

Python Uygulamaları

Kurs Notları

Kaan ASLAN

C ve Sistem Programcılar Derneği

Güncelleme Tarihi: 09/10/2020

Bu kurs notları Kaan ASLAN tarafından yazılmıştır. Kaynak belirtilmek koşuluyla her türlü alıntı yapılabılır.

Python Standard Kütüphanesindeki Bazı Temel Fonksiyonlar ve Sınıflar

Bu bölümde Python'ın standart kütüphanesindeki çeşitli modüller içerisinde bulunan çeşitli temel fonksiyonlar ve sınıflar ele alınacaktır. Bu temel fonksiyonlar ve sınıflar başka konularda da çeşitli biçimlerde kullanılmaktadır.

Python'da Tarih ve Zaman İşlemleri

Bu bölümde Python Standart Kütüphanesindeki tarih ve zaman işlemlerini yapan fonksiyonlar ve sınıflar ele alınacaktır.

time Modülü

time modülü C Programlama Dilindeki prototipleri <time.h> içerisinde bulunan standart tarih zaman işlemleri yapmaktadır. C programcılar bu modüldeki fonksiyonlara oldukça aşina olacaklardır. Bu modülü kullanmadan önce import etmemeyi unutmayın.

time.time Fonksiyonu

time fonksiyonu 01/01/1970'den geçen saniye sayısını float olarak verir. Örneğin:

```
>>> time.time()
1532699789.392054
```

Örneğin:

```
import time

start = time.time()

for i in range(100000000):
    pass

end = time.time()

print(end - start)
```

time.ctime Fonksiyonu

ctime fonksiyonu time fonksiyonundan elde edilen saniye sayısını parametre olarak alır ve tarih zaman bilgisini bize str türünden yazışsal biçimde verir. Örneğin:

```
>>> t = time.time()
>>> time.ctime(time())
>>> time.ctime(t)
'Fri Jul 27 17:01:56 2018'
```

Tabii aynı işlem tek hamlede şöyle de yapılabilirdi:

```
>>> time.ctime(time.time())
'Fri Jul 27 17:06:31 2018'
```

ctime fonksiyon parametresiz çağrılsa o andaki tarih zaman bilgisi elde edilir. Örneğin:

```
>>> time.ctime()
'Sun Aug  5 10:24:38 2018'
```

time.localtime Fonksiyonu

localtime isimli fonksiyon time fonksiyonundan elde edilen değeri parametre olarak alır ve bize o değerin tarih zaman karşılığını struct_time isimli bir sınıf nesnesi olarak verir. Örneğin:

```
>>> t = time.time()
>>> st = localtime(t)
>>> st
time.struct_time(tm_year=2018, tm_mon=7, tm_mday=27, tm_hour=17, tm_min=30, tm_sec=11,
tm_wday=4, tm_yday=208, tm_isdst=0)
```

struct_time isimli sınıfın tm_xxx biçiminde isimlendirilmiş örnek öznitelikleri (instance attribute) vardır. Bu öznitelikler ilgili tarih ve zamanın bileşenlerini bize verir. Örneğin:

```
import time

st = time.localtime(time.time())
print('{0:02d}/{1:02d}/{2:04d}'.format(st.tm_mday, st.tm_mon, st.tm_year))
```

localtime parametresiz olarak kullanılırsa o andaki tarih zaman bilgisini bize verir. struct_time yapısının örnek özniteliklerini (instance attributes) tek tek kullanan aşağıdaki gibi bir fonksiyon yazabiliriz:

```
import time

def dispStructTime(st):
    print('tm_year:', st.tm_year)
    print('tm_mon:', st.tm_mon)
    print('tm_mday:', st.tm_mday)
    print('tm_hour:', st.tm_hour)
    print('tm_min:', st.tm_min)
    print('tm_sec:', st.tm_sec)
    print('tm_wday:', st.tm_wday)
    print('tm_yday:', st.tm_yday)
    print('tm_year:', st.tm_year)
    print('tm_isdst:', st.tm_isdst)
    print("Hafanın günü:", ['Pazartesi', 'Salı', 'Çarşamba', 'Perşembe', 'Cuma', 'Cumartesi',
'Pazar'][st.tm_wday])

dispStructTime(time.localtime())
```

time modülü içerisindeki sleep isimli fonksiyon saniye cinsinden bekleme yaratmak için kullanılmaktadır. Fonksiyonun parametrik yapısı şöyledir:

```
sleep(seconds)
```

Fonksiyonun parametresi float olarak girilebilir. Örneğin:

```
import time

for i in range(10):
    print(i)
    time.sleep(0.5)
```

datetime Modülü

datetime modülünün içerisinde tarih ve zamanı tutmak için kullanılan date, time, timedelta, datetime gibi sınıflar ve bazı yardımcı fonksiyonlar vardır.

datetime.date Sınıfı

Nasıl datetime.time sınıfı zaman bilgisini tutmak için bulundurulmuşsa datetime.date sınıfı da tarih bilgisini tutmak için bulundurulmuştur. Bir datetime.date nesnesi sınıfın başlangıç metodu yoluyla aşağıdaki gibi yaratılabilir:

```
>>> d = datetime.date(2013, 7, 23)
>>> print(d)
2013-07-23
>>> type(d)
<class 'datetime.date'>
```

Yine tarihin bileşenlerini biz sınıfın day, month ve year örnek öznitelikleriyle elde edebiliriz. Örneğin:

```
import datetime

d = datetime.date(2017, 5, 23)
print('{:02d}/{:02d}/{:04d}'.format(d.day, d.month, d.year))
```

date sınıfının today isimli sınıf metodu (class method) bize o anki tarihi date sınıfı nesnesi olarak verir. Örneğin:

```
>>> d = datetime.date.today()
>>> print(d)
2018-08-05
```

date sınıfının ctime örnek metoduyla tarih bilgisini yazışal olarak elde edebiliriz. Örneğin:

```
>>> datetime.date.today().ctime()
'Fri Jul 27 00:00:00 2018'
```

Elimizde bir date nesnesi varsa date sınıfının replace metoduyla onun bazı bileşenlerini değiştirip yeni bir date nesnesi elde edebiliriz. Örneğin:

```
>>> d1 = datetime.date.today()
>>> d2 = d1.replace(year = 1999)
>>> print(d1)
2018-07-27
>>> print(d2)
1999-07-27
```

date sınıfının karşılaştırma operatör metotları olduğu için iki date nesnesini karşılaştırma operatörleriyle işleme sokabiliriz. Örneğin:

```

import datetime

d1 = datetime.date(2020, 10, 23)
d2 = datetime.date(2020, 9, 20)

if d1 > d2:
    print('d1 > d2')
elif d1 < d2:
    print('d1 < d2')
elif d1 == d2:
    print('d1 == d2')

```

date sınıfının weekday isimli metodu ilgili tarihe ilişkin günün haftanın kaçinci günü olduğunu vermektedir. (Haftanın ilk günü Pazartesidir ve 0 değerindedir.) Örneğin:

```

>>> d = datetime.date.today()
>>> ['Pazartesi', 'Salı', 'Çarşamba', 'Perşembe', 'Cuma', 'Cumartesi', 'Pazar'][d.weekday()]
'Pazartesi'

```

Sınıfın isoweekday isimli metodu weekday metodu ile aynı şeys yapmaktadır. Ancak bu metot haftanın ilk gününü Pazar kabul eder.Örneğin:

```

>>> d = datetime.date.today()
>>> ['Pazar', 'Pazartesi', 'Salı', 'Çarşamba', 'Perşembe', 'Cuma', 'Cumartesi'][d.isoweekday()]
'Pazartesi'

```

Sınıfın isocalendar isimli metodu (yıl, yılın haftası, haftanın günü) biçiminde üç elemanlı bir demete geri döner. Örneğin:

```

>>> datetime.date.today().isocalendar()
(2020, 38, 1)

```

Sınıfın isoformat isimli metodu sınıfın tuttuğu tarihi ISO 8601 formatında (yani 'YYYY-MM-DD' formatında) bir yazı olarak verir. Sınıfın __str__ metodu da bu metodun geri dönüş değeriyle geri dönmektedir. Örneğin:

```

>>> datetime.date.today().isocalendar()
(2020, 38, 1)
>>> datetime.date.today().isoformat()
'2020-09-14'
>>> print(datetime.date.today())
2020-09-14

