmalıdır. '**btn_pen2**' adlı düğmeyi ise **text** seçeneğinin hemen önüne yazdığımız '**pencere**' ifadesinden de anlayacağınız gibi **pencere** adlı pencerenin üzerine yerleştiriyoruz. Yukarıda bahsettiğimiz gibi burada **pencere ifadesini kullanmasanız da olur**, çünkü zaten Tkinter, siz belirtmesiniz de düğmeyi **otomatik** olarak ana pencere üzerinde açacaktır. Tabii düğmeyi, ana pencerede değil de ikincil pencereler üzerinde açmak isterseniz bunu Tkinter'e açıkça söylemeniz gerekir.

Listbox Pencere Aracı

Bu araç bize, pencereler üzerinde **bir liste içeren kutu hazırlama imkânı** veriyor. Hemen basit bir **`Listbox()`** kullanımı örneği görelim:

```
liste = Listbox()
liste.pack()
```

Gördüğünüz gibi bu aracın da kullanımı öteki araçlardan **hiç farklı değil**. İsterseniz bu aracı bir de **bağlamında görelim**:

```
from tkinter import *

pencere = Tk()

liste = Listbox()
liste.pack()

pencere.mainloop()
```

Liste kutumuz bu hâliyle, pencerenin **tamamını kapladığı için** çok belirgin olmayabilir. Ama pencereye bir de düğme eklersek liste kutusu **daha rahat seçilebilir hâle gelecektir**:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()

liste = Listbox()
liste.pack()

btn = ttk.Button(text="ekle")
btn.pack()

pencere.mainloop()
```

Şimdi, liste kutumuza bazı ögeler ekleyelim. Bunun için şu **basit ifadeyi kullanacağız**:

```
liste.insert(END, "öğe 1")
```

Bunu, bu şekilde **tek başına kullanamayız**. Hemen bu parçacığı yukarıdaki kod içine yerleştirelim:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()

liste = Listbox(bg="white")

liste.insert(END, "öğe 1")
liste.pack()

btn = ttk.Button(text="ekle")
btn.pack(pady=5)

pencere.mainloop()
```

Gördüğünüz gibi bu parçacığı, liste kutusunu tanımladığımız satırın **hemen altına** ekledik. Biz liste kutumuza aynı anda **birden fazla öge de eklemek isteyebiliriz**.

Bunun için basitçe bir **for** döngüsü kullanabiliriz:

```
gnulinux_dağıtımları = ["Debian", "Ubuntu", "PclinuxOS",
                        "TruvaLinux", "Gelecek Linux"]

for i in gnulinux_dağıtımları:
    liste.insert(END, i)
```

Hemen bu kodları da **yerli yerine koyalım**:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()

liste = Listbox(bg="white")
liste.pack()

gnulinux_dağıtımları = ["Debian", "Ubuntu", "PclinuxOS",
                        "TruvaLinux", "Gelecek Linux"]

for i in gnulinux_dağıtımları:
    liste.insert(END, i)

btn = ttk.Button(text="ekle")
btn.pack(pady=5)

pencere.mainloop()
```

Gayet güzel bir liste kutusu oluşturduk ve listemizdeki ögeleri 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:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()

liste = Listbox(bg="white")
liste.pack()

gnulinux_dağıtımları = ["Debian", "Ubuntu", "PclinuxOS",
                        "TruvaLinux", "Gelecek Linux"]
```

```
for i in gnulinux_dağıtımları:
    liste.insert(END, i)

def yeni():
    global giriş
    pencere2 = Toplevel()
    giriş = ttk.Entry(pencere2)
    giriş.pack()

    btn2 = ttk.Button(pencere2, text="tamam", command=ekle)
    btn2.pack()

def ekle():
    liste.insert(END, giriş.get())
    giriş.delete(0, END)

btn = ttk.Button(text="ekle", command=yeni)
btn.pack(pady=5)

pencere.mainloop()
```

İsterseniz daha anlaşılır olması için **parça parça ilerleyelim**:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()
```

Bu kısmı zaten biliyoruz. Herhangi bir açıklamaya gerek yok.

```
liste = Listbox(bg="white")
liste.pack()

gnulinux_dağıtımları = ["Debian", "Ubuntu", "PclinuxOS",
                        "TruvaLinux", "Gelecek Linux"]

for i in gnulinux_dağıtımları:
    liste.insert(END, i)
```

Bu kısımda bildiğiniz gibi önce **liste** adında bir liste kutusu (**Listbox()**) oluşturduk. Liste kutumuzun arka plan rengini de **beyaz** olarak belirledik. Ardından da **gnulinux_dağıtımları** adında bir **liste oluşturduk** ve **tek tek ögelerini yazdık**. Hemen alt satırda bir **for** döngüsü yardımıyla **gnulinux_dağıtımları** adlı listedeki bütün ögeleri **liste adı verdiğimiz liste kutusu içine yerleştirdik**:

```
def yeni():
    global giriş
    pencere2 = Toplevel()
    giriş = ttk.Entry(pencere2)
    giriş.pack()

    btn2 = ttk.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 giriş 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 giriş** adlı **bir satır daha ekledik**. Bu satırın amacı, fonksiyon içindeki giriş 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, giriş.get())
    giriş.delete(0, END)
```

İşte bu parçada da **ekle()** adlı bir fonksiyon oluşturduk. Burada **liste.insert** komutu **liste** adlı liste kutusuna **ekleme yapmamızı** sağlıyor. Parantez içindeki **giriş.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 **`giriş`** adlı **`Entry()`** pencere aracının içeriğini alıp **`liste`** adlı **liste kutusu içine yerleştiriyoruz**. Alt satırdaki **`giriş.delete(0, END)`** ifadesi ise **`Entry()`** aracına kullanıcı tarafından giriş yapıldıktan sonra **kutunun boşaltılmasını sağlıyor**:

```
btn = ttk.Button(text="ekle", command=yeni)
btn.pack(pady=5)
```

Bu son parçada bilmediğimiz hiçbir şey yok. Burada basit bir **`Button()`** aracı tanımlıyoruz. İsmini belirledikten sonra da **`command` seçeneği yardımıyla** yukarıda tanımladığımız **`yeni()`** adlı fonksiyonu bu düğmeye bağlıyoruz.

Kodlarımızı çalıştırdığımızda karşımıza gayet hoş, üstelik güzel de bir işlevi olan bir pencere çıkıyor. Burada "**`ekle`**" düğmesine bastığımızda **karşımıza yeni bir pencere açılıyor**. Bu yeni pencerede, kullanıcının giriş yapabilmesi için bir **`Entry()`** aracı, bir de işlemi tamamlayabilmesi için **`Button()`** aracı yer alıyor. Kullanıcı **`Entry()`** aracına bir veri girip **`tamam`** düğmesine bastığında **`Entry()`** aracına girilen veri, ana penceredeki **liste kutusuna ekleniyor**. Ayrıca ilk veri girişinin ardından **`Entry()`** aracı içindeki alan tamamen *boşaltılıyor* ki **kullanıcı rahatlıkla ikinci veriyi girebilsin**.

Burada dikkat ederseniz ikinci pencerede, giriş kutusuna **hiçbir şey yazmadan** **`tamam`** düğmesine basarsak ana penceredeki liste kutusuna **boş bir satır ekleniyor**. Şimdi öyle bir kod yazalım ki kullanıcı eğer ikinci penceredeki giriş kutusuna **hiçbir şey yazmadan tamam düğmesine basarsa** giriş kutusu içinde "**`Veri Yok!`**" **yazısı belirsin** ve **bu yazı ana penceredeki liste kutusuna eklenmesin**.

Bunun için kodlarımız içindeki **`ekle()`** fonksiyonuna iki adet **`if`** deyimi eklememiz gerekiyor:

```
def ekle():
    if not giriş.get():
        giriş.insert(END, "Veri Yok!")
    if giriş.get() != "Veri Yok!":
        liste.insert(END, giriş.get())
        giriş.delete(0, END)
```

Gördüğünüz gibi '**giriş**' 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 hâle getirebiliriz:

```
def ekle():
    if not giriş.get():
        giriş.insert(END, "Veri Yok!")
    if not "Veri Yok!" in giriş.get():
        liste.insert(END, giriş.get())
        giriş.delete(0, END)
```

Yani şöyle demiş oluyoruz bu ifadelerle: "Eğer **giriş** adlı **Entry()** aracı **boş** ise araç içinde '**Veri Yok!**' ifadesini göster. Eğer **giriş** adlı **Entry()** aracı içinde '**Veri Yok!**' ifadesi bulunmuyorsa **liste** adlı **Listbox()** aracına giriş içindeki bütün veriyi yazdır!"

Liste kutumuza ögelerimizi ekledik. *Peki, bu ögeleri silmek istersek ne yapacağız?*

Niyetimiz liste kutusundan öge silmek olduğunu göre en başta bir **silme düğmesi** oluşturmamız mantıklı olacaktır:

```
btn_sil = ttk.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 ögenin silinmesini sağlayacak**. *Yani aktif olan öge silinecek*.

Bu iki parçayı öteki komutlarla birlikte bir görelim bakalım:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()

liste = Listbox(bg="white")
liste.pack()

gnulinux_dağıtımları = ["Debian", "Ubuntu", "PclinuxOS",
                        "TruvaLinux", "Gelecek Linux"]

for i in gnulinux_dağıtımları:
    liste.insert(END, i)

def yeni():
    global giriş
    pencere2 = Toplevel()
    giriş = ttk.Entry(pencere2)
    giriş.pack()

    btn2 = ttk.Button(pencere2, text="tamam", command=ekle)
    btn2.pack()

def ekle():
    if not giriş.get():
        giriş.insert(END, "Veri Yok!")
    if not "Veri Yok!" in giriş.get():
        liste.insert(END, giriş.get())
        giriş.delete(0, END)

def sil():
    liste.delete(ACTIVE)

btn = ttk.Button(text="ekle", command=yeni)
btn.pack(pady=5)

btn_sil = ttk.Button(text="sil", command=sil)
btn_sil.pack()

pencere.mainloop()
```

Böylece bir pencere aracını daha bitirmiş olduk. Gelelim sıradaki pencere aracımıza...

Menü 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**:

```
from tkinter import *

pencere = Tk()
```

Şimdi, **menümüzü oluşturmaya başlayabiliriz**.

```
menü = Menu(pencere)
```

Burada menü adlı bir **`Menu()` pencere aracı** oluşturduk. Adı '**`menü`**' 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=menü)
dosya = Menu(menü)
```

Burada **`config()`** metodu yardımıyla, öncelikle **`menü`** adlı aracı **pencereye bağlıyoruz**. Parantez içindeki ilk **`menu`** ifadesi, tıpkı öteki pencere araçlarında gördüğümüz **`text`** ifadesi gibi bir seçenektir. Bu seçeneğin değerini '**`menü`**' olarak belirledik.

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 kez aracımızı, bir önceki menünün üzerinde oluşturuyoruz. Aslında bu satır yardımıyla, bir önceki aşamada oluşturduğunuz araç çubuğu üzerinde iner menü (*drop-down menu*) için **bir yer açmış oluyoruz**:

```
menü.add_cascade(label="Dosya", menu=dosya)
```

Şimdi yapmamız gereken menünün **aşağıya doğru açılmasını**, yani **inmesini** sağlamak. Bu iş için yukarıda gördüğünüz **add_cascade()** metodunu kullanıyoruz. Bu metodun '**menü**' 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)

pencere.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:

```
from tkinter import *

pencere = Tk()

menü = Menu(pencere)
pencere.config(menu=menü)
```

```
dosya = Menu(menü)
menü.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)

pencere.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 hâlinde** 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*).

```
from tkinter import *

pencere = Tk()

menü = Menu(pencere)
pencere.config(menu=menü)

dosya = Menu(menü, tearoff=0)
menü.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)

pencere.mainloop()
```

Konu belki biraz karışık gelmiş olabilir, ama aslında hiç de öyle değil.

İşin mantığını anlamak için yukarıdaki kodlarda geçen şu satırlar bize epey yardımcı olabilir:

```
menü = Menu(pencere)
dosya = Menu(menü, tearoff=0)
menü.add_cascade(label=”Dosya”, menu=dosya)
```

Dikkat ettiyseniz, önce **`Menu()`** pencere aracını oluşturuyoruz. Bu araç ilk oluşturulduğunda parantez içi ifadeden anladığımız gibi **`pencere`** adlı **ana pencere üzerine bağlanıyor**.

İkinci satır vasıtasıyla ikinci kez bir **`Menu()`** pencere aracı oluştururken ise **parantez içi** ifadeden anlaşıldığı gibi oluşturduğumuz menüyü, bir önceki **`Menu()`** pencere aracına bağlıyoruz.

Üçüncü satırda inen menüyü oluştururken de bunu bir önceki **`Menu()`** pencere aracı olan **`dosya`**'ya bağlıyoruz.

Menülere başka alt menüler de eklerken bu mantık çok daha kolay anlaşılıyor. **Şöyle ki**:

```
from tkinter import *

pencere = Tk()

menü = Menu(pencere)
pencere.config(menu=menü)

dosya = Menu(menü, tearoff=0)

menü.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")

pencere.mainloop()
```

Gördüğünüz gibi bu kez **`Menu()`** pencere aracımızı ilk olarak '**`dosya`**' adlı araç üzerine bağlıyoruz. Çünkü yeni pencere aracımız, bir önceki pencere aracı olan **`dosya`**'nın üzerinde oluşturulacak.

Bir sonraki satırda **`add_command()`** metodunu kullanırken de alt menüleri **`yeni`** adlı **`Menu()`** pencere aracı üzerine **bağlıyoruz**. Çünkü bu alt menüler **`yeni`**'nin içinde yer alacak.

Aynı şekilde eğer '**`Dosya`**' başlığının **yanına** bir de mesela '**`Düzen`**' diye bir seçenek eklemek istersek kodlarımız arasına şöyle bir bölüm ekliyoruz:

```
from tkinter import *

pencere= Tk()

menü= Menu(pencere)
pencere.config(menu=menü)

dosya= Menu(menü, tearoff=0)

menü.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(menü, tearoff=0)

menü.add_cascade(label="Düzen", menu=dosya2)

dosya2.add_command(label="Bul")

pencere.mainloop()
```

TEXT PENCERE ARACI

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 hâliyle **`Text()`** adlı pencere aracını şu şekilde oluşturabiliriz:

```
from tkinter import *

pencere= Tk()

metin= Text()
metin.pack()

pencere.mainloop()
```

Oluşturduğumuz bu **`Text()`** aracı **pek çok işlevi** yerine getirebilecek durumdadır. Bu araç içine şu hâliyle **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:

```
from tkinter import *

pencere= Tk()

metin = Text(fg = "blue", font=("Helvetica", "13", "bold"))

metin.insert(END, "Sürüm 0.1.1")
metin.pack()

pencere.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()`** komutu da bize öntanımlı bir metin girme imkânı 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:

```
from tkinter import *

pencere= Tk()

a = "Sürüm 0.1.1"

metin= Text(bg="silver", fg="firebrick", font=("Helvetica", "13",
"bold"))

metin.insert(END, a.center(100, "*"))
metin.pack()

pencere.mainloop()
```

Eğer bir metin içinden **herhangi bir bölümü almak isterseniz** kullanmanız gereken şey **`get()`** metodudur. Bu metodun nasıl kullanıldığını görelim şimdi:

```
from tkinter import *

pencere= Tk()

metin= Text()
metin.pack()

metin.get(1.0, END)

pencere.mainloop()
```

Yukarıdaki örneği sadece **`get()`** metodunun **nasıl kullanıldığını göstermek için** verdik. Şu hâliyle bu kod, bizim **beklentilerimizi karşılayamaz**. Çünkü **`get()`** metodu yardımıyla metni aldık ama aldığımız bu metni kullanmamızı sağlayacak araçları henüz penceremize yerleştirmediğimiz için **`get()`** metodunun bize sağladığı işlevi kullanamıyoruz.