```

Bazen yazışal bir tarih bilgisini datetime.date türüne dönüştürmek isteyebiliriz. Bunun için date sınıfın fromisoformat isimli sınıf metodu kullanılmaktadır. Örneğin:

```

import datetime

d = datetime.date.fromisoformat(input('yyyy-mm-dd formatında bir tarih giriniz:'))
print(d)

```

datetime.time Sınıfı

datetime modülünün içerisindeki time sınıfı bir zaman bilgisini saat, dakika, saniye ve mikrosaniye biçiminde bizden alarak tutar. Örneğin:

```

>>> t = datetime.time(10, 53, 47, 150000)
>>> print(t)
10:53:47.150000
>>> type(t)

```

```
<class 'datetime.time'>
```

time sınıfının hour, minute, second ve microseconds örnek öznitelikleriyle nesnede tutulan zaman bilgisinin bileşenlerini elde edebiliriz. Örneğin:

```
>>> t = datetime.time(10, 52, 34, 150000)
>>> print(t.hour, t.minute, t.second, t.microsecond)
10 52 34 150000
```

Örneğin:

```
import datetime

t = datetime.time(14, 35, 17, 500000)
print('{0:02d}:{1:02d}:{2:02d}.{3}'.format(t.hour, t.minute, t.second, t.microsecond))
```

datetime.time sınıfının da karşılaştırma operatör metotları ile iki time nesnesi karşılaştırılabilir. Örneğin:

```
import datetime

t = datetime.time(14, 35, 17, 500000)
k = datetime.time(14, 35, 18, 500000)

if t > k:
    print('t > k')
elif t < k:
    print('t < k')
else:
    print('t == k')
```

datetime.timedelta Sınıfı

datetime modülünün timedelta sınıfı zaman aralığını turmak için düşünülmüştür. Sınıfın `__init__` metodunun parametrik yapısı şöyledir:

```
datetime.timedelta(days=0, seconds=0, microseconds=0, milliseconds=0, minutes=0, hours=0, weeks=0)
```

Örneğin:

```
>>> td = datetime.timedelta(hours=3, minutes=25, seconds=15)
>>> print(td)
3:25:15
```

Nesne oluşturulurken verilen bu zaman değerlerini geri alabilmek için yalnızca üç örnek özniteliği kullanılmaktadır: days, seconds ve microseconds. Örneğin:

```
>>> td = datetime.timedelta(hours=25, minutes=2, seconds=3)
>>> td.days
1
>>> td.seconds
3723
>>> td.microseconds
0
```

Nesne oluşturulurken verilen bu zaman değerleri float türden de olabilir. Örneğin:

```
>>> td = datetime.timedelta(hours=1.5, minutes=2.5, seconds=3.5)
>>> print(td)
```

1:32:33.500000

İki timedelta nesnesi de toplanıp çıkartılabilir. Ürün timedelta türünden olacaktır. Örneğin:

```
import datetime

td1 = datetime.timedelta(hours=3, minutes=25, seconds=15)
td2 = datetime.timedelta(hours=2, minutes=50, seconds=35)

td3 = td1 - td2
print(td3)

td3 = td1 + td2
print(td3)
```

date nesnesi ile timedelta nesnesi + ve - operatörleriyle işleme sokulabilir. Bu durumda ürün olarak date elde edilir. Örneğin:

```
import datetime

today = datetime.date.today()
td = datetime.timedelta(days=4)

result = today - td
print(result)

result = today + td
print(result)
```

Ancak bir time nesnesi ile bir timedelta nesnesi de + ve - operatörleriyle işleme sokulamaz.

timedelta nesnesi bir tamsayıyla ya da bir float sayıyla çarpılıp bölünebilir. Sonuç yine timedelta nesnesi olarak elde edilecektir. İki timedelta nesnesi karşılaştırma operatörleriyle karşılaştırma işlemeye sokulabilir.

datetime.datetime Sınıfı

datetime modülünün datetime sınıfı hem tarih hem de zaman bilgisini tutar. datetime sınıfının başlangıç metodu şöyledir:

```
class datetime.datetime(year, month, day, hour=0, minute=0, second=0, microsecond=0, tzinfo=None, *, fold=0)
```

Örneğin:

```
>>> dt = datetime.datetime(2009, 12, 26, 1, 3, 7)
>>> print(dt)
2009-12-26 01:03:07
```

Bir datetime nesnesini datetime sınıfının combine isimli sınıf metoduya date ve time nesnesi vererek de oluşturabiliriz. Örneğin:

```
>>> t = datetime.time(1, 3, 7)
>>> d = datetime.date(2009, 12, 26)
>>> dt = datetime.datetime.combine(d, t)
>>> print(dt)
2009-12-26 01:03:07
```

Yine datetime sınıfının year, month, day, hour, minute, second gibi örnek öznitelikleri nesnenin içerisindeki tarih ve zamanın bileşenlerini bize verir.

`datetime` sınıfının `now` isimli sınıf metodu bize o anki tarih ve zamanı `datetime` nesnesi olarak vermektedir. Örneğin:

```
>>> datetime.datetime.now()  
datetime.datetime(2020, 9, 19, 0, 13, 40, 985585)
```

Benzer biçimde sınıfın `utcnow` isimli sınıf metodu ise o anki tarih ve zamanı UTC olarak verir. Örneğin:

```
>>> datetime.datetime.utcnow()  
datetime.datetime(2020, 9, 18, 21, 13, 57, 787950)
```

`datetime` modülündeki bu sınıfları orijinal dokümanlardan bir kez daha inceleyiniz:

<https://docs.python.org/3/library/datetime.html>

İki `datetime` nesnesi birbirinden çıkarılabilir. Sonuç `timedelta` türünden olur. Bir `datetime` nesnesiyle bir `timedelta` nesnesi de toplanıp çıkarılabilir. Bu durumda sonuç `timedelta` türündendir. Örneğin:

```
>>> dt1 = datetime(2020, 9, 19)  
>>> td = timedelta(days=3, hours=2, minutes=5)  
>>> dt2 = dt1 + td  
>>> dt2  
datetime.datetime(2020, 9, 22, 2, 5)  
>>> dt2 = dt1 - td  
>>> dt2  
datetime.datetime(2020, 9, 15, 21, 55)
```

İki `datetime` nesnesi karşılaştırma operatörleriyle işleme sokulabilmektedir.

Anahtar Notlar: Python'da import edilmiş olan modüllerin kaynak kodlarını görmek mümkündür. Bunun için PyCharm IDE'sinde ilgili ismin üzerine gelinip bağlam menüsünden "Go To/Declaration" seçilebilir. Eğer bir IDE'de çalışmıyorsak bu işlem `inspect` modülüyle de yapılabilmektedir. `inspect` modülünün `getsource` isimli sınıf metodu bize ilgili ögenin kaynak kodlarını görüntülemektedir. Örneğin:

```
import inspect  
import datetime  
  
print(inspect.getsource(datetime))
```

Ancak bubuilt-in fonksiyonlar ve sınıfların kaynak kodları görüntülenmemektedir. Çünkü bunlar yorumlayıcı içerisinde gömülüdür.

calendar Modülü

`calendar` isimli modül bazı temel takvimsel işlemleri yapan fonksiyonlara ve sınıflara sahiptir. Modülün `isleap` isimli fonksiyonu ilgili yılın artık olup olmadığını bize verir. Örneğin:

```
>>> import calendar  
>>> calendar.isleap(2000)  
True
```

`calendar` modülünün `TextCalendar` isimli sınıfı takvim işlemleri için kullanılmaktadır. Sınıfın başlangıç metodunu haftanın hangi günden başlayacağını belirtir (0 = Monday, 1 = Tuesday, 2 = Wednesday, ...). Günler için `Calendar` sınıfında tanımlanmış sembolik sabitler vardır. `TextCalendar` sınıfının `prmonth` isimli metodu ay takvimini ekrana bastırır. Örneğin:

```
import calendar  
  
cal = calendar.TextCalendar()  
cal.prmonth(2018, 8)  
  
August 2018  
Mo Tu We Th Fr Sa Su
```

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		

TextCalendar sınıfının pryear isimli örnek metodu yılın tüm aylarının takvimini ekrana (stdout dosyasına) basar.

Sınıfın formatmonth metodu ay takvimini ekrana bastırmak yerine bir str nesnesi olarak verir. Parametrik yapısı şöyledir:

```
formatmonth(theyear, themonth, w=0, l=0)
```

TextCalendar sınıfının formatyear isimli örnek metodu da yılın takvimini ekrana basmak yerine onu bir string olarak verir. Metodun parametrik yapısı şöyledir:

```
formatyear(theyear, w=2, l=1, c=6, m=3)
```

Düzenli İfadeler (Regular Expressions)

Düzenli ifadeleri "yazışal kalıpların ifade edilmesinde kullanılan küçük bir dil" olarak tanımlayabiliriz. Gerçekten de özellikle yazılıarda belli kalıpların aranması sürecinde düzenli ifadelerden sıkça faydalankmaktadır. Örneğin bir text editörde "dd/mm/yyyy" kalıbına uyan tarihleri ya da xxxx@yyyyy.com kalıbına uyan e-posta adreslerini bulmak isteyebiliriz. Bu kalıpların editörlerdeki klasik metin arama özellikleriyle bulunamayacağına dikkat ediniz. İşte düzenli ifadeler böyle kalıpların ifade edilmesini sağlayan kurallar topluluğundan oluşmaktadır. Gelişmiş pek çok kelime işlemci düzenli ifadeler yoluyla arama işlemi yapabilmektedir. Düzenli ifadeler üzerinde işlem yapan araçların kullandıkları kodlara "düzenli ifade motorları (regular expression engines)" denilmektedir. Malesef düzenli ifadelerin kurallarına ilişkin bir standart yoktur. Bu nedenle düzenli ifade motorlarının da tamamen aynı kurallara sahip olduklarını söyleyemeyiz. Ancak pek çok motor büyük ölçüde birbirlerine benzemektedir.

Düzenli ifadelerde iki tür karakter kümesi vardır: Normal karakterler ve meta karakterler. Normal karakterler kalıpta karakter olarak bulunması gereken öğelerdir. Yani kalıptaki normal bir karakter başka bir şeyi değil kendisini temsil eder. Meta karakterler ise kalıpta kendisini temsil etmeyen, özel anlamda gelen karakterlerdir. Örneğin '+' bir meta karakterdir. '+' karakterinin düzenli ifadelerde özel başka bir anlamı vardır. Bu karakter onun solundaki karakterden "bir tane ya da daha fazla bulunma" durumunu belirtir. Örneğin "ab+c" kalıbı aşağıdaki yazılarla uyusabilir:

```
abc
abbbbC
abbbbbbbbc
abbbbbbbbbbC
```

İste '+' gibi değişik anlamlara gelen pek çok meta karakter bulunmaktadır. Zaten düzenli ifade dilinin öğrenilmesi büyük ölçüde bu meta karakterlerin öğrenilmesi sürecidir.

Düzenli ifade motorlarının kullandığı tipik meta karakterler ve anlamları şunlardır:

Meta Karakterler	Anlamı
. (nokta)	'\n' dışındaki herhangi bir karakter
?	Solundaki karakterden 0 tane ya da 1 tane
*	Solundaki karakterden 0 tane ya da çok tane
+	Solundaki karakterden 1 tane ya da çok tane
{n}	Burada n bir sayıdır. Solundaki karakterden tam olarak n tane anlamına gelir.
{n,}	Burada n bir sayıdır. Solundaki karakterden tam olarak en az n tane anlamına gelir.
{n,m}	Burada n ve m birer sayıdır. Solundaki karakterden en az n tane en fazla m tane anlamına gelir.
[karakterler]	Köşeli parantez içerisindeki karakterlerden herhangi birisi anlamına gelir.
[x-y]	x ve y aralığındaki herhangi bir karakter

[^]	Köşeli parantez içerisindeki karakterlerden olmayan herhangi bir karakter
\w	Herhangi bir alfanümerik karakter
\W	Herhangi bir alfanümerik olmayan karakter
\s	Herhangi bir boşluk karakteri
\S	Herhangi bir boşluk olmayan karakter
\d	Herhangi bir sayısal karakter
\D	Herhangi bir sayısal olmayan karakter
\$	Satırın sonunun belli karakterlerle sonlanması durumu (Örneğin "kaan\$")
^	Satırın başı belli karakterlerle sonlanması durumu (Örneğin ^kaan")
(...)	Gruplama amacıyla kullanılır. Böylece bunun sağındaki metakarakterler bu grup için anlam kazanır.
	Veya anlamına gelmektedir. Örneğin "ali veli" yazı içerisindeki "ali" veya "veli" ile uyşur.

Düzenli ifadelerde kullanılan tüm meta karakterlerin bunlarla sınırlı olmadığını belirtelim. Diğer meta karakterler için Python'ın standard kütüphanesindeki dokümantasyondan faydalana bilirisiniz:

<https://docs.python.org/3/library/re.html>

Parantezlerin gruplama amacıyla kullanıldığından dikkat ediniz. Örneğin ([a-z]_){3} kalıbında {3}'a' ile 'z' arasındaki karakterlerden biri ile '_' karakterinin birleşimlerinde üç tane olacağı anlamına gelmektedir (örneğin "x_y_z_" gibi). Kalıp içerisindeki meta karakterlerin normal karakterlerle karışmaması için Python da dahil olmak üzere pek çok dil ve kütüphanede ters bölüleme yöntemi kullanılmaktadır. Örneğin + bir meta karakterdir. Solundaki karakterden bir ya da birden çok eşleşmeyi sağlar. Ancak biz bu meta karakteri ters bölü ile \+ biçiminde yazarsak bu gerçekten + karakteri anlamına gelir. Şimdi bu meta karakterlerin anımlarına ilişkin bazı örnekler verelim:

Kalıp	Neyle Uyuşur?
[_a-zA-Z]+	'_' karakterinden ve alfabetik karakterlerden oluşan karakter dizileriyle uyuşur. Köşeli parantez içerisindeki [a-z] gibi bir kalıbin 'a' ile 'z' arasındaki herhangi bir karakter anlamına geldiğini anımsayınız.
[+-]?[0-9]+\.[0-9]*	Gerçek sayı kalıplarıyla uyuşur. Örneğin "123", "123.45", "-1" gibi. (Ancak ".12" ya da ".12" gibi kalıplarla uyuşmaz)
([0-9]{1,3}\.){3}[0-9]{1,3}	Bölümleri "." ile ayrılmış IP numaralarıyla uyuşur. Örneğin "192.160.0.100" gibi. Burada parantezler gruplama amacıyla kullanılmıştır. Dolayısıyla kalıbin [0-9]{1,3} kısmı 0'dan 9'a kadar karakterlerden 1 ya da 2 ya da 3 tane olacağını belirtir. kalıbin ([0-9]{1,3}\.){3} kısmı ise üç basamağa kadar sayı ve noktaların toplamda üç tane bulunacağını belirtmektedir.
^\w*	Satırların başındaki sözcüklerle uyuşur

Python'da düzenli ifadeler "re" isimli modülle gerçekleştirilmektedir. Bu modüldeki çeşitli fonksiyonlar düzenli ifadelerle ilgili işlemler yaparlar.

re modülündeki findall fonksiyonu bütün uyuşan kalıpları bir liste olarak verir. Örneğin

```
import re
```

```
text = 'Eskişehir 26, İstanbul 34, İzmir 35'
result = re.findall(r'\d+', text)
print(result)
```

Cıktı şöyle olacaktır:

```
['26', '34', '35']
```

split fonksiyonu yazılı belli bir düzenli ifade kalibinden parçalara ayırmaktadır. Bu fonksiyonu str sınıfının split metodunun daha ileri bir biçimi olarak düşünebilirisiniz:

```
import re

text = 'Eskişehir 26, İstanbul 34, İzmir 35'
result = re.split(r', *', text)
print(result)
```