Şimdilik burada şuna dikkat edelim: **`metin.get()`** gibi bir ifade kullanırken parantez içinde belirttiğimiz ilk sayı **`1.0`**. Bu sayı metin kutusunun **ilk satırının ilk sütununa** işaret ediyor. Burada ilk satırın **1**'den; ilk sütunun ise **0**'dan başladığına dikkat edelim. **Virgülden sonra gelen `END`** ifadesi ise **`Text()`** aracı içindeki metnin **en sonuna** işaret ediyor. Yani bu koda göre **`get()`** metodu yardımıyla **`Text()`** aracı içindeki bir metni, en başından en sonuna kadar alabiliyoruz. İsterseniz parantez içinde farklı sayılar belirterek alınacak metnin **ilk** ve **son konumlarını** belirleyebiliriz. **Şu koda bir bakalım**:

```
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 hâle getirelim**:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()

def al():
    a = metin.get(1.0, END)
    giriş.insert(0, a)

metin = Text()
metin.pack()

btn = ttk.Button(text="al", command=al)
btn.pack(pady=5)

giriş= ttk.Entry()
giriş.pack(pady=5)

pencere.mainloop()
```

Burada öncelikle **`Text()`** aracı içindeki metnin tamamını alıp (**`metin.get(1.0, END)`**), **`giriş`** adlı **pencere aracına** yerleştiren (**`giriş.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 **`giriş`** 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:

```
from tkinter import *
import tkinter.ttk as ttk

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(100, "*"))
metin.pack()

metin2= Text(height=5, bg="light blue", fg="red")
metin2.pack(pady=5)

btn= ttk.Button(text="al", command=al)
btn.pack(pady=5)

pencere.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! Herhâlde 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:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()

def al():
    a = metin.get("sel.first", "sel.last")
    metin2.insert(END, a)

a = "Sürüm 0.1.1"

metin = Text(height=15, bg="black", fg="white",
             font=("Helvetica", "13", "bold"))

metin.insert(END, a.center(100, "*"))
metin.pack()

metin2= Text(height=5, bg="light blue", fg="red")
metin2.pack(pady=5)

btn= ttk.Button(text="al", command=al)
btn.pack(pady=5)

pencere.mainloop()
```

Bu kodları çalıştırdığımızda, üstteki kutuya yazdığımız metnin **istediğimiz bir kısmını seçtikten sonra** alttaki düğmeye basarsak **seçili kısım ikinci kutuya işlenecektir**.

Scrollbar Pencere Aracı

`Scrollbar()` adlı pencere aracı, Tkinter ile yazdığımız uygulamalara **kaydırma çubuğu ekleme** imkânı veriyor. Bu pencere aracının özelliği, öteki pencere araçlarının aksine **tek başına kullanılmamasıdır**. Kaydırma çubuğu kavramının mantığı gereği, bu pencere aracını **başka pencere araçları ile birlikte kullanacağız**. Mesela **`Text()`** pencere aracılığıyla bir metin kutusu oluşturmuş isek şimdi öğreneceğimiz **`Scrollbar()`** adlı **pencere aracı sayesinde** bu metin kutusuna bir **kaydırma çubuğu ekleyebileceğiz**.

İsterseniz hemen ufak bir **örnek** yapalım:

```
from tkinter import *

pencere = Tk()
```

Buraya kadar olan kısımda bir yenilik yok. Artık ezbere bildiğimiz komutlar bunlar. **Devam ediyoruz**:

```
kaydırma = Scrollbar(pencere)
kaydırma.pack(side=RIGHT, fill=Y)
```

Gördüğünüz gibi **`Scrollbar()`** adlı pencere aracımızı oluşturup bunu '**`pack`**' yöneticisiyle paketledik. **`pack()`** içindeki argümanlara dikkat edin. **`side=RIGHT`** ifadesi yardımıyla kaydırma çubuğumuzu **sağa yaslarken `fill=Y`** ifadesi yardımıyla da **Y** düzlemi üzerinde (*yani yukarıdan aşağıya doğru*) boylu boyunca uzatıyoruz. **Devam edelim**:

```
metin = Text(yscrollcommand=kaydırma.set)
metin.pack()
```

Kaydırma çubuğunu bir **metin kutusu içinde kullanacağımız için `Text()`** pencere aracı vasıtasıyla **metin kutumuzu tanımlıyoruz**. Burada **`yscrollcommand=kaydırma.set`** gibi bir ifade kullandığımıza dikkat edin. Bu ifade ile **kaydırma işlemini etkinleştiriyoruz**.

Şimdi **son darbeyi** vurabiliriz:

```
kaydırma.config(command=metin.yview)
pencere.mainloop()
```

Burada **`kaydırma.config()`** satırını, **metin aracını tanımladıktan sonra yazmamız önemli**. Eğer metin aracını tanımlamadan **`config`** satırını yazarsak komut satırında hataların uçuştuğunu görürüz.

Dilerseniz yukarıda parça parça verdiğimiz uygulamayı bir de **topluca görelim**:

```
from tkinter import *

pencere = Tk()

kaydırma = Scrollbar(pencere)
kaydırma.pack(side=RIGHT, fill=Y)

metin = Text(yscrollcommand=kaydırma.set)
metin.pack()

kaydırma.config(command=metin.yview)

pencere.mainloop()
```

Böylece metin kutusuna yazdığımız yazılar **aşağıya doğru uzadıkça sağ tarafta bir kaydırma çubuğu görüntülenecektir**.

Peki, kaydırma çubuğunun, ekranın alt tarafında görünmesini istersek ne yapacağız? Yani yukarıdaki örnekte yaptığımız gibi aşağıya doğru giden metinleri değil de *yana doğru giden metinleri kaydırmak istersek ne yapmalıyız?* Çok kolay. **Adım adım gidelim yine**:

```
from tkinter import *

pencere = Tk()
```

Devam ediyoruz...

```
kaydırma = Scrollbar(pencere, orient=HORIZONTAL)
kaydırma.pack(side=BOTTOM, fill=X)
```

Gördüğünüz gibi kaydırma çubuğunun **pozisyonunu belirlemek için** yeni bir seçenekten yararlanıyoruz. Bu yeni seçeneğin adı **`orient`**. Buna vermemiz gereken değer ise **`HORIZONTAL`**. Bu İngilizce kelime Türkçede **yatay** anlamına geliyor. Kaydırma çubuğumuz ekranda **düşey değil**, **yatay vaziyette görüneceği için bu seçeneği kullandık**. Ayrıca **`pack`** parametrelerine de dikkat edin. **`side`** seçeneğine **`BOTTOM`** değerini verdik. Çünkü kaydırma çubuğumuzun ekranın alt tarafında görünmesini istiyoruz. Bir önceki örnekte, çubuğun ekranın sağında görünmesini istediğimiz için **`side=RIGHT`** şeklinde bir kullanım benimsemiştik (**`RIGHT`** *kelimesi sağ anlamına gelirken* **`BOTTOM`** *kelimesi dip-alt gibi anlamlara gelmektedir*). Bu arada **`fill`** seçeneğinin değerinin de bir önceki örneğin aksine, **`X`** olduğuna dikkat edin. Çünkü bu defa **kaydırma çubuğumuzun x düzlemi üzerinde** (*yani soldan sağa doğru*) **boylu boyunca uzanması gerekiyor**.

Devam ediyoruz:

```
metin = Text(wrap=NONE, xscrollcommand=kaydırma.set)
metin.pack()
```

Şimdi de **metin kutumuzu tanımladık**. Buradaki parametrelere dikkat edin. Öncelikle metin kutumuza **`wrap`** adlı bir seçenek tanımlıyoruz. Bu **`wrap`** seçeneği metin kutusuna yazılan yazıların, **ekranın sağ sınırına ulaşınca bir satır kaydırılıp kaydırılmayacağını belirliyor**. Varsayılan olarak yazılan metin ekranın sağ sınırına dayandığında **otomatik olarak alt satıra geçecektir**. Ama eğer biz bu **`wrap`** seçeneğine **`NONE`** değerini verirsek metin sınırı geçip sağa doğru uzamaya devam edecektir. İşte aynı parantez içinde belirttiğimiz **`xscrollcommand=kaydırma.set`** komutu sayesinde **sağa doğru uzayan bu metni kaydırabileceğiz**. Burada, yukarıdaki örnekten farklı olarak **`xscrollcommand`** ifadesini kullandığımıza dikkat edin. Bir önceki örnekte **`yscrollcommand`** ifadesini kullanmıştık. **Bu farkın nedeni**; bir önceki örnekte y düzlemi üzerinde çalışırken şimdiki örnekte x düzlemi üzerinde çalışıyor olmamız...

Artık **son darbeyi** vurabiliriz:

```
kaydırma.config(command=metin.xview)
pencere.mainloop()
```

Gördüğünüz gibi burada da bir önceki örnekten farklı olarak **xview** ifadesini kullandık. Bir önceki örnekte **yview** ifadesini kullanmıştık. Bu farkın nedenini söylemeye gerek yok...

Gelin, isterseniz bu parçalı örneği bir araya getirelim:

```
from tkinter import *

pencere = Tk()

kaydırma = Scrollbar(pencere, orient=HORIZONTAL)
kaydırma.pack(side=BOTTOM, fill=X)

metin = Text(wrap=NONE, xscrollcommand=kaydırma.set)
metin.pack()

kaydırma.config(command=metin.xview)

pencere.mainloop()
```

Burada da yazılan metin, **sağa doğru uzayarak pencerenin sınırlarını aştığında altta bir kaydırma çubuğu belirecektir**.

TKINTER UYGULAMALARINI GÜZELLEŞTİRMEK

5

Tkinter arayüz takımı konusunda en fazla şikayet edilen şey, **Tkinter ile yazılan uygulamaların çirkin görünmesidir**. Bu bölümde, **bu sıkıntıyı aşmanın bazı yollarını** göstermeye çalışacağız. Ayrıca bu bölümde, *güzellik ile ilgili olduğunu düşündüğüm* başka konulara da değineceğiz. Mesela bu bölümde, Tkinter uygulamaları içinde kullandığımız pencere araçlarına, *nasıl simge yerleştirebileceğimizi de* inceleyeceğiz. Bu bölüm bir yandan da **örnek uygulamalarla** zenginleştirilmeye çalışılacaktır. Yani konuları anlatırken ufak program örnekleri üzerinden gitmeye çalışacağız.

Tkinter Programlarının Renk Şemasını Değiştirmek

Tkinter programlarını güzelleştirme yolunda ilk inceleyeceğimiz konu, yazdığımız uygulamaların **renk şemasını değiştirmek** olacaktır.

Biz herhangi bir Tkinter uygulaması yazdığımız zaman, bu uygulamanın **varsayılan bir renk şeması olacaktır**.

Hemen bir örnekle başlayalım:

```
from tkinter import *

pencere = Tk()

etiket = Label(text="Örnek Uygulama")
etiket.pack()

pencere.mainloop()
```

Buna benzer bütün uygulamalarımızda belli bir renk şeması, **varsayılan olarak** pencerelerimize uygulanacaktır. Eğer biz *müdahale etmezsek* oluşturacağımız bütün pencereler ve üzerindeki pencere araçları **tek bir renkte** görünecektir.

Ama eğer istersek **bu gidişe bir dur diyebiliriz**. Bunu nasıl yapabileceğimizi daha önceki derslerimizde de kısmen görmüştük.

Mesela şu seçenekler bize yardımcı oluyordu:

```
from tkinter import *

pencere = Tk()

etiket = Label(text="Örnek Uygulama", fg="white", bg="blue")
etiket.pack()

pencere.mainloop()
```

Burada, kullandığımız **`Label()`** adlı pencere aracını **`bg`** ve **`fg`** seçenekleri yardımıyla farklı renklerde çizdik. Siz renk adları yerine *https://www.html-color-codes.info/* adresine başvurarak renk kodlarını da kullanabileceğinizi biliyorsunuz.

Aslında benim kastettiğim **güzelleştirme operasyonu**, sadece pencere araçlarının rengini değiştirmekle ilgili değil. Aynı zamanda, **penceremizin rengini değiştirmekten de bahsediyorum**. İsterseniz hemen bir örnek vererek ne demek istediğimizi biraz daha net olarak anlatmaya çalışayım.