Kodun çıktısı şöyle olacaktır:

```
['Eskişehir 26', 'İstanbul 34', 'İzmir 35']
```

Örneğin:

```
import re

text = 'ali23423423veli2456564selami23487243678ayşe000012fatma'
pattern = r'\d+'

s = re.split(pattern, text)
print(s)
```

re modülündeki sub isimli fonksiyon yazı içerisinde bulunan kalıpları başka bir yazıyla yer değiştirir. Tabii orijinal yazı değiştirilmez yeni bir yazı verilmektedir. Fonksiyonun parametrik yapısı şöyledir:

```
sub(pattern, repl, string, count=0, flags=0)
```

Örneğin:

```
import re

text = 'ali23423423veli2456564selami23487243678ayşe000012fatma'
pattern = r'\d+'

s = re.sub(pattern, '-', text)
print(s)
```

Fonksiyondaki count parametresi baştan itibaren bulunan kaç kalının değiştirileceğini belirtir. Default durumda (yani count=0) durumunda bulunan tüm kalıplar yer değiştirilmektedir.

re modülündeki search fonksiyonu bir yazı içerisinde düzenli ifadelerle arama yapmaktadır. Arama başarılı olursa bulunan yere ilişkin bir match nesnesi elde edilmektedir. Fonksiyonun parametrik yapısı şöyledir:

```
re.search(pattern, string, flags=0)
```

Eğer arama sonucunda bir şey bulunamazsa fonksiyon None değeri ile geri döner. Örneğin:

```
import re

text = "ali veli 03/25/2009 selami"
m = re.search(r'\d\d/\d\d/\d\d\d\d\d\d', text)
if m:
    print('bulundu')
else:
    print('bulunmadı')
```

match nesnesinin start metodu kalının bulunduğu offset değerini end fonksiyonu da kalının bittiği offset değerinin bir fazlasını vermektedir. Örneğin:

```

import re

text = "ali veli 03/25/2009 selami"
m = re.search(r'\d\d/\d\d/\d\d\d\d', text)
if m:
    print('start: {}, end: {}'.format(m.start(), m.end()))
    s = text[m.start():m.end()]
    print(s)

else:
    print('bulunmadı')

```

span metodu kalının başlangıç ve bitiş offset'inin bir fazlasını bir demet biçiminde verir.

re modülünün match fonksiyonu da search fonksiyonu gibidir. Ancak kalıcı her zaman yazının başında arar. fullmatch fonksiyonu ise yazının tamamen kalıptan oluşup olmadığını bakmaktadır. Hem match hem de fullmatch fonksiyonları başarı durumunda match nesnesine başarısızlık durumunda None değerine geri dönmektedir.

re modülünün finditer fonksiyonu yazı içerisindeki kalıpları bulan dolaşılabilir bir nesne verir. Nesne her dolaşıldıkça match nesneleri elde edilmektedir. Örneğin:

```

import re

text = "ali veli 03/25/2009 selami 08/12/2017 ayşe fatma"
for m in re.finditer(r'\d\d/\d\d/\d\d\d\d', text):
    print(text[m.start():m.end()])

```

Aramada (yani kalıpta) parantezler kullanılırsa bunlara "grup" denilmektedir. match nesnesi içerisinde gruplar group isimli metotla elde edilirler. 0'inci grup her zaman eşleşen yazının tamamını veririm. Sonraki her grup eşleşen yazının parantez içerisindeki kısımlarını vermektedir. Örneğin:

```

import re

text = 'bugün hava çok güzel. 12/10/2009. Evet çok güzel 10/07/2009'
pattern = r'(\d\d)/(\d\d)/(\d\d\d\d)'

for m in re.finditer(pattern, text):
    print(m.group(0))
    print(m.group(1), m.group(2), m.group(3))

```

Match sınıfının __getitim__ metodu da yazılmıştır. group metodu yerine ilgili gruba köşeli parantez operatörleriyle de erişebiliriz. Örneğin:

```

import re

text = 'bugün hava çok güzel. 12/10/2009. Evet çok güzel 10/07/2009'
pattern = r'(\d\d)/(\d\d)/(\d\d\d\d)'

for m in re.finditer(pattern, text):
    print(m[0])
    print(m[1], m[2], m[3])

```

Python'da Veritabanı İşlemleri

İçeriğine hızlı bir biçimde erişilmek ve onları güncellemek için düzenlenmiş (organize edilmiş) bilgilerden oluşan dosyalara veritabanı denilmektedir. Şüphesiz veritabanları işletim sistemlerinin sunduğu dosya sistemi ile ele alınıp kontrol edilmektedir. Ancak veritabanları bilginin hızlı elde edilmesi için daha yüksek seviyeli algoritmik bir organizasyon içermektedir. Ticari uygulamaların çok büyük çoğunluğu veritabanı kullanmaktadır. Bu nedenle

veritabanı işlemleri bunların performanslarını belirleyecek en önemli etkenlerden biri durumundadır. Tabii veritabanları yalnızca ticari uygulamakrda değil aslında her türlü uygulamalarda karşımıza çıkabilmektedir.

Veritabanı işlemleri kabaca üç biçimde yapılabilmektedir:

- 1) Programcı algoritmik alt yapıya sahipse veritabanı işlemini yapan fonksiyonları, sınıfları ve metotları kendine uygun bir biçimde gerçekleştirebilir.
- 2) Programcı firmalar ya da kurumlar tarafından oluşturulmuş olan veritabanı kütüphanelerini kullanabilir. (Örneğin tarihsel açıdan bakıldığında DBVista, Btree, CTree gibi pek çok veritabanı kütüphanesi kullanılmıştır.)
- 3) Programcı veritabanı işlemlerini yapmak için oluşturulmuş ismine "Veritabanı Yönetim Sistemi (Database Management System)" denilen özel yazılımları kullanabilir. Bugün veritabanı işlemleri ağırlıklı olarak VTYS'ler kullanılarak yapılmaktadır.

Veritabanı Yönetim Sistemleri

VTYS'ler veritabanı işlemlerini yapmak için geliştirilmiş özel yazılımlardır. Tipik olarak VTYS yazılımlarının özellikleri şunlardır:

- VTYS'lerde aşağı seviyeli dosya işlemleriyle kullanıcının ilişkisi kesilmiştir. Kullanıcılar VTYS'lerle çalışırken yüksek seviyeli soyutlamalar kullanırlar. Örneğin bilgilerin hangi dosyalarda nasıl tutulduğuyla kullanıcılar ilgilenmezler.
- VTYS'ler client-server mimariye uygun olarak tasarınlırlar. Yani onlara birden fazla kişi aynı anda erişip işlem yaptırabilir.
- VTYS'ler belli bir güvenlik mekanizmasına sahiptir. Yani bunlardaki veritabanlarına erişmek için "user name", "password" gibi bilgilere sahip olmak gereklidir.
- VTYS'ler bilgilerin bozulmasına karşı dirençli biçimde tasarlanmışlardır. Örneğin elektrik kesilmesi gibi bir durumda sistem kendini onarabilmektedir.
- VTYS'ler bize ilave bazı araçlar da sunarlar. Örneğin backup-restore gibi utility'lere sahiplerdir.
- VTYS'ler işleri kolay yapmak için "yönetim konsollarına" da sahiptirler. Yani bunlar üzerinde komut satırından ya da görsel olarak işlemler yapılabilmektedir.
- VTYS'ler kullanıcıdan istekleri yüksek seviyeli deklaratif diller yoluyla almaktadır. Örneğin SQL bu amaçla kullanılan bir dildir. Biz VTYS'nin veritabanına kayıt eklemesi için bir SQL komutunu veririz. VTYS o SQL komutunu inceler ve bizim istediğimiz işlemi arka planda C/C++ ile yazılmış olan kodları çalıştırarak yapar. SQL yalnızca bizim VTYS'ye istekte bulunmamız için kullanılmaktadır. Yoksa VTYS aşağı seviyeli disk işlemlerini C/C++'ta yazılmış motor kısmıyla yapar.

Bugün için çok kullanılan ilişkisel DBMS'ler şunlardır:

- DB2 (IBM)
- Oracle (Oracle)
- Sql Server (Microsoft'un. Paralı fakat bedavaşı da var)
- MySql (Open Source, fakat Oracle tarafından yönetiliyor)
- PostgreSQL (Open Source)
- H2 (Open Source)
- SqLite (Open Source, Embedded)
- Access (Jet Engine) (Microsoft, Embedded)

Gömülü VTYS Kavramı (Embedded DBMS)

Bir VTYS'nin kendisinin kurulması zaman alan bir süreçtir. Ayrıca VTYS'ler arka planda servis olarak çalıştırıldıklarından belli bir sistem kaynağını kullanırlar. Bazı küçük ve orta ölçekli uygulamalarda bir VTYS'nin kurulması istenmeyebilir. Örneğin küçük bir rehber uygulaması için MySql gibi bir VTYS'nin hedef bilgisayara kurulması ve konfigüre edilmesi zahmetli bir süreçtir. Bu tür uygulamalarda kendisi VTYS gibi davranışın fakat aslında tek bir kütüphaneden (DLL'den oluşan) VTYS'ler kullanılmaktadır. Bunlara gömülü VTYS'ler denir. Gömülü VTYS'ler gömülü sistemlerde de yoğun olarak kullanılmaktadır. Gömülü VTYS'ler client-server biçimde çalışmazlar. Aslında bunlar yapı bakımından veritabanı kütüphanelerine benzemektedir. Ancak VTYS'lerin bazı özelliklerini barındırmaktadır. Gömülü VTYS'lerin en çok kullanıldığı SQLite'tir. Microsoft'un Jet Motoru da Windows sistemlerinde çok kullanılmaktadır. (Örneğin Access bu Jet motorunu kullanıyor. Bu yüzden bu VTYS'ye access de denilmektedir.)

MySql'in Kurulumu

MySQL'i kurmak için tek yapılacak şey server programı indirip yüklemektir. Kurulum basittir. Birtakım sorular default değerlerle geçilebilir. Ancak kurulum sırasında bizden bir root parolası istenecektir. Bu parola yetkili olarak VTYS'ye bağlanmak için gereklidir. Server programın yanı sıra bir yönetim ekranı elde etmek için ayrıca "MySQL Workbench" programı da kurulabilir. "MySQL Workbench" eskiden "MySQL GUI Tools" isimli paketteki programların birleştirilmesiyle oluşturulmuştur. Bu eski versiyonlar da hala kullanılmaktadır. "MySQL GUI Tools" paketinin de kurulmasını tavsiye ederiz.

Sql Server'in Kurulumu

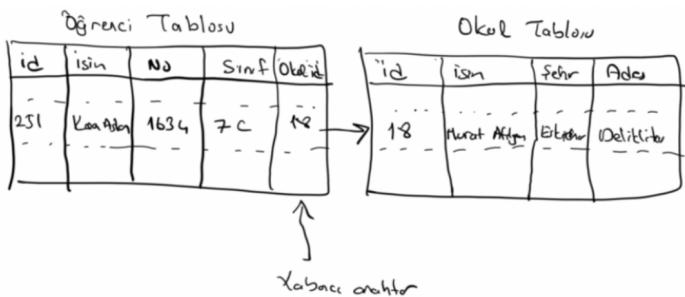
SqlServer paralı bir üründür. Fakat bunun da "Express Edition" isminde bedava bir sürümü vardır. Bu sürüm Microsoft'un sayfasından indirilip kurulabilir. Tıpkı MySql'de olduğu gibi Sql Server'da da bir yönetim programı da vardır. Buna "Sql Server Management Studio" denilmektedir. Bunun da indirilip kurulması tavsiye edilir.

SqLite'in Kurulumu

SqLite zaten tek bir DLL'den oluşmaktadır. Dolayısıyla kurulumu diye bir durum söz konusu değildir. Ancak yönetim konsolu olarak pek çok alternatif vardır. Bu yönetim konsolları SQLite'in işlemlerini yapan DLL'i de zaten indirmektedir. DSQLite için yönetim konsollarından biri "FireFox Add On" olarak çalışmaktadır. FireFox üzerinden bu hemen yüklenebilir. Diğer iki seçenek "SQLite Studio" ve "SQLite Management Studio" programlarıdır. Bunlar bedavadır ve doğrudan yüklenebilir.

İlişkisel Veritabanı Yönetim Sistemleri (Relational Database Management Systems)

Veritabanlarının gerçekleştirilebilmesinde için çeşitli modeller (paradigmalar) kullanılmaktadır. Günümüzde en çok kullanılan model ilişkisel (relational) modeldir. İlişkisel modelde veritabanı kullanıcıya tablolar biçiminde gösterilir. Bir tablo satır ve sütunlardan oluşur. Tablonun sütunlarına alan (field) satırlarına genel olarak kayıt (record) denilmektedir. Bir ilişkisel veritabanı birden fazla tablodan oluşabilir. Kullanıcı bu tablolardan verileri SQL kullanarak VTYS'den talep eder. Tablolar birbirleriyle ilişkili olabilir. Bu durumda bir tablodan diğerine geçiş yapmak için ilişkili bir alandan faydalıdır. Böyle alanlara "yabancı anahtar (foreign key)" denilmektedir. Örneğin:



Çok fazla tablodan oluşan veritabanlarında tabloların tasarımları önemli olmaktadır. Tasarımda çeşitli ilkeler vardır. Bunların en önemlisi "veri tekrarının ortadan kaldırılmasıdır". Yani bir veri bir tabloda varsa başka bir tabloda olmamalıdır. Buna "normalizasyon" da denilmektedir.

SQL Veri Türleri

SQL ISO tarafından standardize edilmiş bir dildir. Ancak VTYS'ler bu standartları desteklemekle birlikte kendilerine özgü eklentilere ve komutlara da sahip olabilmektedir. Bu nedenle örneğin MySql'deki SQL ile SqlServer'daki SQL arasında ayrıntılarda farklılıklar olabilir. SQL veri türleri tablo sütunlarını oluştururken o sütunlardaki bilginin formatını belirlemekte kullanılmaktadır. Standart SQL veri türlerinin önemli olanları şunlardır:

INTEGER: Tamsayısal bilgileri tutan bir türdür. İstenirse kaç digitlik sayıların tutulacağı da belirtilebilir.

INT: Tipik olarak 4 byte uzunluğunda işaretli tamsayı türündür.

SMALLINT: Tipik olarak 2 byte'lık işaretli tamsayı türündür.

BIGINT: Tipik olarak 8 byte uzunluğunda işaretli tamsayı türündür.

FLOAT: Tipik olarak 4 byte'lık gerçek sayı türündür.

DOUBLE: Tipik olarak 8 byte'lık gerçek sayı türündür.

TIME: Zaman bilgisini saklamak için kullanılan türdür.

DATE: Tarih bilgisini saklamak için kullanılan türdür.

DATETIME: Hem tarih hem zaman bilgisini saklamak için kullanılan türdür.

CHAR(n): n karakterli yazıyı tutmak için kullanılan türdür.

VARCHAR(n): En fazla n karakterli bir yazıyı tutmak için kullanılan türdür.

TINYTEXT: Yazışsal bilgileri tutmak için kullanılan türdür. (Tipik olarak 256 byte'a kadar)

TEXT: Yazışsal bilgileri tutmak için kullanılan türdür. (Tipik olarak 64K'ya kadar)

LONGTEXT: (Tipik olarak 4GB'ye byte'a kadar)

TINYBLOB: Binary bilgileri tutmak için kullanılan türdür. (Tipik olarak 256 byte'a kadar)

BLOB: Binary bilgileri tutmak için kullanılan türdür. (Tipik olarak 64K'ya kadar)

LONGBLOB: Binary bilgileri tutmak için kullanılan türdür. (Tipik olarak 4GB'ye byte'a kadar)

BOOLEAN: Böyle sütunlarda TRUE, FALSE biçiminde ikil değerler tutulmaktadır.