Şu örneğe bir bakın:

```
from tkinter import *

pencere = Tk()

pencere.tk_setPalette("#D0A9F5")

etiket = Label(text="Renk şeması değiştirilmiş bir örnek uygulama")
etiket.pack()

pencere.mainloop()
```

Gördüğünüz gibi sadece tek bir pencere aracının rengini değil, aynı zamanda bütün bir pencerenin **rengini değiştirdik**. Bunu yapmamızı sağlayan şey de kodlar içinde kullandığımız, **`pencere.tk_setPalette("#D0A9F5")`** satırı oldu. Yani buradaki **`tk_setPalette()`** adlı metodu kullanarak **bütün pencerenin renk şemasını değiştirebildik**. Bu **`tk_setPalette()`** komutu, **`Tk()`** sınıfının metotlarından yalnızca biridir. Bu sınıfın bize hangi metotları sunduğunu görmek için Python'ın **etkileşimli kabuğunda** şu komutları verebilirsiniz:

```
from tkinter import *
dir(Tk())
```

Bu komutun çıktısı olarak elde edilen liste **alfabe sırasına göre dizilmiştir**. Dolayısıyla bu listede **`tk_setPalette()`** metodunu rahatlıkla bulabilirsiniz.

Gördüğünüz gibi **`Tk()`** sınıfının bir yığın metodu var. İşte, **`tk_setPalette()`** metodu da bunlardan biri olup Tkinter pencerelerinin **renk şemasını değiştirmemizi sağlıyor**.

Şimdi, **şu örneğe bakalım**:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()
pencere.tk_setPalette("#F7F2E0")

etiket = Label(text="Hangi GNU/Linux Dağıtımını Kullanıyorsunuz?")
etiket.pack(pady=10, padx=10)

liste = Listbox()
liste.pack(side=LEFT, pady=5)

GNULinux = ["Debian", "Ubuntu", "Kubuntu", "RedHat"]

for i in GNULinux:
    liste.insert(END, i)

düğme = ttk.Button(text="KAPAT", command=pencere.quit)
düğme.pack()

pencere.mainloop()
```

Gördüğünüz gibi **tk_setPalette()** metodu ile pencerenin r**enk şemasını değiştirdiğimizde**, **bundan pencere ile birlikte üzerindeki bütün pencere araçları da etkileniyor**. *Peki, biz pencere araçlarının pencereden farklı bir renkte olmasını istersek ne yapacağız?* **Çok basit**:

```
from tkinter import *

pencere = Tk()
pencere.tk_setPalette("#F7F2E0")

etiket = Label(text="Hangi GNU/Linux Dağıtımını Kullanıyorsunuz?",
               fg="#B4045F")
etiket.pack(pady=10, padx=10)

liste = Listbox()
liste.pack(side=LEFT, pady=5)

GNULinux = ["Debian", "Ubuntu", "Kubuntu", "RedHat"]

for i in GNULinux:
    liste.insert(END, i)

düğme = Button(text="KAPAT", bg="#000000",
               fg="#b7f731", relief="flat",
               width=10, command=pencere.quit)
düğme.pack()

pencere.mainloop()
```

Gördüğünüz gibi **farklı renkte olmasını istediğimiz pencere araçlarının renklerini**, daha önce öğrendiğimiz **seçenekler yardımıyla** açık açık belirtiyoruz. Tabii sizin benden daha zevkli renk kombinasyonları seçeceğinizi tahmin ediyorum.

Böylelikle bir Tkinter programının renk şemasını nasıl değiştireceğimizi öğrenmiş olduk. Bir sonraki bölümde Tkinter uygulamaları üzerinde **resimleri**/**simgeleri nasıl kullanabileceğimizi** göreceğiz. Neticede resimler ve simgeler de bir programı güzelleştiren ögelerdir.

Pencere Araçlarına Simge Eklemek

Şimdi, Tkinter'de yazdığımız programlara *nasıl simge ekleyebileceğimizi göreceğiz*. Bu iş için **`PIL`** adlı modülü kullanacağız. **Bu modülü kullanmak için `pillow` adlı haricî bir paket indirmemiz gerekecek**. Bu paketi, şu komutla bilgisayarımıza kurabiliriz:

```
pip install Pillow
```

Gelelim asıl konumuza. Öncelikle aşağıdaki gibi bir **başlangıç** yapalım:

```
from tkinter import *
from PIL import ImageTk
```

Böylelikle gerekli bütün modülleri içe aktarmış olduk. Burada **`PIL`** adlı modülü de **içe aktardığımıza dikkat edin**. Devam ediyoruz:

```
pencere = Tk()
simge = ImageTk.PhotoImage(file="simge.jpg")
```

Yukarıdaki kodlar yardımıyla her zamanki gibi önce bir pencere oluşturduk. Ardından da programın en başında içe **`PIL`** modülünden aktardığımız **`ImageTk`** adlı sınıfın **`PhotoImage()`** adlı metodundan ve **`file`** adlı seçeneğinden yararlanarak bilgisayarımızdaki '**`simge.jpg`**' adlı dosyayı programımız içinde kullanılabilir hâle getirdik. Dikkat edin, "**kullandık**" diyemiyoruz; "**kullanılabilir hâle getirdik**" diyoruz.

Yani kabaca ifade etmek gerekirse bu şekilde bir **`resim`** **nesnesi** oluşturmuş oluyoruz.

Devam edelim:

```
düğme = Button(text="Resim1", image=simge, compound="top")
düğme.pack()
```

Burada bildiğimiz şekilde, **`Button()`** pencere aracını kullanarak bir düğme oluşturduk. **Ama burada bilmediğimiz birkaç şey var**.

Hemen anlatalım: Öncelikle kodlarımız arasında, **`image`** adlı **yeni bir seçenek** görüyoruz. Buraya, yukarıda kullanılabilir hâle getirdiğimiz **`resim`** nesnesini yerleştiriyoruz. Bu seçeneğin ardından, **`compound`** adlı başka bir seçenek daha geliyor. Bu da **`text`** ve **`image`** seçenekleri ile tanımlanan ögelerin pencerede nasıl görüneceğini belirliyor. Burada kullandığımız '**`top`**' değeri, **`image`** seçeneğiyle gösterilen ögenin **`text`** seçeneği ile gösterilen ögenin **üstünde** yer alacağını ifade ediyor.

Bu **`compound`** seçeneği **dört farklı değer** alabilir:

1. **top:** resim metnin *üstünde*
2. **bottom:** resim metnin *altında*
3. **right:** resim metnin *sağında*
4. **left:** resim metnin *solunda*

Biz örneğimizde '**`top`**' değerini tercih ettik. Tabii ki siz bu dört seçenek içinden istediğinizi kullanabilirsiniz. Yukarıdaki kodlarda görünen **`pack()`** ve **`mainloop()`** artık adımız soyadımız gibi bildiğimiz şeyler. O yüzden üzerinde durmuyoruz.

Şimdi, isterseniz bu kodları **topluca görelim**:

```
from tkinter import *
from PIL import ImageTk

pencere = Tk()

simge = ImageTk.PhotoImage(file="simge.jpg")

düğme = Button(text="Resim1", image=simge, compound="top")
düğme.pack()

pencere.mainloop()
```

Böylece Tkinter'de **resimleri** ve **metinleri bir arada nasıl kullanacağımızı öğrenmiş olduk**.

Şimdi gelin, isterseniz yukarıdaki bilgilerimizi kullanarak **örnek bir arayüz** yazalım:

```
from tkinter import *
from PIL import ImageTk

pencere = Tk()
pencere.tk_setPalette("#CECEF6")

çerçeve1 = Frame()
çerçeve1.grid(row=0, column=0, pady=5, padx=5)

çerçeve2 = Frame()
çerçeve2.grid(row=1, column=0, pady=10, padx=10)

simge1 = ImageTk.PhotoImage(file="simge1.jpg")
simge2 = ImageTk.PhotoImage(file="simge2.jpg")
simge3 = ImageTk.PhotoImage(file="simge3.jpg")
simge4 = ImageTk.PhotoImage(file="simge4.jpg")

etiket = Label(çerçeve1,
               text="Kendinize Uygun bir Simge Seçiniz...",
               fg="#610B0B",
               font="Verdana 10 bold")
etiket.pack()

düğme1 = Button(çerçeve2, text="Resim1", image=simge1,
compound="top")
düğme1.grid(row=3, column=0, padx=6)

düğme2 = Button(çerçeve2, text="Resim2", image=simge2,
compound="top")
düğme2.grid(row=3, column=3, padx=6)

düğme3 = Button(çerçeve2, text="Resim3", image=simge3,
compound="top")
düğme3.grid(row=3, column=6, padx=6)

dugme4 = Button(çerçeve2, text="Resim4", image=simge4,
compound="top")
dugme4.grid(row=3, column=9, padx=6)

pencere.mainloop()
```

Mesela bu programı *https://www.iconarchive.com/show/face-avatars-icons-by-hopstarter.html* adresindeki simgelerden birkaçını kullanarak çalıştırmayı deneyebilirsiniz.

Tabii bu programın **düzgün çalışması için resimlerinizle program dosyasını aynı klasöre koymayı unutmuyorsunuz**. Eğer resimlerle program dosyası **ayrı klasörlerde duracaksa** simge ögelerini belirlerken resimlerin bulunduğu klasör adlarını tam olarak yazmaya dikkat etmelisiniz.

Burada düğmeleri konumlandırmak için `pack()` geometri yöneticisi yerine `grid()` adlı geometri yöneticisini kullandığımıza dikkat edin. **Ögeleri bu şekilde sıralamak için** `grid()` **yöneticisi daha uygundur**. Ayrıca yukarıdaki kodlarda iki farklı `Frame()` oluşturduğumuza da dikkat edin. `Label()` adlı pencere aracımızı bir `Frame()` üzerine, geri kalan pencere araçlarımızı (*yani düğmelerimizi*) ise öteki `Frame()` üzerine yerleştirdik. **İki farklı** `Frame()` **kullanmamız sayesinde** aynı program içinde hem `grid()` hem de `pack()` adlı geometri yöneticilerini **bir arada kullanma imkânı elde ettik**.

İki farklı `Frame()` kullanmamız aynı zamanda bize, pencere araçlarımızı daha **özgür bir şekilde konumlandırma olanağı** da sağladı. Ayrıca `grid()` yöneticisinin `padx` seçeneğini kullanarak pencere araçlarının **birbirlerine olan uzaklıklarını da ayarlayabildik**.

Böylelikle bu konuyu da bitirmiş olduk. Bir sonraki bölümde Tkinter ile yazdığımız pencere araçlarına nasıl **ipucu metni** (*tooltip*) ekleyeceğimizi göreceğiz.

Pencere Araçlarına İpucu Metni (Tooltip) Eklemek

Fareyi **pencere araçlarının üzerine getirdiğimizde ekranda beliren** ve **o aracın ne işe yaradığını kısaca açıklayan ufak metin parçalarına** *ipucu metni* diyoruz. İşte bu bölümde Tkinter'deki pencere araçlarına **nasıl ipucu metni ekleyeceğimizi** öğreneceğiz. **Öncelikle şunu söyleyelim**: Tkinter böyle bir özelliği bize kendiliğinden sunmuyor. Ama hem Python'ın hem de Tkinter'in en güçlü yanlarından biri, dilde olmayan bir özelliği **üçüncü parti yazılımlar aracılığıyla** dile ekleyebilmemizdir.

Pencere araçlarına ipucu metni ekleyebilmek için bu üçüncü parti yazılımlardan birini kullanacağız. Şu noktada yapmamız gereken ilk iş, *https://www.dropbox.com/s/85telj22lqgkfz3/tooltip.py?dl=1* adresinden **`tooltip`** adlı modülü **bilgisayarımıza indirmek** olacaktır. Bu modül, pencere araçlarına **ipucu metni** eklememizi sağlayacak. Bu modülü, içinde ipucu metni kullanacağımız programlarımızla **aynı klasör içinde** tutmamız gerekiyor. Örneğin diyelim ki '**falanca.py**' adında bir program yazdık. Bu programın **FALANCA** adlı klasör içinde yer aldığını varsayarsak **`tooltip`** modülünü **falanca.py** ile aynı klasör içine, yani FALANCA adlı klasöre atmamız gerekiyor. Bu sayede **`tooltip`** modülünü programımız içine aktarırken (*import*) herhangi bir sorun yaşamayacağız. Bununla ilgili ufak bir **örnek** yapalım:

```
from Tkinter import *
import tooltip
```

Böylece **`tkinter`** modülüyle birlikte **`tooltip`** modülünü de **içe aktarmış olduk**.

Python, biz bu modülü içe aktarmaya çalıştığımızda öncelikle programımızın içinde bulunduğu dizini kontrol ederek, **tooltip.py** adlı bir dosya olup olmadığına bakacaktır. Eğer modül bu dizinde **bulunamaz ise** Python **`sys.path`** çıktısında görünen dizinlerin **içini de denetleyecektir**. Eğer modül orada da yoksa Python bize bir **hata mesajı** gösterecektir. Bu durumda, **`tooltip`** gibi **tek dosyalık haricî bir modülü**, **programın bulunduğu dizin içine atmak en kolay yol olacaktır**. Neyse, biz örneğimize devam edelim:

```
pencere = Tk()

etiket = Label(text="Herhangi bir etiket")
etiket.pack()

düğme = Button(text="Tamam", command=pencere.destroy)
düğme.pack()
```

Burada, normal bir şekilde penceremizi tanımladık. Bunun yanında bir adet **etiket** ve bir adet **düğme** oluşturduk. Yeni bir şey yok.

Devam edelim:

```
tooltip.Tooltip(düğme, "Programı kapat")
pencere.mainloop()
```

İşte **tooltip** modülü burada devreye giriyor. Bu modülün **ToolTip()** adlı sınıfını kullanarak düğme adlı pencere aracının üzerine **fare ile geldiğimizde**, **ekranda görünmesini istediğimiz ipucu metnini oluşturuyoruz**. Burada ipucu metnimiz şu: "**Programı kapat**"

Programımızı **bütünüyle** bir görelim bakalım:

```
from tkinter import *
import tooltip

pencere = Tk()

etiket = Label(text="Herhangi bir etiket")
etiket.pack()

düğme = Button(text="Tamam", command=pencere.destroy)
düğme.pack()

tooltip.ToolTip(düğme, "Programı kapat")

pencere.mainloop()
```

Gördüğünüz gibi düğmenin üzerine **fare ile geldiğimizde ipucu metnimiz ekranda görünüyor**. İşte, Tkinter'de ipucu metinlerini kullanmak bu kadar **kolaydır**. Kendi yazdığınız programlarda bu modülü kullanarak programınızı kullanıcılar açısından daha cazip ve anlaşılır bir hâle getirebilirsiniz. Gelin, isterseniz bununla ilgili bir **örnek** daha yapalım:

```
from tkinter import *
import tooltip
import time

pencere = Tk()
```

```
etiket = Label(text="Saat 17:00'da randevunuz var!")
etiket.pack(expand=YES, fill=BOTH)

def saat():
    şuan = time.strftime("%H:%M:%S")
    return f"Şu anda saat {şuan}"

düğme = Button(text="Tamam", command=pencere.destroy)
düğme.pack(expand=YES)

tooltip.ToolTip(pencere, saat())

pencere.mainloop()
```

Burada, ipucu metnimizi **doğrudan pencerenin kendisine atadık**. Böylece fare ile pencere üzerine geldiğimizde **o an saatin kaç olduğu ekranda görünecektir**.

6 NASIL YAPILIR?

Tkinter'de Fare ve Klavye Hareketleri

Tkinter'de yazdığımız bir uygulamada; kullanıcının **fare hareketlerini** ve klavyede **bastığı tuşları belli fonksiyonlara bağlamak** ve **bu hareketleri yakalamak** bu bölümün konusunu oluşturuyor. Mesela kullanıcı klavyedeki **ENTER** tuşuna bastığında programımızın özel bir işlemi yerine getirmesini sağlamak, kullanıcının hangi tuşa bastığını bulmak bu konunun kapsamına giriyor. Her zamanki gibi lafı hiç uzatmadan bir örnekle derdimizi anlatmaya çalışalım. **Diyelim ki şöyle bir şey var elimizde**:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()

etiket = Label(text="Aşağıdaki kutucuğa adınızı yazıp \
Tamam düğmesine basın!\n")
etiket.pack(padx=10)

giriş = ttk.Entry()
giriş.pack()

def kaydet():
    dosya = open("isimler_python.txt", "a", encoding="utf-8")
```

```
    veri = giriş.get()

    dosya.write(veri + "\n")
    dosya.close()

    etiket["text"] = etiket["text"] + f"{giriş.get()} dosyaya
başarıyla eklendi\n"
    giriş.delete(0, END)

düğme = ttk.Button(text="Tamam", command=kaydet)
düğme.pack(pady=5)

pencere.mainloop()
```

Burada kullanıcı **Tamam** düğmesine bastığında **`kaydet()` fonksiyonu içinde tanımlanan işlemler gerçekleştirilecektir**. Peki, biz aynı işlemin **`ENTER`** düğmesine basılarak da yapılabilmesini istersek ne olacak? *Yani kullanıcının mutlaka Tamam düğmesine basmasını zorunlu kılmadan klavyedeki ENTER tuşuna bastığında da kutucuğa yazılan isimlerin dosyaya eklenmesini nasıl sağlayacağız?* İşte bunu yapabilmek için **`bind()`** fonksiyonundan **yararlanmamız** gerekiyor. Bu fonksiyon bize klavye ve fare hareketlerini **belli fonksiyonlara bağlama** ve **bu hareketleri yakalama** imkânı verecek. *Nasıl mı?*

Hemen görelim:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()

etiket = Label(text="Aşağıdaki kutucuğa adınızı yazıp \
Tamam düğmesine basın!\n")
etiket.pack(padx=10)

giriş = ttk.Entry()
giriş.pack()

def kaydet(event=None):
    dosya = open("isimler_python.txt", "a", encoding="utf-8")
```

```
    veri = giriş.get()

    dosya.write(veri + "\n")
    dosya.close()

    etiket["text"] = etiket["text"] + f"{giriş.get()} dosyaya
başarıyla eklendi\n"
    giriş.delete(0, END)

düğme = ttk.Button(text="Tamam", command=kaydet)
düğme.pack(pady=5)

pencere.bind("<Return>", kaydet)

pencere.mainloop()
```

Burada, önceki kodlarımıza yalnızca şu eklemeleri yaptık:

```
def kaydet(event=None):
....
....

pencere.bind("<Return>", kaydet)
...
```

`kaydet()` fonksiyonuna eklediğimiz **`event`** parametresi **zorunlu bir adlandırma değildir**. Hatta oraya kendi isminizi dahi yazdığınızda programın düzgün çalıştığını göreceksiniz. Ancak oraya bir parametre yazacağınız zaman herhangi bir kelime yerine '**`event`**' **ifadesini kullanmanızı tavsiye ederim**. Çünkü **`event`** tıpkı sınıflardaki '**`self`**' gibi kemikleşmiştir.

Burada dikkat etmemiz gereken iki şey var: Birincisi, o fonksiyonu parametresiz bırakmamak ikincisi, parametreye varsayılan değer olarak **`None`** vermek. Eğer bu **`None`** değerini vermezseniz kullanıcı **`ENTER`** tuşuna bastığında program çalışır, ama **Tamam düğmesine bastığında çalışmaz**. Çünkü Tkinter, **`event`** parametresinin değerini **`command=kaydet`** şeklinde gösterdiğimiz koda **otomatik olarak atamayacaktır**. Bu yüzden, oraya varsayılan değer olarak **`None`** yazmazsak **`kaydet()`** fonksiyonunun **bir adet argüman alması gerektiğine dair bir hata alırız**.

Kodlarımıza yaptığımız ikinci ekleme, **`bind()`** fonksiyonudur. Burada dikkat etmemiz gereken birkaç şey var. Öncelikle bu **`bind()`** fonksiyonunu pencere ile birlikte kullandığımıza dikkat edin. **Bu satırla aslında Tkinter'e şu komutu vermiş oluyoruz**:

"Ey Tkinter! '**`pencere`**' seçili iken, yani etkinken kullanıcı **`RETURN`** düğmesine (*veya başka bir söyleyişle ENTER düğmesine*) bastığında **`kaydet()`** adlı fonksiyonu çalıştır!"

Artık kullanıcımız, kutucuğa bir isim yazıp **`ENTER`** düğmesine bastığında **yazdığı isim dosyaya eklenecektir**.

Gördüğünüz gibi eğer klavyedeki **`ENTER`** düğmesini bağlamak istiyorsak kullanmamız gereken ifade **`<Return>`**. *Peki, başka hangi ifadeler var?* Mesela **`ESCAPE`** için bir örnek verelim. Yukarıdaki kodlara ufacık bir ekleme yapıyoruz:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()

etiket = Label(text="Aşağıdaki kutucuğa adınızı yazıp \
Tamam düğmesine basın!\n")
etiket.pack(padx=10)

giriş = ttk.Entry()
giriş.pack()

def kaydet(event=None):
    dosya = open("isimler_python.txt", "a", encoding="utf-8")

    veri = giriş.get()

    dosya.write(veri + "\n")
    dosya.close()

    etiket["text"] = etiket["text"] + f"{giriş.get()} dosyaya
başarıyla eklendi\n"
    giriş.delete(0, END)
```

```
düğme = ttk.Button(text="Tamam", command=kaydet)
düğme.pack(pady=5)

pencere.bind("<Return>", kaydet)
pencere.bind("<Escape>", sys.exit)

pencere.mainloop()
```

Gördüğünüz gibi klavyedeki **ESCAPE** düğmesini kullanabilmek için kodlarımız içine **<Escape> ifadesini eklememiz yeterli oluyor**. *Peki, klavyedeki diğer düğmelerin karşılıkları nelerdir?*

Listeleyelim:

- **F1:** <F1> (*Bütün F'li tuşlar aynı şekilde kullanılabilir*)
- **Num Lock:** <Num_Lock>
- **Scroll Lock:** <Scroll_Lock>
- **Backspace:** <BackSpace>
- **Delete:** <Delete>
- **Sol Ok:** <Left>
- **Sağ Ok:** <Right>
- **Aşağı Ok:** <Down>
- **Yukarı Ok:** <Up>
- **Alt Düğmesi:** <Alt_L>
- **Ctrl Düğmesi:** "<Control_L>
- **Shift Düğmesi:** <Shift_L>
- **Ctrl+S:** <Control-s> (*Öteki harfler de aynı şekilde kullanılabilir*)
- **Shift+S:** <Shift-s> (*Öteki harfler de aynı şekilde kullanılabilir*)
- **Alt+S:** <Alt-s> (*Öteki harfler de aynı şekilde kullanılabilir*)
- **Boşluk Düğmesi:** <space>
- **Print Düğmesi:** <Print>

Klavyedeki harfleri **pencere araçlarına bağlamak için** ise **doğrudan o harfin adını kullanabiliriz**. Mesela bir işlemin, **f** harfine basıldığında gerçekleşmesini istersek yazmamız **gereken ifade f olacaktır**. Bunun **<f>** değil, sadece

f şeklinde olduğuna dikkat edin. Tabii burada **Türkçeye özgü** harfleri kullanmamaya dikkat ediyoruz. Mesela "**q**" harfine basıldığında **programdan çıkılmasını istiyorsanız** şu kodu yazacaksınız:

```
pencere.bind("q", sys.exit)
```

Klavye hareketlerini bağlamayı öğrendik. *Peki, ya fare hareketlerini nasıl bağlayacağız?* O da çok basit. Burada da yine **bind()** fonksiyonundan yararlanacağız.

Gelin, isterseniz şöyle bir uygulama yazalım:

```
from tkinter import *

pencere = Tk()
pencere.geometry("400x200")

etiket = Label(text="Farenin pencere üzerindeki konumu:")
etiket.pack(anchor=NW)

girişx_etiket = Label(text="X:")
girişx_etiket.pack(side=LEFT, anchor=N)

girişx = Entry(width=5)
girişx.pack(side=LEFT,padx=15, anchor=N)

girişy_etiket = Label(text="Y:")
girişy_etiket.pack(side=LEFT, anchor=N)

girişy = Entry(width=5)
girişy.pack(side=LEFT, anchor=N)

def hareket(event=None):
    girişx.delete(0, END)
    girişy.delete(0, END)
    girişx.insert(0, event.x)
    girişy.insert(0, event.y)

pencere.bind("<Button-1>", hareket)
pencere.mainloop()
```

Yukarıdaki uygulamayı çalıştırıp pencere üzerinde herhangi bir yere **farenin sol tuşu** ile tıkladığımızda, tıkladığımız **noktanın koordinatları** (*x ve y düzlemine göre*) **kutucuklar içinde görünecektir**.

Bu kodları biraz açıklayalım:

Kodlar arasında gördüğünüz `anchor` ifadesi, `pack()` adlı pencere yöneticisinin seçeneklerinden biridir. Bu seçenek, ilgili pencere aracının pencere üzerinde **bir noktaya çapa atmasını sağlar** (*İngilizce anchor kelimesi Türkçede çapa atmak, demir atmak gibi anlamlara gelir*). **Bu seçenek dört farklı değer alabilir**:

- **N:** Kuzey anlamına gelen **North** kelimesinin **kısaltmasıdır**. Pencere aracının kuzey yönüne sabitlenmesini sağlar.
- **S:** Güney anlamına gelen **South** kelimesinin **kısaltmasıdır**. Pencere aracının güney yönüne sabitlenmesini sağlar.
- **W:** Batı anlamına gelen **West** kelimesinin **kısaltmasıdır**. Pencere aracının batı yönüne sabitlenmesini sağlar.
- **E:** Doğu anlamına gelen **East** kelimesinin **kısaltmasıdır**. Pencere aracının doğu yönüne sabitlenmesini sağlar.

Bu yönleri bir arada da kullanabiliriz. Örneğin pencere aracının **kuzeybatı yönünde sabitlenmesini istersek** kullanmamız gereken ifade **NW** olacaktır. Mesela yukarıdaki kodlarda bunu kullandık. `etiket.pack(anchor=NW)` ifadesi yardımıyla, `etiket` adlı pencere aracımızın kuzeybatı yönüne çapa atmasını sağladık (*Yani pencerenin üst-sol ucuna*). Öteki pencere araçlarında sadece `anchor=N` ifadesini kullandık. Çünkü bu araçlardaki `side=LEFT` ifadesi, aracımızın **sol yana yerleşmesini** zaten sağlıyor. `pack()` yöneticisi içinde kullandığımız `padx` seçeneğini zaten biliyorsunuz. **Bu seçenek yardımıyla pencere araçlarının dirsek mesafesini ayarlıyoruz...**

Yukarıdaki kodlar içinde en önemli nokta tabii ki `hareket()` adlı fonksiyon. Burada, **parantez içinde** `event` **parametresini belirtmeyi unutmuyoruz**. Kabaca ifade etmek gerekirse bu parametre bizim örneğimizde **fare hareketlerini temsil ediyor**. Buna göre fonksiyon içindeki `event.x` ve `event.y` ifadelerini, farenin **x düzlemi üzerindeki hareketi** ve **farenin y düzlemi üzerindeki hareketi** şeklinde düşünebiliriz. Bu arada, `giriş.delete()` fonksiyonları yardımıyla, `Entry()` **aracının içini sürekli olarak boşalttığımıza**

dikkat edin. Eğer giriş.`delete()` **fonksiyonlarını yazmazsak fare tıklaması ile bulduğumuz eski ve yeni koordinatlar birbirine karışacaktır**.

`pencere.mainloop()` satırından hemen önce, programımızdaki asıl işi yapan `bind()` fonksiyonunu görüyoruz. Farenin sol düğmesini `Button-1` ifadesinin temsil ettiğini görüyorsunuz.

Peki, buna benzer başka neler var?