Yukarıdaki türler pek çok VTYS'de vardır. Bunların dışında başka standart SQL veri türleri de bulunmaktadır. Ayrıca yukarıda belirtildiği gibi her VTYS'de o VTYS'YE özgü diğerlerinde olmayan türler bulunuyor olabilir.

Temel SQL Komutları

Bu bölümde temel bazı SQL komutları yüzeysel olarak ele alınacaktır. SQL kolay bir dildir. Şüphesiz ne kadar iyi SQL bilinirse o kadar iyidir. Ancak biz kursumuzda temel SQL bilgisiyle işlerimizi yürüteceğiz. SQL büyük harf duyarlılığı olan bir dil değildir. Ancak geleneksel olarak anahtar sözcükler büyük harflerle书写ırlar. Veritabanı isimleri, tablo isimleri, alan isimleri vs. küçük harflerle isimlendirilmektedir. SQL komutları normal olarak ';' atomu ile sonlandırılmaktadır. Ancak pek çok VTYS komutlar ';' ile sonlandırılmasa bile komutları kabul etmektedir.

Anahtar Notlar: Bazı işlemler SQL komutlarıyla değil VTYS'ler için yazılmış yönetim programlarıyla görsel olarak da yapılabilir. Aslında bu yönetim programları arka planda yine SQL komutlarını kullanarak işlemleri yaparlar. Fakat çoğu zaman bazı işlemler için (örneğin veritabanı yaratmak tablo oluşturmak vs.) yönetim ekranlarını kullanmak daha pratiktir.

CREATE DATABASE Komutu: Bu komut veritabanı yaratmak için kullanılır. Komutun genel biçimi şöyledir:

```
CREATE DATABASE <isim>;
```

Örneğin:

```
CREATE DATABASE student;
```

USE Komutu: Belli bir veritabanı üzerinde işlemler yapmak için öncelikle onun seçilmesi gereklidir. Bu işlem USE komutuyla yapılır. Komutun genel biçimi şöyledir:

```
USE <isim>;
```

Örneğin:

```
USE student;
```

SHOW DATABASES Komutu: Bu komut VTYS'de yaratılmış olarak bulunan veritabanlarını gösterir. Komutun genel biçimi şöyledir:

```
SHOW DATABASES;
```

CREATE TABLE Komutu: Bu komut veritabanı için bir tablo yaratmak amacıyla kullanılır. Komutun genel biçimi şöyledir:

```
CREATE TABLE <isim> (<isim><tür>, <isim><tür>, <isim><tür>... );
```

Örneğin:

```
CREATE TABLE person(person_name VARCHAR(64), person_no INTEGER, person_bdate DATE);
```

Bazen yanlışlıkla (ya da kasten) aynı isimli bir tablo yaratılmak istenebilir. Bu durumda yaratım sırasında VTYS hata bildirecektir. İşte eğer tablo yalnızca yoksa yaratılmak isteniyorsa CREATE TABLE IF NOT EXISTS komutu kullanılabilir.

DROP TABLE Komutu: Bu komut tabloyu silmek için kullanılır. Genel biçimi şöyledir:

```
DROP TABLE <isim>;
```

INSERT INTO Komutu: Bu komut bir tabloya bir satır eklemek için kullanılır. Komutun genel biçimi şöyledir:

```
INSERT INTO <tablo ismi> (sütun1, sütun2, sütun3,...) VALUES (değer1, değer2, değer3,...);
```

Örneğin:

```
INSERT INTO person(person_name, person_no, person_bdate) VALUES('Abit Süzülmüş', 1234, '1954/07/02');
```

Değerler girilirken yazılar tek tırnak içerisinde belirtilmelidir.

WHERE Cümlesi: Pek çok komut bir WHERE kısmı içermektedir. WHERE cümlesi koşul belirtmek için kullanılır. Koşullar karşılaştırma operatörleriyle oluşturulur. Mantıksal operatörlerle birleştirilebilir. Örneğin:

```
WHERE yas > 20 AND dogum_yeri = 'Eskişehir';
```

LIKE operatörü joker karakterleri kullanılarak belli bir kalıba uyan yazı koşulu oluşturur. Örneğin:

```
WHERE adi_soyadi LIKE 'a%';
```

Burada adi_soyadi 'a' ile başlayanlar koşulu verilmiştir. % karakteri "geri kalanı herhangi biçimde olabilir" anlamına gelir. Örneğin:

```
WHERE adi_soyadi LIKE '%an';
```

Burada sonu 'an' ile bitenler koşulu verilmiştir. Örneğin:

```
WHERE adi_soyadi LIKE '%an%';
```

Burada içinde 'an' geçenler koşulu belirtilmiştir. LIKE operatörü default durumda ASCII karakterler için büyük harf küçük harf duyarlılığı olmayan (case insensitive) koşul oluşturmaktadır. Yani örneğin:

```
WHERE adi_sotadi LIKE 'a%'
```

koşulu adı soyadı 'a' ya da 'A' ile başlayanlar anlamına gelir. Ancak örneğin:

```
WHERE adi_soyadi LIKE '$%'
```

koşulunda yalnızca '\$' ile başlayan isimler elde edilir, '\$' ile başlayanlar elde edilmez.

WHERE koşulunda '=' operatörü her zaman büyük harf küçük harf duyarlılığı ile karşılaştırma yapmaktadır. Örneğin:

```
WHERE adi_soyadı = 'Kaan Aslan'
```

bu koşul 'KAAN ASLAN' gibi bir ismi bulmamaktadır.

BETWEEN AND "iki değer arasında" koşulunu belirtir. Örneğin:

```
WHERE person_no BETWEEN 10 AND 20;
```

IN "belli değerlerden herhangi biri olabilir" koşulunu belirtir. Örneğin:

```
WHERE person_no IN (1, 2, 3, 4, 5);
```

WHERE cümlecığının bazı detayları vardır. Kullanıldığı yerde ele alınacaktır.

DELETE FROM Komutu: Bu komut bir tablodan satır silmek için kullanılır. Genel biçimi şöyledir:

```
DELETE FROM <tablo ismi> [WHERE cümlecigi];
```

Örneğin:

```
DELETE FROM ogrenci WHERE adi_soyadi = 'Kaan Aslan' AND no = 12345;
```

Burada öğrenci tablosundan adi_soyadi 'Kaan Aslan' ve numarası 12345 olan kayıt silinecektir. Bu komut kullanılırken dikkat etmek gereklidir. Çünkü koşulu sağlayan ne kadar kayıt varsa hepsi tek hamlede silinmektedir.

UPDATE Komutu: Update komutu belli kayıtların alan bilgilerini değiştirmek amacıyla kullanılır. Örneğin ismi "Kağan" olan bir kaydı "Kaan" olarak değiştirmek isteyebiliriz. Ya da bir müşterinin bakiyesini değiştirmek isteyebiliriz. Komutun genel biçimi şöyledir:

```
UPDATE <tablo ismi> SET alan1 = değer1, alan2 = değer2, ... WHERE <koşul>;
```

Örneğin:

```
UPDATE student_info SET student_name = 'Kaan Kaplan' WHERE student_name = 'Kaan Aslan';
```

SELECT Komutu: Veritabanında belirli koşulları sağlayan kayıtlar SELECT komutuyla elde edilir. SELECT geniş bir komuttur. Genel biçimci oldukça karmaşıktır. Burada SELECT komutunun tipik kullanımlarını ele alacağız.

SELECT komutunun yalın kullanımı şöyledir:

```
SELECT <alan listesi> FROM <tablo ismi> WHERE <koşul>;
```

Örneğin:

```
SELECT student_name FROM student_info WHERE student_name LIKE 'K%'
```

Burada ismi K ile başlayan tüm kayıtların yalnızca isimleri elde edilmişdir. Birden fazla sütun aralarına ',' konularak beirtilir. Örneğin:

```
SELECT school_name, school_address FROM school_info WHERE school_name LIKE '%Eskişehir%'
```

Burada okul ismi içerisinde "Eskişehir" geçen okulların isimleri ve adresleri elde edilmişdir.