- **Button-2:** Farenin orta düğmesine (*veya tekerine*) basılması
- **Button-3:** Farenin sağ düğmesine basılması
- **Motion:** Farenin, hiçbir düğme basılı değilken hareket ettirilmesi
- **B1-Motion:** Farenin, sol düğme basılı hâlde hareket ettirilmesi
- **B2-Motion:** Farenin, orta düğme (*veya teker*) basılı hâlde hareket ettirilmesi
- **B3-Motion:** Farenin, sağ düğme basılı hâlde hareket ettirilmesi
- **ButtonRelease-1:** Farenin sol düğmesinin serbest bırakılması (*yani düğmeye basıldığındaki değil, düğmenin bırakıldığındaki hâli...*)
- **ButtonRelease-2:** Farenin orta düğmesinin (*veya tekerinin*) serbest bırakılması
- **ButtonRelease-3:** Farenin sağ düğmesinin serbest bırakılması
- **Double-Button-1:** Farenin sol düğmesine çift tıklanması
- **Double-Button-2:** Farenin orta düğmesine (*veya tekerine*) çift tıklanması
- **Double-Button-3:** Farenin sağ düğmesine çift tıklanması
- **Enter:** Farenin pencere aracının üzerine gelmesi (*Bunun ENTER tuşuna basmak anlamına gelmediğine dikkat edin.*)
- **Leave:** Farenin pencere aracını terk etmesi

Sırada **klavye hareketlerini yakalamak var**. Yani şimdiki görevimiz, bir kullanıcının klavyede hangi tuşlara bastığını bulmak. Bunun için `keysym` **niteliğinden yararlanacağız**:

```
from tkinter import *

pencere = Tk()
pencere.geometry("200x200")
```

```
etiket = Label(text="Basılan Tuş:\n", wraplength=150)
etiket.pack()

def göster(event=None):
    etiket["text"] += event.keysym

pencere.bind("<Key>", göster)

pencere.mainloop()
```

Yine kodlarımızı biraz açıklayalım.

Öncelikle, gördüğünüz gibi **`Label()`** pencere aracımız içinde **`wraplength`** adlı bir seçenek kullandık. Bu seçenek, **etikete yazdırılabilecek metnin uzunluğunu sınırlandırıyor**. Biz bu değeri **150 piksel** olarak belirledik. Buna göre etikete yazılan metin, **150 pikseli aştığında kendiliğinden bir alt satıra geçecektir**.

Şimdi, fonksiyonumuza bir göz atalım: Burada **`event.keysym`** adlı bir ifade görüyoruz. Bu ifade, klavye üzerindeki sembolleri, yani tuşları temsil ediyor. Fonksiyonumuz içinde yazdığımız koda göre kullanıcının girdiği klavye sembolleri **ana penceremiz üzerindeki etikete yazdırılacak**. Bir sonraki satırda yine **`bind()`** fonksiyonunu kullanarak **`göster()`** adlı fonksiyon ile **`<Key>`** adlı özel ifadeyi **birbirine bağlıyoruz**. Bu **`<Key>`** ifadesi, kullanıcının klavye üzerindeki tuşlara basma hareketini temsil ediyor.

Kodlarımızı **bir bütün hâlinde** düşünürsek; yukarıdaki uygulama, kullanıcının klavyede bastığı bütün tuşları ana pencere üzerindeki etikete atayacaktır. Bu uygulama yardımıyla, esasında **salt okunur** (*read-only*) **bir pencere aracı olan `Label()` aracını yazılabilir** (*writable*) **hâle getirmiş oluyoruz** (*en azından görünüş olarak*).

Öğrendiğimiz bu **`keysym`** niteliği yardımıyla bazı faydalı işler de yapabiliriz. Mesela bir **`Entry()`** aracına kullanıcı tarafından girilen bir metnin **silinmesini engelleyebiliriz**. *Nasıl mı?*

```
from tkinter import *

pencere = Tk()
pencere.geometry("200x200")

giriş = Entry()
giriş.pack()

def silemezsin(event=None):
    if event.keysym == "BackSpace" or event.keysym == "Delete":
        return "break"

giriş.bind("<Key>", silemezsin)

pencere.mainloop()
```

Buradaki kodlara göre eğer kullanıcı **`BACKSPACE`** veya **`DELETE`** **tuşlarına** basarak **yazdığı yazıyı silmek isterse beklentilerinin aksine bu tuşlar çalışmayacaktır**. Burada **`return`** "**`break`**" adlı özel bir ifadeden yararlanıyoruz. Bu ifade, normal şartlarda gerçekleşmesi engellenemeyecek olan işlemlerin **etkisizleştirilmesini sağlayan** özel bir kullanımdır. Örneğin yukarıdaki **`silemezsin()`** fonksiyonunu bir de şöyle yazmayı deneyin:

```
def silemezsin(event=None):
    if event.keysym:
        return "break"
```

Bu şekilde, kullanıcı **`Entry()`** aracı içine hiçbir şekilde yazı yazamayacaktır. Dolayısıyla yukarıdaki fonksiyonun adını "**`silemezsin`**" yerine "**`yazamazsın`**" koymak daha uygun olacaktır!

Son olarak bu konuyla ilgili olduğu için **`focus_set()`** fonksiyonundan da bahsetmekte fayda var. Bu fonksiyon, pencere araçlarını **etkin hâle getirmemizi sağlıyor**. Bu bölümün en başında verdiğimiz örneği hatırlıyorsunuz. Kullanıcı bir kutucuğa adını giriyor ve girdiği bu ad bir dosyaya yazdırılıyordu. O örneği isterseniz *yeniden çalıştırın*. Göreceğiniz gibi kutucuğa bir şey yazabilmek için öncelikle o kutucuğa **bir kez tıklamak gerekiyor**.

Yani aslında programımız ilk açıldığında o kutucuk etkin değil. Bizim onu etkin hâle getirmek için üzerine bir kez tıklamamız gerekiyor. Ama **istersek o kutucuğun açılışta etkin hâle gelmesini sağlayabiliriz**. **Böylece kullanıcılarımız doğrudan kutuya yazmaya başlayabilirler**:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()

etiket = Label(text="Aşağıdaki kutucuğa adınızı yazıp \
Tamam düğmesine basın!\n")
etiket.pack()

giriş = ttk.Entry()
giriş.pack()
giriş.focus_set()

def kaydet():
    dosya = open("isimler_python.txt", "a", encoding="utf-8")

    veri = giriş.get()

    dosya.write(veri+"\n")
    dosya.close()

    etiket["text"] += f"{giriş.get()} dosyaya başarıyla eklendi\n"

    giriş.delete(0, END)

düğme = ttk.Button(text="Tamam", command=kaydet)
düğme.pack(pady=10)

pencere.mainloop()
```

Burada **`giriş.focus_set()`** fonksiyonu sayesinde, programımız **açılır açılmaz kutucuk etkin hâle geliyor**. Böylece hemen adımızı yazmaya başlayabiliyoruz.

Bu konuyla bağlantılı olan şu örneğe bakalım bir de:

```
from tkinter import *
import tkinter.ttk as ttk
from sys import exit

pencere = Tk()

etiket = Label(text="Programdan çıkmak istediğinize emin misiniz?")
etiket.pack()

çerçeve = Frame()
çerçeve.pack(pady=10)

def sağol(event=None):
    yeni = Toplevel()
    etiket = Label(yeni, text="Teşekkürler... :)")
    yeni.bind("<Return>", exit)
    etiket.pack()

evet = ttk.Button(çerçeve, text="Evet", command=pencere.destroy)
evet.grid(row=1, column=0)

hayır = ttk.Button(çerçeve, text="Hayır", command=sağol)
hayır.grid(row=1, column=1)

evet.bind("<Return>", exit)
hayır.bind("<Return>", sağol)

pencere.mainloop()
```

Gördüğünüz gibi klavyedeki **ENTER** tuşunu bazı fonksiyonlara bağladığımız hâlde, programı çalıştırdığımızda **ENTER** tuşuna basmak **hiçbir işe yaramıyor**. Bunun nedeni, programımız ilk açıldığında **pencerenin kendisi hariç hiç bir pencere aracının etkin olmamasıdır**.

Bağladığımız tuşların çalışabilmesi için öncelikle ilgili pencere araçlarının **etkinleştirilmesi gerekiyor**:

```
from tkinter import *
import tkinter.ttk as ttk
from sys import exit

pencere = Tk()

etiket = Label(text="Programdan çıkmak istediğinize emin misiniz?")
etiket.pack()

çerçeve = Frame()
çerçeve.pack(pady=10)

def sağol(event=None):
    yeni = Toplevel()
    etiket = Label(yeni, text="Teşekkürler... :)")
    yeni.bind("<Return>", exit)
    etiket.pack()

kapat = ttk.Button(çerçeve, text="Evet", command=pencere.destroy)
kapat.grid(row=1, column=0)

kapatma = ttk.Button(çerçeve, text="Hayır", command=sağol)
kapatma.grid(row=1, column=1)

kapat.focus_set()

kapat.bind("<Return>", exit)
kapatma.bind("<Return>", sağol)

pencere.mainloop()
```

Buradaki tek fark, kodlar arasına **`kapat.focus_set()` ifadesinin eklenmiş olması**. Bu ifade sayesinde, programımız ilk açıldığında odak **Evet** düğmesi üzerinde olacaktır. Dolayısıyla **ENTER** tuşuna bastığımızda bu **düğmenin bağlı olduğu fonksiyon çalışacak** ve **programımız kapanacaktır**.

Program açıkken klavyedeki **TAB tuşuna basarak odağı değiştirebiliriz**. Mesela programımızı ilk çalıştırdığımızda odak **Evet** düğmesi üzerinde olacağı için **TAB** tuşuna bir kez bastığımızda odak **Hayır** düğmesine geçecektir. Bu hâldeyken **ENTER** tuşuna basarsak **Hayır düğmesinin bağlı olduğu fonksiyon çalışacaktır**.

İsterseniz konuyu kapatmadan önce basit bir *oyun denemesi yapalım*: Önce *https://www.dropbox.com/s/o69kdjnsjst41h7/sinek.png?dl=1* adresine gidip oradaki sinek resmini bilgisayarınıza indirin. Adını da **sinek.png** koyun.

Daha sonra şu kodları yazın:

```
from tkinter import *
import random
from PIL import ImageTk

pencere = Tk()

pencere.geometry("600x600")
pencere.tk_setPalette("white")

def yakala(event=None):
    k = random.randint(0,550)
    v = random.randint(0,550)
    print(k, v)

    düğme.place(x=k, y=v)

simge = ImageTk.PhotoImage(file="sinek.png")

düğme = Button(image=simge, command=yakala, relief="flat")
düğme.place(x=1, y=0)

düğme.bind("<Enter>", yakala)

pencere.mainloop()
```

Gördüğünüz gibi **Enter** ifadesi, kullanıcının **Enter tuşuna bastığı anlamına gelmiyor**. O iş için **Return** ifadesini kullanıyoruz.

Burada sineği yakalama ihtimaliniz var. Eğer **`random`** modülünün ürettiği sayılar birbirine yakın olursa sinek elinizden kaçamayabilir.

Böylelikle bu konuyu da bitirmiş olduk. Her ne kadar başlangıçta karışıkmış gibi görünse de fare ve klavye hareketleri konusu aslında hem çok **kolay** hem de **çok keyifli** bir konudur. Bu yazıyı birkaç kez gözden geçirerek bazı noktaların tam olarak zihninize yerleşmesini sağlayabilirsiniz.

Listbox Ögelerine Görev Atamak

Daha önce **`Listbox()`** (*Liste kutusu*) adlı pencere aracının ne olduğunu ve nasıl kullanıldığını öğrenmiştik. **Tkinter'de liste kutularını şöyle oluşturuyorduk**:

```
from tkinter import *

pencere = Tk()

listekutusu = Listbox()
listekutusu.pack()

pencere.mainloop()
```

İsterseniz bu liste kutusunun **daha belirgin olması için** buna birkaç öge ekleyelim:

```
from tkinter import *

pencere = Tk()

listekutusu = Listbox()
listekutusu.pack()

dağıtımlar = ["Debian", "Ubuntu", "Mandriva", "Arch", "Gentoo"]

for i in dağıtımlar:
    listekutusu.insert(0, i)

pencere.mainloop()
```

Elbette **`dağıtımlar`** adlı listedeki ögelerin, **`Listbox()`** içindeki konumlanışını **ters de çevirebiliriz**:

```
from tkinter import *

pencere = Tk()

listekutusu = Listbox()
listekutusu.pack()

dağıtımlar = ["Debian", "Ubuntu", "Mandriva", "Arch", "Gentoo"]

for i in dağıtımlar:
    listekutusu.insert(END, i)

pencere.mainloop()
```

Burada **`listekutusu.insert()`** satırında bir önceki kodlarda **`0`** sıra numarasını verirken **yukarıdaki kodlarda `END` ifadesini kullandığımıza dikkat edin**.

Gelin şimdi, isterseniz bu liste ögelerinin **her birine bir görev atayalım**. Yani mesela kullanıcı "**`Debian`**" adlı ögenin üzerine **çift tıkladığında** kendisine **Debian GNU/Linux** hakkında bilgi verelim:

```
from tkinter import *

pencere = Tk()
pencere.geometry("400x200")

listekutusu = Listbox()
listekutusu["relief"] = "raised"
listekutusu.pack(side=LEFT, anchor=NW, fill=BOTH)

etiket = Label()
etiket["text"] = ""

dağıtımlar = ["Debian", "Ubuntu", "Mandriva", "Arch", "Gentoo"]

for i in dağıtımlar:
    listekutusu.insert(END, i)
```

```
def gösterici(event):
    etiket["text"] = f"{listekutusu.get(ACTIVE)} bir GNU/Linux
dağıtımıdır!"
    etiket.pack()

listekutusu.bind("<Double-Button-1>", gösterici)

pencere.mainloop()
```

Burada bizi ilgilendiren kısım şu satır:

```
listekutusu.bind("<Double-Button-1>", gösterici)
```

Liste kutusuna görev **atama işlemini bu satır yardımıyla** yaptık. Kullanıcı liste kutusu içindeki herhangi bir ögeye **çift tıkladığı zaman**, tıklanan ögeyle ilgili bilgi veriyoruz. **`Listbox()`** adlı pencere aracının kendisine ait bir **`command`** seçeneği olmadığı için **istediğimiz işlevi, daha önceki derslerde gördüğümüz `bind()` metodu yardımıyla hâllediyoruz**. Burada **`Double-Button-1`** ifadesini kullandığımıza dikkat edin. Bu ifade, farenin sol tuşuna **çift tıklama hareketini** temsil ediyor. Liste kutularındaki ögelere işlev atamak istediğimiz zaman en doğru sonucu **`Double-Button-1`** ile elde ediyoruz. Öteki seçenekler her zaman istediğimiz işlevi yerine getirmeyebilir.

Pencereleri Başlıksız Hâle Getirmek

Tkinter'de **standart bir pencerenin** nasıl oluşturulacağını biliyoruz:

```
pencere = Tk()
```

Bu şekilde bir pencere oluşturduğumuzda elde ettiğimiz pencere, **bir pencerenin sahip olması gereken bütün özellikleri taşır**. Dolayısıyla pencereyi *kapatmak*, pencereyi *küçültmek* veya pencereyi *aşağı indirmek* istediğimizde **özel olarak herhangi bir kod yazmamıza gerek yoktur**. Pencere başlığı üzerindeki **çarpı**, **kare** ve **eksi düğmelerine tıklayarak gerekli işlevleri yerine getirebiliriz**.

Normal şartlar altında, bu tür özelliklerin olmadığı bir pencere pek bir işimize yaramaz. Ama bazen pencerenin *sürükleme çubuğu dâhil hiçbir özelliğinin olmamasını isteyebiliriz*. İşte bu bölümde, **bu isteğimizi nasıl yerine getirebileceğimizi göreceğiz**. Şimdi, normal bir şekilde programımızı yazalım:

```
from tkinter import *

pencere = Tk()

pencere.mainloop()
```

Bu şekilde **içi boş bir pencere** oluşturduk. Şimdi, bu kodlara şöyle bir şey ekleyelim:

```
pencere.overrideredirect(1)
```

Burada '**overrideredirect**' kelimesinin yazılışına **azami özen göstermek** gerekir. Uzun bir kelime olduğu için yanlış yazılma olasılığı oldukça yüksektir.

Kodlarımızın **son hâli** şöyle oldu:

```
from tkinter import *

pencere = Tk()

pencere.overrideredirect(1)

pencere.mainloop()
```

Yalnız bu kodları mutlaka komut satırından çalıştırın. Kesinlikle yazdığınız betiğin üzerine **çift tıklayarak** çalıştırmayı denemeyin.

Şimdi, **komut satırını kullanarak betiğimizi** çalıştırıyoruz. Gördüğünüz gibi **bomboş bir kare** elde ettik. Bu pencerenin, kendisini kapatmamızı sağlayacak bir düğmesi bile yok. Şimdi, sanırım neden bu betiği komut satırından çalıştırdığımızı anlamışsınızdır. Bu başlıksız pencereyi **CTRL+C** (*GNU/Linux*) veya **CTRL+Z** (*Windows*) tuşlarına basarak kapatabilirsiniz. Ya da **doğrudan komut ekranını kapatarak** da bu pencereyi ortadan kaldırabilirsiniz.

Peki, böyle başlıksız bir pencere oluşturmak ne işimize yarar? Mesela bu başlıksız pencereleri, bir düğmeye tıklandığında aşağıya doğru açılan bir menü olarak kullanabiliriz.

Şu örneğe bakalım:

```
from tkinter import *

pencere = Tk()

def liste():
    yenipencere = Toplevel()

    x = dgm.winfo_rootx()
    y = dgm.winfo_rooty() + dgm.winfo_height()

    genişlik = dgm.winfo_width()

    yenipencere.geometry(f"{genişlik}x100+{x}+{y}")

    menu = Listbox(yenipencere)
    menu.pack()

    yenipencere.overrideredirect(1)

    dağıtımlar = ["Debian", "Ubuntu", "Mandriva", "Arch", "Gentoo"]

    for i in dağıtımlar:
        menu.insert(0, i)

dgm = Button(text="listeyi göster", command=liste)
dgm.pack(fill=X)

pencere.mainloop()
```

Bu kodlar içinde geçen **x** ve **y** *değişkenlerinin ne olduğunu bir sonraki bölümde inceleyeceğiz*. Şimdilik bu değişkenlere fazla takılmayalım. Bu **x** ve **y** değişkenleri **yenipencere**'nin **ana pencereye** ve **bunun üzerindeki deneme** adlı düğmeye göre **konumunu belirlememizi sağlıyor**. Burada **deneme** adlı

düğmeye tıkladığımızda bir **seçim kutusu** açıldığını görüyoruz. Elbette burada sadece kavramsal olarak durumu göstermek için bir **örnek** verdik. Şu hâliyle yukarıdaki kodlar oldukça eksik. Amacım yalnızca başlıksız pencerelerle neler yapılabileceğine dair ufak bir örnek göstermekti... Yukarıdaki kodları **işe yarar bir hâle getirmek için** üzerinde biraz çalışmanız gerekir.

Pencere / Ekran Koordinatları ve Boyutları

Bu bölümde, **bir pencerenin veya üzerinde çalıştığımız ekranın koordinatlarını ve boyutlarını nasıl öğrenebileceğimizi inceleyeceğiz**.

Peki, bu bilgi ne işimize yarayacak?

Mesela yazdığımız program ilk açıldığında kullanıcının ekranını ortalasın istiyorsak programımızın çalıştırıldığı bilgisayar monitörünün **boyutlarını**/**çözünürlüğünü** bilmemiz gerekir ki buna göre **ekranın orta noktasını hesaplayabilelim**.

Ayrıca yazdığımız program içindeki pencere araçlarını da özellikle **belli noktalarda konumlandırmak istiyorsak pencere** ve **üzerindeki araçların koordinatlarını bilmemiz gerekir**. İsterseniz öncelikle hangi metotlardan yararlanabileceğimize bakalım:

```
>>> from tkinter import *
>>> pencere = Tk()
>>> for i in dir(pencere):
...     if i.startswith("winfo"):
...         print(i)
...
winfo_atom
winfo_atomname
winfo_cells
winfo_children
winfo_class
winfo_colormapfull
winfo_containing
winfo_depth
winfo_exists
```

```
winfo_fpixels
winfo_geometry
winfo_height
winfo_id
winfo_interps
winfo_ismapped
winfo_manager
winfo_name
winfo_parent
winfo_pathname
winfo_pixels
winfo_pointerx
winfo_pointerxy
winfo_pointery
winfo_reqheight
winfo_reqwidth
winfo_rgb
winfo_rootx
winfo_rooty
winfo_screen
winfo_screencells
winfo_screendepth
winfo_screenheight
winfo_screenmmheight
winfo_screenmmwidth
winfo_screenvisual
winfo_screenwidth
winfo_server
winfo_toplevel
winfo_viewable
winfo_visual
winfo_visualid
winfo_visualsavailable
winfo_vrootheight
winfo_vrootwidth
winfo_vrootx
winfo_vrooty
winfo_width
winfo_x
winfo_y
```

Gördüğünüz gibi elimizde epey metot var. Yukarıdaki kod yardımıyla **`Tk()`** sınıfı içindeki '**`winfo`**' **ile başlayan bütün metotları listelediğimize dikkat edin**. Biz bu bölümde bu metotların hepsini değil, bunlar arasında en sık kullanılanları inceleyeceğiz.

İnceleyeceğimiz metotlar şunlar olacak:

```
winfo_height
winfo_width
winfo_rootx
winfo_rooty
winfo_screenheight
winfo_screenwidth
winfo_x
winfo_y
```

Hemen ilk metodumuzla işe başlayalım.

WINFO_HEIGHT()

Bu metot, oluşturduğumuz **pencerenin** (*veya pencere aracının*) **yüksekliği** hakkında bilgi verir bize.

Şöyle bir örnekle durumu görelim:

```
>>> pencere = Tk()
>>> yükseklik = pencere.winfo_height()
>>> print(yükseklik)
1
```

Galiba bu **çıktı** pek fazla bir şey anlatmıyor bize. Bu kodlardan böyle bir çıktı almamızın nedeni bir pencere ilk oluşturulduğunda **yükseklik değerinin `1` olarak kabul edilmesidir**. Eğer pencerenin **gerçek boyutunu** görmek istersek yukarıdaki kodları şu şekilde yazmamız gerekir:

```
pencere = Tk()

pencere.update()

yükseklik = pencere.winfo_height()
```

```
print(yükseklik)

pencere.mainloop()
```

Burada kullandığımız **`pencere.update()`** komutu penceremizin mevcut durumunu **güncellememizi** sağlıyor. Dolayısıyla bu komut bize doğru bir şekilde **`200`** gibi bir **çıktı** veriyor. Buradan aldığımız sonuca göre oluşturduğumuz pencerenin yüksekliği **200 piksel**. Dediğimiz gibi bu metodu yalnızca pencere üzerine uygulamak zorunda değiliz. **Aynı metodu pencere araçlarıyla birlikte de kullanabiliriz**:

```
from tkinter import *

pencere = Tk()
pencere.geometry("200x200")

btn = Button(text="deneme")
btn.pack()

btn.update()

yükseklik = btn.winfo_height()

print(yükseklik)

pencere.mainloop()
```

Burada da **düğme bilgilerini güncellemek için** **`btn.update()`** gibi bir komuttan yararlandığımıza dikkat edin. Eğer bu örnekte **`update()`** metodunu kullanmazsak biraz önce olduğu gibi alacağımız **çıktı** **`1`** olacaktır.

Elbette **`winfo_height()`** metodunu **bir değişkene atamadan da** kullanabiliriz. Ama açıkçası o şekilde pek pratik olmayacaktır...

Şimdi sıra geldi, **`winfo_width()`** metodunu incelemeye.

WINFO_WIDTH()

İlk incelediğimiz metot olan **`winfo_height()`**, bir **pencere** veya **pencere aracının yüksekliğini** veriyordu. **`winfo_width()`** metodu ise pencere veya pencere aracının genişliğini verir. **Hemen görelim**:

```
from tkinter import *

pencere = Tk()

btn = Button(text=”deneme”)
btn.pack()

pencere.update()

genişlik = pencere.winfo_width()

print(genişlik)

pencere.mainloop()
```

Buradan aldığınız **çıktı**, pencerenin **genişliğini** gösterir. Eğer pencerenin değil de **düğmenin genişliğini almak isterseniz ne yapmanız gerektiğini tahmin edebilirsiniz**:

```
from tkinter import *

pencere = Tk()

btn = Button(text=”deneme”)
btn.pack()

pencere.update()

genişlik = btn.winfo_width()

print(genişlik)

pencere.mainloop()
```

WINFO_ROOTX()

Bu metot **pencere** veya **pencere araçlarının x düzlemi üzerindeki koordinatını verir.** Mesela:

```
from tkinter import *

pencere = Tk()
pencere.geometry("200x200")

btn = Button(text="deneme")
btn.pack()

pencere.update()

xkoord = pencere.winfo_rootx()

print(xkoord)

pencere.mainloop()
```

Ben bu kodları, kendi bilgisayarımda çalıştırdığımda **112** çıktısını aldım. Benim monitörümün çözünürlüğü **1366x768**. Demek ki **bu pencerenin sol kenarı, ekranın soldan sağa 112'nci noktasına denk geliyormuş**.

Bir de şuna bakalım:

```
from tkinter import *

pencere = Tk()
pencere.geometry("200x200")

btn = Button(text="deneme")
btn.pack()
btn.update()

xkoord = btn.winfo_rootx()

print(xkoord)

pencere.mainloop()
```

Bu kodları çalıştırdığımda ise konsolda **105** çıktısını gördüm. Demek ki deneme **adlı düğmenin sol kenarı 1366x768'lik monitörün 105'inci noktasına denk geliyormuş**...

WINFO_ROOTY()

Bir önceki metodumuz olan **winfo_rootx(), x** koordinatlarının bilgisini veriyordu. **winfo_rooty()** ise **y** koordinatlarının bilgisini verir:

```
from tkinter import *

pencere = Tk()
pencere.geometry("200x200")

btn = Button(text="deneme")
btn.pack()

pencere.update()

ykoord = pencere.winfo_rooty()
print(ykoord)

pencere.mainloop()
```

Buradan aldığımız sonuç, **pencerenin üst sınırının y düzlemi üzerinde hangi noktaya karşılık geldiğini gösteriyor**. Penceremiz ekranın en tepesinde bile olsa bu kodlar **0** veya **1** gibi bir **sonuç vermez**. Çünkü pencere başlığının, hemen altındaki alanın **karşılık geldiği nokta hesaplanmaktadır**.

Bir de şu örneğe bakalım:

```
from tkinter import *

pencere = Tk()
pencere.geometry("200x200")

btn = Button(text="deneme")
btn.pack(pady=30)

btn.update()

ykoord = btn.winfo_rooty()

print(ykoord)

pencere.mainloop()
```

Burada, **pady** seçeneğini kullanarak düğmeyi biraz **aşağıya kaydırdık ki bir önceki kodla arasındaki farkı görebilelim...**

WINFO_SCREENHEIGHT()

Bu metot, ekran **yüksekliğinin kaç olduğunu söyler**. Örneğin 1366x768'lik bir çözünürlükte bu metodun verdiği değer 768 olacaktır...

```
from tkinter import *

pencere = Tk()

ekran_y = pencere.winfo_screenheight()

print(ekran_y)

pencere.mainloop()
```

WINFO_SCREENWIDTH()

Bu metot da **`winfo_screenheight()`** metoduna benzer. **Ama onun aksine, bir pencerenin yüksekliğini değil genişliğini verir**. Dolayısıyla 1366x768'lik bir ekran çözünürlüğünde bu değer **`1366`** olacaktır:

```
from tkinter import *

pencere = Tk()

ekran_g = pencere.winfo_screenwidth()

print(ekran_g)

pencere.mainloop()
```

En önemli **`winfo`** metotlarını gördüğümüze göre bunları kullanarak bazı yararlı işler yapmaya başlayabiliriz...

Programı Tam Ekran olarak Çalıştırmak

Tkinter'de bir pencereyi **tam ekran yapmak için** şu kodu kullanıyoruz:

```
from tkinter import *

pencere = Tk()
pencere.state("zoomed")

pencere.mainloop()
```

Burada pencereyi **tam ekran olarak açan kod şu**:

```
pencere.state("zoomed")
```

Ama eğer isterseniz **tam ekrana yakın bir etkiyi** `winfo` **metotları ile de elde edebilirsiniz**. Örneğin:

```
from tkinter import *

pencere = Tk()

gen = pencere.winfo_screenwidth()
yuks = pencere.winfo_screenheight()

pencere.geometry(f"{gen}x{yuks}")

dgm = Button(text="~~~~~~~~~~~~TAM EKRAN~~~~~~~~~~")
dgm.pack(expand=YES, fill=BOTH)

pencere.mainloop()
```

Burada yaptığımız şey şu: Önce **`pencere.winfo_screenwidth()`** ifadesi yardımıyla **ekran genişliğini** alıyoruz. Daha sonra **`pencere.winfo_screenheight()`** ifadesini kullanarak **ekran yüksekliğini** öğreniyoruz.

Bu bilgileri, kullanım kolaylığı açısından **gen** ve **yuks** adlı iki değişkene atadık. Ardından da **pencere.geometry()** içinde bu değerleri kullanarak programımızın **ilk açılışta ekranın hemen hemen tamamını kaplamasını sağladık**.

Python'ın **arka planda neler çevirdiğini daha net görmek için** kullandığımız bu **gen** ve **yuks** değişkenlerini ekrana yazdırmak isteyebilirsiniz.

Ekranı Ortalamak

winfo metotlarını kullanarak yazdığımız bir programın ilk açıldığında **ekranın tam ortasına denk gelmesini** de sağlayabiliriz:

```
from tkinter import*

pencere = Tk()

pgen = 200
pyuks = 200

ekrangen = pencere.winfo_screenwidth()
ekranyuks = pencere.winfo_screenheight()

x = (ekrangen - pgen) // 2
y = (ekranyuks - pyuks) // 2

pencere.geometry(f"{pgen}x{pyuks}+{x}+{y}")

pencere.mainloop()
```

Burada önce **pgen** ve **pyuks** değişkenleri içinde programımızın **genişliğini** ve **yüksekliğini** belirttik. Bunları elbette doğrudan **pencere.geometry** ifadesi içine de yerleştirebilirdik. Ama bu iki değerle **bazı hesaplamalar yapacağımız için** en iyisi bunları bir değişkene atamak.

İlk değişkenlerimizi tanımladıktan sonra **ekran genişliğini** ve **ekran yüksekliğini** öğreniyoruz. Bunun için **winfo_screenwidth()** ve **winfo_screenheight()** metotlarını kullandık. Yine kullanım kolaylığı açısından bu iki değeri sırasıyla **ekrangen** ve **ekranyuks** adlı değişkenlere atıyoruz.

Şimdi, programımızın **ekranı tam ortalayabilmesi için** ufak bir hesap yapmamız gerekecek. **x** değerini bulabilmek için ekran genişliğinden programımızın pencere genişliğini çıkarıp **elde ettiğimiz değeri ikiye bölüyoruz**. Mesela; eğer kullandığımız ekran çözünürlüğü **1024x768** ise **x** değeri şöyle olacaktır:

```
x = (1024 - 200) // 2
```

Burada, **taban bölme işlemi yaptığımıza dikkat edin**. Çünkü biraz sonra **`geometry()`** metodu içinde bir kayan noktalı sayıya değil, tam sayıya ihtiyacımız olacak.

y değerini bulmak için de benzer bir şekilde **ekran yüksekliğinden program yüksekliğini çıkarıp** bu değeri yine **ikiye** bölüyoruz. Dolayısıyla **y** değeri **1024x768**'lik bir ekranda şöyle olur:

```
y = (768 - 200) // 2
```

Bu hesaplamadan elde ettiğimiz verileri **`pencere.geometry()`** içinde uygun yerlere yerleştirdiğimizde, **programımız ekranın tam ortasında açılacaktır**.

Pencereleri Her Zaman En Üstte Tutmak

Bazen ana pencereye **ek olarak ikinci bir pencere daha oluşturmamız gerekir**. Ancak bazen programın işleyişi sırasında, aslında hep üstte durması gereken bu ikinci pencerenin, **ana pencerenin arkasına düştüğünü görürüz**. Böyle bir durumda yapmamız gereken şey, **ikinci pencerenin daima üstte kalmasını sağlayacak bir kod yazmaktır**. Neyse ki Tkinter bize böyle bir durumda kullanılmak üzere faydalı bir metot sunar. Bu metodun adı **`transient()`**. İsterseniz hemen bununla ilgili bir **örnek** yapalım. Diyelim ki şöyle bir uygulamamız var:

```
from tkinter import *
from tkinter import filedialog as fd

pencere = Tk()
```

```
pgen = 200
pyuks = 200

ekrangen = pencere.winfo_screenwidth()
ekranyuks = pencere.winfo_screenheight()

x = (ekrangen - pgen) // 2
y = (ekranyuks - pyuks) // 2

pencere.geometry(f"{pgen}x{pyuks}+{x}+{y}")

def yeniPencere():
    yeni = Toplevel()

    xkonum = pencere.winfo_rootx()
    ykonum = pencere.winfo_rooty()

    yeni.geometry(f"+{xkonum}+{ykonum}")

    ybtn = Button(yeni, text="Dosya aç", command=yeniDosya)
    ybtn.pack()

def yeniDosya():
    dosya = fd.askopenfilename()

btn = Button(pencere, text="yeni pencere aç", command=yeniPencere)
btn.pack()

pencere.mainloop()
```

Burada gördüğümüz **`filedialog`** modülü, dosya açma işlemlerinde kullanılan ve **`tkinter`** modülü içinde yer alan **ayrı bir modüldür**.

Programımızın, **ekranın tam ortasında açılacak şekilde ayarlandığına dikkat edin**. Aynı şekilde ikinci pencere de ana pencerenin üzerinde açılacak şekilde ayarlandı. Bu işlemleri **`winfo`** metotları yardımıyla yaptığımızı görüyorsunuz.

Bu programı çalıştırdığımızda pencere araçlarının biraz tuhaf davrandığını görebilirsiniz. Mesela ikinci pencere üzerindeki **Dosya aç** düğmesine bastığımızda açılan dosya seçme penceresi **ikinci pencerenin altında kalıyor** olabilir.

Aynı şekilde, dosya seçme ekranında **Cancel** tuşuna bastığımızda ikinci pencere **bir anda ortadan kaybolacak** ve **ana pencerenin arkasına gizlenecektir**. Bu program belki **her sistemde aynı tepkiyi vermeyebilir**. Hatta belki bazı sistemlerde bu şekilde bile düzgün çalışıyor olabilir. Ama bizim istediğimiz şey, **programımızın hemen her sistemde mümkün olduğunca aynı şekilde çalışmasını sağlamak**. O hâlde hemen gerekli kodları yazalım. Yazacağımız kod çok basittir. Tek yapmamız gereken ikinci pencerenin ana pencereye göre **üstte kalmasını garanti etmek**.

Bunu şu satırla yapacağız:

```
yeni.transient(pencere)
```

Yani kodlarımızın **son hâli** şöyle olacak:

```
from tkinter import *
from tkinter import filedialog as fd

pencere = Tk()

pgen = 200
pyuks = 200

ekrangen = pencere.winfo_screenwidth()
ekranyuks = pencere.winfo_screenheight()

x = (ekrangen - pgen) //2
y = (ekranyuks - pyuks) //2

pencere.geometry(f"{pgen}x{pyuks}+{x}+{y}")

def yeniPencere():
    yeni = Toplevel()
    yeni.transient(pencere)
```

```
    xkonum = pencere.winfo_rootx()
    ykonum = pencere.winfo_rooty()

    yeni.geometry(f"+{xkonum}+{ykonum}")

    ybtn = Button(yeni, text="Dosya aç", command=yeniDosya)
    ybtn.pack()

def yeniDosya():
    dosya = fd.askopenfilename()

btn = Button(pencere, text="yeni pencere aç", command=yeniPencere)
btn.pack()

pencere.mainloop()
```

Yeni eklediğimiz satır, **yeni** adlı **ikinci pencerenin ana pencereye göre hep üstte kalmasını temin ediyor**. Programımızı bu şekilde çalıştırdığımızda her şeyin olması gerektiği gibi olduğunu göreceğiz. Kodlarımızı bu şekilde yazdığımızda, ikinci pencereyi açtıktan sonra **ana pencereye tıklasak bile ikinci pencere ana pencerenin arkasına düşmeyecektir**.

STANDART BİLGİ PENCERELERİ

7

Bu bölümün konusu, **Tkinter'de Standart Bilgi Pencereleri'nin kullanımı**. *Nedir bu "Standart Bilgi Penceresi" denen şey?* Diyelim ki bir program yazıyorsunuz. Bu programda herhangi bir işlemle ilgili olarak kullanıcıya bir **mesaj** göstermek istediğimizde karşımızda iki yol var. **Birinci yolda**, Tkinter'in **`Toplevel()`** aracını kullanarak göstermek istediğimiz mesajı ve gerekli düğmeleri içeren bir pencere oluşturabiliriz. Hemen bununla ilgili bir **örnek** verelim:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()

def kapat():
    if not entry.get():
        yenipen = Toplevel()
        uyarı = Label(yenipen)
        uyarı["text"] = "Lütfen geçerli bir e.posta adresi yazın!"
        uyarı.pack()

        uybtn = ttk.Button(yenipen)
        uybtn["text"] = "Tamam"
        uybtn["command"] = yenipen.destroy
        uybtn.pack()
    else:
        pencere.destroy()
```

```
etiket = Label()
etiket["text"] = "e.posta adresiniz:"
etiket.pack()

entry = ttk.Entry()
entry.pack(padx=10)

dgm = ttk.Button()
dgm["text"] = "Gönder"
dgm["command"] = kapat
dgm.pack(pady=10)

pencere.mainloop()
```

Burada eğer kullanıcı **`Entry()`** aracına herhangi bir şey yazmadan "**Gönder**" düğmesine basarsa bir **uyarı penceresi açılacak** ve kullanıcıyı **`Entry()`** aracına geçerli bir e-posta yazması konusunda uyaracaktır.

Dediğimiz gibi böyle bir işlev yerine getirmeniz gerekirse yukarıdaki yöntemi kullanabilirsiniz. Ama aslında buna hiç gerek yok. Tkinter bize, **bu tür işleri yapabilmemiz için bazı araçlar sunar**. İşte biz de bu bölümde Tkinter'in bize sunduğu bu araçları inceleyeceğiz.

Hata Mesajı Gösteren Pencere

Bu bölümde inceleyeceğimiz ilk penceremiz, **hata mesajı gösteren pencere** olacaktır. Yalnız bahsettiğimiz bu pencereleri kullanabilmek için öncelikle **`tkinter`** modülü içinde ayrı bir modül olan **`messagebox`** adlı başka bir modülü **içe aktarmamız gerekiyor**:

```
from tkinter import messagebox
```

Tkinter'de kullanıcıya bilgi mesajı gösteren bütün pencereler **`messagebox`** modülü içinde yer alıyor. Dolayısıyla bu pencereleri kullanmak istediğimizde, Tkinter modülünün yanı sıra **`messagebox`** **modülünü de ayrıca içe aktarmayı unutmuyoruz**.

Bir önceki bölümde hatırlarsanız, kullanıcıdan e-posta adresi girmesini istediğimiz ufak bir uygulama yazmıştık. Yine o **örnek** üzerinden gidelim.

Öncelikle kodlarımızın ne olduğuna bakalım:

```
from tkinter import *
import tkinter.ttk as ttk

pencere = Tk()

def kapat():
    if not entry.get():
        yenipen = Toplevel()
        uyarı = Label(yenipen)
        uyarı["text"] = "Lütfen geçerli bir e.posta adresi yazın!"
        uyarı.pack()

        uybtn = ttk.Button(yenipen)
        uybtn["text"] = "Tamam"
        uybtn["command"] = yenipen.destroy
        uybtn.pack()
    else:
        pencere.destroy()

etiket = Label()
etiket["text"] = "e.posta adresiniz:"
etiket.pack()

entry = ttk.Entry()
entry.pack(padx=10)

dgm = ttk.Button()
dgm["text"] = "Gönder"
dgm["command"] = kapat
dgm.pack(pady=10)

pencere.mainloop()
```

Gördüğünüz gibi aslında **basit bir iş için çok fazla kod** yazmışız.

Gelin bu kodları, **messagebox** modülünün de yardımıyla **sadeleştirelim** ve hatta **güzelleştirelim**:

```
from tkinter import *
import tkinter.ttk as ttk
from tkinter import messagebox

pencere = Tk()

def kapat():
    if not entry.get():
        messagebox.showerror("Hata!",
                             "Lütfen geçerli bir e.posta adresi
girin!")
    else:
        pencere.destroy()

etiket = Label()
etiket["text"] = "e.posta adresiniz:"
etiket.pack()

entry = ttk.Entry()
entry.pack(padx=10)

dgm = ttk.Button()
dgm["text"] = "Gönder"
dgm["command"] = kapat
dgm.pack(pady=10)

pencere.mainloop()
```

Ne kadar hoş, değil mi? Bu şekilde hem **daha az kod** yazdık hem de **elde ettiğimiz çıktı daha oturmuş bir görünüme sahip**. Dikkat ederseniz **hata mesajımız sevimli bir hata simgesi de içeriyor**. Bu hata penceresinin bize sağladıklarını kendimiz yapmaya çalışırsak epey kod yazmamız gerekir. Dolayısıyla yazacağımız programlarda böyle bir imkânımızın olduğunu da göz önünde bulundurmamız bizim için faydalı olacaktır.

Bu arada, her defasında **messagebox** modülünün adını yazmak *sıkıcı* ve *yorucu* olacaktır. Ayrıca kodlarımız içinde de **gereksiz yere fazla yer kaplayacaktır**. O yüzden bu modülü şu şekilde içe aktarmayı tercih edebiliriz:

```
from tkinter import messagebox as mb
```

Böylece **bir hata penceresinin sahip olması gereken bütün özellikleri taşıyan bu hata penceresini**, şu kod yardımıyla oluşturabileceğiz:

```
mb.showerror("Hata!",
             "Lütfen geçerli bir e.posta adresi girin!")
```

Gördüğünüz gibi '**mb**' adıyla **içe aktardığımız messabox** modülü içindeki **showerror()** adlı bir fonksiyonu kullanıyoruz. Bu fonksiyonu **iki parametreli** olarak kullandık. **showerror()** fonksiyonu içinde kullandığımız ilk parametre "**Hata**" adlı bir karakter dizisi. Bu ifade, oluşacak hata penceresinin **başlığı** olacak. İkinci parametremiz olan "**Lütfen geçerli bir e.posta adresi girin!**" adlı karakter dizisi ise hata penceremizin gövdesi, yani **kullanıcıya göstereceğimiz mesajın ta kendisidir**.