Sütun listesi yerine '*' karakteri kullanılırsa "tüm sütunlar" kastedilmiş olur. Örneğin:

```
SELECT * FROM school_info WHERE school_name LIKE '%Eskişehir%'
```

Eğer SELECT komutunda WHERE cümleciği yoksa tüm kayıtlar elde edilir. Örneğin:

```
SELECT school_name FROM school_info
```

Burada tüm okulların isimleri elde edilecektir.

SELECT komutuna ORDER BY cümleciği eklenebilir. ORDER BY anahtar sözcüklerini sütun listesi izler. Böylece ilgili kayıtlar burada belirtilen alanlara göre sıraya dizilir. Default dizilim küçükten büyüğe biçimdedir. ORDER BY cümlecığını ASC ya da DESC izleyebilir. Örneğin:

```
SELECT school_name, school_address FROM school_info ORDER BY school_name DESC;
```

ORDER BY cümleciği birden fazla alan içerebilir. Örneğin:

```
SELECT student_name, school_id FROM student_info WHERE student_name LIKE 'K%' ORDER BY student_name ASC, school_id DESC;
```

Burada sıralama öğrenci ismine göre artan sırada yapılmaktadır. Ancak aynı isimli öğrenciler varsa bunlar da kendi aralarında okul id'lerine göre büyükten küçüğe elde edilecektir.

LIMIT cümleceği belli sayıda kaydın elde edilmesi için kullanılmaktadır. Örneğin:

```
SELECT student_name, school_id FROM student_info WHERE student_name LIKE 'K%' ORDER BY student_name ASC, school_id DESC LIMIT 10;
```

Tabii bazı sorgular yalnızca SQL standardındaki komutlarla elde edilemeyebilirler. Örneğin ismi 5 karakter olan öğrencilerin listesini almak isteyelim.

```
WHERE student_name = 5
```

gibi bir koiul geçersizdir. İşte her VTYS'nin birtakım özel fonksiyonları vardır. Bu fonksiyonlar sütun isimlerini parametre olarak alıp birtakım değerler verirler. Örneğin:

```
SELECT * FROM student_info WHERE LENGTH(student_name) = 5;
```

MySQL'de LENGTH fonksiyonu sütundaki yazıların karakter uzunluğunu vermektedir. İlgili VTYS'deki içsel fonksiyonların neler olduğunu dokümanlardan öğrenebilirsiniz. Aşağıda SQLite'te kullanılan bazı özel fonksiyonları görüyorsunuz:

```
abs  
coalesce  
ifnull  
last_insert_rowid  
length  
lower  
ltrim  
max  
min  
printf  
round  
rtrim  
substr  
trim  
upper
```

Veritabanı tabloları yaratılırken belli bir sütun PRIMARY KEY olarak belirlenebilir. Genellikle PRIMARY KEY sütunları tamsayısal sütunlar olmaktadır. PRIMARY KEY kayıtlar arasında tek (unique) elemanlara sahip olmak zaorundadır. Yani aynı değere sahip b PRIMARY KEY sütununa ilişkin birden fazla kayıt bulunamaz. Bu da bazı işlemleri çeşitli bakımlardan kolaylaştırmaktadır. Eğer bir sütun tablo yaratılırken AUTOINCREMENT olarak belirlenmişse insert işlemi sırasında bu sütuna değer atama zorunluluğu yktur ve bu durumda VTYS bu sütuna önceki son değerin bir sonraki değerini otomatik olarak atar. İleride de görüleceği gibi yabancı anahtar (foreign key) sütunları bazen AUTOINCREMENT yapılmaktadır.

İlişkisel Veritabanlarında Join İşlemleri

İlişkisel veritabanlarında bire-cok (one-to-many) ilişki durumları farklı tablolar oluşturarak çözümlenmektedir. Örneğin elimizde bir Ülkelere ilişkin bilgileri tutan bir "country" tablosu olsun. Biz bu ülkelerde konuşulan dilleri de tutmak isteyelim. Bir ülkede tek bir dil konuşulmayabilir. Eğer biz "country" tablosuna bir "language" alanı eklersek oraya yalnızca tek dil yazabilirim. Bu tür durumlarda çoklu ilişki için ayrı bir tablo oluşturulur. İki tablo arasında geçişe izin veren sütunlara ise "yabancı anahtar (foreign key)" denilmektedir. Örneğin:

country Tablosu

```
country_name country_continent
```

ABD	Amerika
Belçika	Avrupa
Fransa	Avrupa

language Tablosu

```
country_name country_language
```

ABD	İngilizce
ABD	İspanyolca

Belçika	İngilizce
Belçika	Almanca
Belçika	Fransızca
Fransa	Fransızca

Burada belli bir ülkede konuşulan dilleri bulmak için language tablosu kullanılacaktır. Örneğin:

```
SELECT country_language FROM language WHERE country_name = 'ABD'
```

Örneğin öğrenci veritabanında student_info tablosu öğrenci bilgisini tutuyor olsun (öğrencini ismi, doğum tarihi, numarası vs.). Bir öğrencinin hangi okulda olduğu bu tabloda yer alabilir. Ancak o okulun da birtakım ayrıntıları varsa okullar için de school_info gibi ayrı bir tablo kullanılmalıdır. Bu durumda her okula bir id numarası (ya da isim) verilebilir. student_info tablosunda okulun yalnızca id'si ya da ismi bulunur. Öğrencinin okulu hakkında detaylı bilgi school_info tablosundan alınır. Burada okul id'si (ya da ismi) "yabancı anahtar (foreign key)" durumundadır. Örneğin:

student_info Tablosu

student_id	student_name	student_school_id
1	Kaan Aslan	1
2	Ali Serçe	2
3	Sacit Ünlü	1
...

school_info Tablosu

school_id	school_name	school_city
1	Eskişehir Atatürk Lisesi	Eskişehir
2	Tarsus Amerikan Lisesi	Mersin
...

Bire-çok ilişkinin tablolar halinde organize edilmesinin asıl nedeni veri tekrarının engellenmesidir. Yukarıdaki örnekte school_info tablosu olmasaydı okul bilgilerinin student_info tablosunda tekrarlanması gerekirdi. İşte veri tekrarına yol açmamak için verilerin tablolara bölünmesine normalizasyon (normalization) denilmektedir.

Birden fazla tablodan yabancı anahtarlar yoluyla bilgi toplama işlemeye JOIN işlemi denir. JOIN işleminin birden fazla biçimde olsa da (LEFT OUTER JOIN, RIGHT OUTER JOIN gibi) en çok kullanılan JOIN işlemi INNER JOIN işlemidir. INNER JOIN sentaksı şöyledir:

```
SELECT <alan listesi> FROM <tablo ismi> INNER JOIN <diğer tablo ismi> ON <koşul>
```

JOIN işleminde tabloların sütun isimleri aynıysa bunları ayırmak için bu isimleri tablo isimleriyle '.' operatörü ile birleştirilir. Örneğin:

```
SELECT student_info.name, school_info.name FROM student_info INNER JOIN school_info ON
student_info.school_id = school_info.id
```

Burada öğrencilerin isimleriyle onların okuduğu okulun isimleri elde edilmek istenmiştir. Yani hangi öğrencinin hangi okula gittiği bilgisi elde edilmektedir. Örneğin:

```
SELECT country.country_name, language.country_language FROM country INNER JOIN language ON
country.country_name = language.country_name
```

INNER JOIN işlemi alternatif bir sentaksla da yapılabilmektedir. Bu sentaksta FROM kısmında birden fazla tablo ismi belirtilir ve WHERE kısmında bağlantı koşulu belirtilmektedir. Bu sentaksa "implicit join syntax" denilmektedir. Örneğin yukarıdaki join işleminin eşdeğeri şöyle de yazılabilir.

```
SELECT country.country_name, language.country_language FROM country, language WHERE  
country.country_name = language.country_name
```

Burada ülkeler ve o ülkelerde konuşulan diller elde edilmektedir. Örneğin:

```
SELECT student_info.student_name, school_info.school_name FROM student_info, school_info WHERE  
student_info.school_id = school_info.school_id
```