Burada kullandığımız **showerror()** fonksiyonu başka bir parametre daha alabilir. Bu parametrenin adı **detail**'dir. **Bu parametreyi şöyle kullanıyoruz**:

```
mb.showerror("Hata!",
             "Lütfen geçerli bir e.posta adresi girin!",
             detail = "e.posta adresini doğru yazmaya özen
gösterin!")
```

detail parametresi, varsa hatayla ilgili ayrıntıları da kullanıcıya göstermemizi sağlıyor. Bu arada **yukarıdaki örnek kod satırı eğer gözünüze uzun görünüyorsa bu kodları şöyle de yazabilirsiniz**:

```
baslik = "Hata!"
mesaj = "Lütfen geçerli bir e.posta adresi girin!"
detay = "e.posta adres uydurma olmasın!"

mb.showerror(baslik, mesaj, detail = detay)
```

detail parametresi dışında, **showerror()** fonksiyonu için (*ve tabii öteki bilgi mesajları için*) kullanabileceğimiz başka parametreler de vardır. Mesela bu parametrelerden biri **type** parametresidir. **Bu parametre ile bilgi ekranında gösterilecek düğmelerin tipini belirleyebiliriz**.

Bu parametre ile birlikte kullanabileceğimiz altı farklı seçenek vardır:

- **ABORTRETRYIGNORE:** Pencere üzerinde vazgeç, tekrar dene, yoksay düğmelerini gösterir.
- **OK:** Pencere üzerinde tamam düğmesi gösterir.
- **OKCANCEL:** Pencere üzerinde tamam ve iptal düğmelerini gösterir.
- **RETRYCANCEL:** Pencere üzerinde yeniden dene ve iptal düğmelerini gösterir.
- **YESNO:** Pencere üzerinde evet ve hayır düğmelerini gösterir.
- **YESNOCANCEL:** Pencere üzerinde evet, hayır ve iptal düğmelerini gösterir.

Bu seçenekler şöyle kullanılır:

```
mb.showerror(baslik, mesaj, detail=detay, type=mb.ABORTRETRYIGNORE)
```

ABORTRETRYIGNORE seçeneğinin **messagebox** modülünün bir niteliği olduğuna dikkat ediyoruz. O yüzden bu seçeneği **her zaman modül adı ile birlikte kullanıyoruz**. Biz kodlarımızda **messagebox** modülünü '**mb**' adıyla **içe aktardığımız için** **mb.ABORTRETRYIGNORE** ifadesini kullandık.

Burada **ABORTRETRYIGNORE** yerine, başka herhangi bir seçeneği de koyabilirsiniz elbette...

Başka bir parametre ise **default** parametresidir. Bu parametre yardımıyla, hata penceresi ilk açıldığında hangi düğmenin **seçili** (*etkin*) olacağını belirleyebiliriz. Yani doğrudan **ENTER** tuşuna basıldığında, **hangi düğmenin çalışacağını** bu **default** adlı parametre ile belirliyoruz.

default parametresi yedi farklı seçeneğe sahiptir:

- **ABORT**
- **RETRY**
- **IGNORE**
- **OK**
- **CANCEL**
- **YES**
- **NO**

Bu parametreyi de şöyle kullanıyoruz:

```
messagebox.showerror(baslik, mesaj, detail=detay,
                     type=mb.ABORTRETRYIGNORE, default=mb.RETRY)
```

En son parametremiz ise **icon** adlı parametredir. Bu parametre yardımıyla, bilgi penceresi üzerinde gördüğümüz simgenin **türünü** belirleyebiliriz. Bu parametre ile şu seçenekleri kullanabiliriz:

- **ERROR** (*Hata simgesi*)
- **INFO** (*Bilgi simgesi*)
- **QUESTION** (*Soru simgesi*)
- **WARNING** (*Uyarı simgesi*)

Bu parametreyi ise şöyle kullanacağız:

```
mb.showerror(baslik, mesaj, detail=detay,
             type=mb.ABORTRETRYIGNORE, default=mb.RETRY,
             icon=mb.WARNING)
```

Bu arada, **mümkün olan yerlerde kod satırlarımızı yukarıda gibi uzun tutmamak iyi bir yaklaşım olacaktır**. Dolayısıyla yukarıdaki satırı biraz daha bölmek okunaklığı artırır:

```
mb.showerror(baslik,
             mesaj,
             detail = detay,
```

```
            type = mb.ABORTRETRYIGNORE,
            default = mb.RETRY,
            icon= mb.WARNING)
```

Peki, kullanıcı bu bilgi ekranındaki herhangi bir düğmeye bastığında hangi işlemin yapılacağını nasıl belirleyeceğiz? Yani mesela diyelim ki kullanıcı **OK** tuşuna bastı, işte böyle bir durumda hangi işlemin yapılması gerektiğini *nasıl bulacağız?* **Bu işi yapmakta oldukça kolaydır**.

Hemen şöyle bir **örnek** verelim:

```
from tkinter import *
import tkinter.ttk as ttk
from tkinter import messagebox as mb

pencere = Tk()

def soru():
    cevap = mb.showerror("Uyarı",
                         "Bu işlem diskteki bütün verileri
silecektir.",
                         type=mb.OKCANCEL)
    if cevap == "ok":
        etiket["text"] = "disk biçimlendirildi!"

    if cevap == "cancel":
        etiket["text"] = "işlem durduruldu!"

etiket = Label(text="işlem durumu: ")
etiket.pack()

dgm = ttk.Button(text="diski biçimlendir!", command=soru)
dgm.pack(pady=10)

pencere.mainloop()
```

Gördüğünüz gibi **`showerror()`** fonksiyonunu bir değişkene atadık. Burada **`type`** parametresini kullanarak içinde **OK** ve **CANCEL** gibi iki farklı düğme barındıran bir **bilgi ekranı** oluşturduğumuza dikkat edin.

Eğer kullanıcı OK tuşuna basarsa **cevap** değişkeninin değeri **ok**, yok eğer kullanıcı CANCEL tuşuna basarsa cevap değişkeninin değeri **cancel** olacaktır. İşte bu değerleri birer **if** deyimine bağlayarak **her bir düğmeye ayrı bir işlev atayabiliriz**.

Burada kendi kendinize, **type** parametresine farklı değerler vererek denemeler yapmanızı tavsiye ederim. Mesela; **OKCANCEL**, **RETRYCANCEL**, **ABORTRETRYIGNORE** gibi değerlerin ne tür çıktılar verdiğini kontrol ederek **çıktıya göre farklı fonksiyonlar yazabilirsiniz**.

Mesela şöyle bir test uygulaması ile her bir düğmenin nasıl bir **çıktı** verdiğini kontrol edebilirsiniz:

```
from tkinter import *
import tkinter.ttk as ttk
from tkinter import messagebox as mb

pencere = Tk()

def soru():
    cevap = mb.showerror("Uyarı",
                         "Bu işlem diskteki bütün verileri
silecektir.",
                         type=mb.ABORTRETRYIGNORE)
    print(cevap)

dgm = ttk.Button(text="diski biçimlendir!", command=soru)
dgm.pack(pady=10)

pencere.mainloop()
```

Elbette burada, **çıktıları konsola veriyoruz**. Ama eğer siz isterseniz arayüz üzerindeki **Label()** aracının **text** seçeneğine de bu çıktıları yazdırabilirsiniz:

```
from tkinter import *
import tkinter.ttk as ttk
from tkinter import messagebox as mb

pencere = Tk()
```

```
def soru():
    cevap = mb.showerror("Uyarı",
                         "Bu işlem diskteki bütün verileri
silecektir.",
                         type=mb.ABORTRETRYIGNORE)
    etiket["text"] = cevap

etiket = Label()
etiket.pack()

dgm = ttk.Button(text="diski biçimlendir!", command=soru)
dgm.pack(pady=10)

pencere.mainloop()
```

Gördüğünüz gibi **showerror()** fonksiyonunu kullanmak zor değil. Gelin isterseniz şimdi, öteki bilgi pencerelerine de şöyle bir göz atalım.

Bilgi Mesajı Gösteren Pencere

Bir önceki bölümde, **hata mesajı** gösteren pencereyi işlemiştik. Bu bölümde ise "**bilgi mesajı gösteren pencere**"yi öğreneceğiz.

Bilgi mesajı gösteren pencere de tıpkı hata mesajı gösteren pencere gibidir. Bu kez **showerror()** fonksiyonu yerine **showinfo()** fonksiyonunu kullanacağız. En basit şekliyle **showinfo()** fonksiyonu şöyle kullanılır:

```
mb.showinfo("Bilgi", "İşlem başarıyla tamamlandı!")
```

Gelin, isterseniz bu fonksiyonu **kapsamlı bir örnek** içinde inceleyelim:

```
from tkinter import *
import tkinter.ttk as ttk
from tkinter import messagebox as mb

pencere = Tk()

def kapat():
```

```
    if not entry.get():
        mb.showerror("Hata!", "Ürün sayısı belirtmediniz!")
        return "break"

    try:
        int(entry.get())
        mb.showinfo("Bilgi", f"{entry.get()} ürün sepete eklendi!")

    except ValueError:
        mb.showerror("Hata!", "Lütfen bir sayı girin!")

etiket = Label()
etiket["text"] = "ürün adedi:"
etiket.pack()

entry = ttk.Entry()
entry.pack(padx=10)

dgm = ttk.Button()
dgm["text"] = "Gönder"
dgm["command"] = kapat
dgm.pack(pady=10)

pencere.mainloop()
```

Burada, **birkaç işlemi aynı anda yaptık**. Programımızın amacı, **kullanıcının kutucuğa bir sayı yazmasını sağlamak**. Ayrıca kullanıcının, kutucuğa hiçbir şey yazmadan "**Gönder**" tuşuna basabileceğini de hesaba katmamız gerekir... Böyle bir duruma karşı, kodlarımız içinde şu satırları yazdık:

```
if not entry.get():
    mb.showerror("Hata!", "Ürün sayısı belirtmediniz!")
    return "break"
```

Eğer kullanıcımız, kutucuğu **boş bırakırsa** kendisine bir **hata mesajı** gösteriyoruz. Burada `return "break"` ifadesini kullanmamızın nedeni, **hata mesajı gösterildikten sonra programın çalışmaya devam etmesini engellemek**.

Eğer bu ifadeyi yazmazsak **programımız kullanıcıya hata mesajı gösterdikten sonra alt satırdaki kodları da işletmeye devam edecektir**. İsterseniz, o satırı kaldırıp kendi kendinize *birtakım denemeler yapabilirsiniz*...

Bildiğimiz gibi **`entry.get()`** metodundan elde edilen verinin tipi **karakter dizisidir**. Kullanıcının bir sayı mı yoksa bir harf mi girdiğini tespit edebilmek için öncelikle, **kutucuğa kullanıcı tarafından yazılan değerin sayıya dönüştürülebilen bir veri olup olmadığına bakıyoruz**.

Bunun için şu kodları yazdık:

```
try:
    int(entry.get())
    mb.showinfo("Bilgi", f"{entry.get()} ürün sepete eklendi!")

except ValueError:
    mb.showerror("Hata!", "Lütfen bir sayı girin!")
```

Burada, eğer kullanıcının yazdığı şey **sayıya dönüştürülebilen bir veri ise** kendisine **`showinfo()`** fonksiyonunu kullanarak **kaç ürünün sepete eklendiği bilgisini veriyoruz**. Ama eğer kullanıcı sayı girmek yerine, örneğin "**fsdfd**" gibi bir şey yazarsa bu defa **`showerror()`** fonksiyonunu kullanarak ona, **sayı girmediğine dair bir hata mesajı gösteriyoruz**.

Burada **`try... except`** bloklarını nasıl kullandığımıza dikkat edin...

Gördüğünüz gibi **`showinfo()`** fonksiyonunun kullanımı, **`showerror()`** fonksiyonunun kullanımına **çok benziyor**. Hatta hemen hemen aynı bile diyebiliriz...

Geçen bölümde, **showerror()** fonksiyonunu işlerken bu fonksiyonun **birtakım parametreler de alabildiğini söylemiştik**. Orada söylediklerimiz **showinfo()** fonksiyonu için de geçerlidir. Mesela burada da **detail** parametresini kullanarak kullanıcıya konuyla ilgili **ek bilgi** verebiliriz:

```
mb.showinfo("Bilgi", f"{entry.get()} ürün sepete eklendi!",
            detail = "Ama bu kadar ürün az!" +
                     "\nÜrünlerimizden biraz daha alın!")
```

showerror() ile **showinfo()** arasındaki benzerlikler bununla da sınırlı değildir. Mesela geçen bölümde öğrendiğimiz **type** parametresini burada da kullanabiliriz:

```
mb.showinfo("Bilgi", f"{entry.get()} ürün sepete eklendi!",
            detail = "Ama bu kadar ürün az!" +
                     "\nÜrünlerimizden biraz daha alın!",
            type = mb.OKCANCEL)
```

Gördüğünüz gibi **showerror()** konusunda ne öğrendiysek burada da uygulayabiliyoruz. İsterseniz geçen bölümde öğrendiğimiz bilgileri **showinfo()** fonksiyonuna da uygulayarak *kendi kendinize el alıştırmaları yapabilirsiniz*.

Sıra geldi, başka bir bilgi mesajı penceresini incelemeye...

Uyarı Mesajı Gösteren Pencere

Bu bölümde, kullanıcılarımıza bir **uyarı mesajı gösteren pencerenin nasıl oluşturulacağına bakacağız**. Burada da yeni bir fonksiyonla karşılaşacağız. Ama endişelenmeye gerek yok. Bu fonksiyon da tıpkı **showerror()** ve **showinfo()** fonksiyonlarına benzer. **Adı ve tabii ki üzerindeki ünlem simgesi dışında her yönüyle onlarla aynıdır**. Bu fonksiyonumuzun adı **showwarning()**. İsterseniz bununla ilgili de basit bir **örnek** yapıp bu konuyu kapatalım:

```
from tkinter import *
import tkinter.ttk as ttk
from tkinter import messagebox as mb
```