Join işlemleri kartezyen çarpım oluşturup bu kartezyen çarpımdan eleman seçmek esasına dayanmaktadır. Örneğin student_info ve school_info tablolarının kartezyen çarpımı şu görünümdedir:

```
student_info.student_id|student_info.student_name|student_info.school_info|school_info.school_i  
d|school_info.school_name|school_info.school_city
```

1 Kaan Aslan 1 1 Eskişehir Atatürk Lisesi Eskişehir
1 Kaan Aslan 1 2 Tarsus Amerikan Lisesi Mersin
2 Ali Serçe 2 1 Eskişehir Atatürk Lisesi Eskişehir
2 Ali Serçe 2 2 Tarsus Amerikan Lisesi Mersin
3 Sabit Ünlü 1 1 Eskişehir Atatürk Lisesi Eskişehir
3 Sabit Ünlü 2 2 Tarsus Amerikan Lisesi Mersin

Şimdi aşağıdaki gibi bir join işlemi yapalım:

```
SELECT student_info.student_name, school_info.school_name FROM student_info, school_info WHERE  
student_info.school_id = school_info.school_id
```

Tabloda student_info.school_id = school_info.school_id koşulunu sağlayan satırların student_info.student_name ve school_info.school_name bilgileri elde edilmektedir:

1 <u>Kaan Aslan</u> 1 1 <u>Eskişehir Atatürk Lisesi</u> Eskişehir
2 <u>Ali Serçe</u> 2 2 <u>Tarsus Amerikan Lisesi</u> Mersin
3 <u>Sabit Ünlü</u> 1 1 <u>Eskişehir Atatürk Lisesi</u> Eskişehir

Join işleminde yine WHERE cümleciği kullanılmak zorunda değildir.

Burada student_info tablosundaki tüm satırlarla school_info tablosundaki tüm satırlar tek tek birleştirilerek bize verilecektir. Yani bize verilecek kayıt sayısı student_info ile school_info tablolarındaki kayıt sayısının çarpımı kadardır. Şimdi soruya WHERE cümlecığını ekleyelim:

İkiden fazla tablo da aynı biçimde join işlemine sokulabilir. Örneğin:

```
SELECT student_info.student_name, school_info.school_name, city_info.city_name FROM  
student_info, school_info, city_info WHERE student_info.student_id = school_info.school_id AND  
school_info.city_id = city_info.city_id
```

Burada öğrencilerin isimleri, gittiklerin okulun isimleri ve o okulun hangi şehirde olduğu bilgisi satır satır elde edilmek istenmiştir. Bu üç bilgi de üç ayrı tabloda bulunmaktadır. Bu komutun WHERE cümlecığını inceleyiniz. Aynı işlem INNER JOIN sentaksiyle şöyle de yapılabilirildi:

```
SELECT student_info.student_name, school_info.school_name, city_info.city_name FROM  
student_info INNER JOIN school_info ON student_info.school_id = school_info.school_id INNER  
JOIN city_info ON school_info.city_id = city_info.city_id
```

Burada biz yalnızca en çok kullanılan INNER JOIN isimli join işlemini ele aldık. Diğer join işlemlerini ilgili dokümanlardan inceleyebilirsiniz. Diğer join işlemleri de yine önce join işlemine sokulan tabloların kartezyen çarpımı ile elde edilen sonuç üzerinde işlemler yapılarak gerçekleştirilmektedir.

Python'da SQLite Veritabanı İşlemleri

Yukarıda da belirtildiği gibi Python belli bir versiyondan sonra sqlite VTY'sini standart kütüphanesine eklemiştir. Sqlite VTY'si ile işlemler sqlite3 isimli modülde bulunmaktadır. Dolayısıyla programcının bu modülü import etmesi gerekmektedir.

Sqlite ile Python'da çalışmanın tipik ve yalın yöntemi şöyledir:

1) Önce connect isimli fonksiyon çağrılarak VTY ile bağlantı sağlanır. connect fonksiyonunun parametrik yapısı şöyledir:

```
sqlite3.connect(database[, timeout, detect_types, isolation_level, check_same_thread, factory, cached_statements, uri])
```

Fonksiyon en basit olarak sqlite dosyasının yol ifadesini parametre olarak alır. Geri dönüş değeri olarak da bir connection nesnesi verir. Bu fonksiyonun geri döndürdüğü nesne Connection isimli bir sınıf türündendir. Default durumda eğer ilgili veritabanı dosyası yoksa boş olarak yaratılmaktadır. Örneğin:

```
conn = sqlite3.connect('student.sqlite')
```

2) Connection nesnesinden Connection sınıfının cursor isimli örnek metodu çağrılarak bir cursor nesnesi elde edilir. Cursor nesnesi de Cursor isimli bir sınıf türündendir. Örneğin:

```
cur = conn.cursor()
```

3) Cursor nesnesi ile Cursor sınıfının execute isimli örnek metodu çağrılarak SQL cümlesi VTY'ye gönderilir. Bu metodun parametresi gönderilecek SQL cümlesinin metnini almaktadır. Örneğin:

```
cur.execute("CREATE TABLE IF NOT EXISTS student(student_no INTEGER PRIMARY KEY, student_name TEXT)")  
cur.execute("INSERT INTO student VALUES(1634, 'Kaan Aslan')")
```

Kayıtlar üzerinde değişilik yapılan SQL cümlelerinin veritabanına yansıtılması için connection nesnesi ile commit metodunun yapılması gereklidir. Örneğin:

```
conn.commit()
```

execute metodu bize yine cursor nesnesinin kendisini geri döndürmektedir.

4) İşlemler bitince connection nesnesi Connection sınıfının close isimli örnek metodu çağrılarak kapatılmalıdır. Örneğin:

```
conn.close()
```

Şimdi bu işlemleri bir arada bir kodla geerekleştirelim:

```
import sqlite3  
  
conn = sqlite3.connect('student.sqlite')  
cur = conn.cursor()  
cur.execute("CREATE TABLE IF NOT EXISTS student(student_no INTEGER PRIMARY KEY, student_name TEXT)")  
cur.execute("INSERT INTO student VALUES(1634, 'Kaan Aslan')")  
conn.commit()  
conn.close()
```

Veritabanı ile ilgili herhangi bir hatalı işlem exception'a yol açmaktadır. Örneğin execute metodunda biz SQL sentaks hatası yaparsak execute bir exception fırlatır. Bu durumda kodumuzun exception kontrolü içerisinde çalıştırılması

uygun olur. sqlite3 modülündeki metodlar birkaç tür ile raise işlemi yapabilmektedir. Ancak bu exception türlerinin hepsi sqlite3.Error sınıfından türetilmiştir. Bu durumda biz bu sınıfı kullanarak tüm sqlite3 exception'larını yakalayabiliriz. Örneğin:

```
import sqlite3

conn = None
try:
    conn = sqlite3.connect('student.sqlite')
    cur = conn.cursor()
    cur.execute("CREATE TABLE IF NOT EXISTS student(student_no INTEGER PRIMARY KEY,
student_name TEXT)")
    cur.execute("INSERT INTO student VALUES(1634, 'Kaan Aslan')")
    conn.commit()
except sqlite3.Error as e:
    print('Error', e)
finally:
    if conn:
        conn.close()
```

Connection sınıfı bağlam yönetim protokolünü desteklediği için aynı işlemleri şöyle deyapabilirdik:

```
import sqlite3

conn = None
try:
    with sqlite3.connect('student.sqlite') as conn:
        cur = conn.cursor()
        cur.execute("CREATE TABLE IF NOT EXISTS student(student_no INTEGER PRIMARY KEY,
student_name TEXT)")
        cur.execute("INSERT INTO student VALUES(1345, Kaan Aslan')")
        conn.commit()
except sqlite3.Error as e:
    print('Error', e)
```

Şimdi yukarıda eklemiş olduğumuz kaydı güncelleymeli:

```
import sqlite3

conn = None
try:
    conn= sqlite3.connect('student.sqlite')
    cur = conn.cursor()
    cur.execute("UPDATE student SET student_no = 1789 WHERE student_no = 1634")
    conn.commit()
except sqlite3.Error as e:
    print('Error', e)
finally:
    if conn:
        conn.close()
```

Bazen işleme sokulacak bilgiler statik olarak değil dinamik olarak elde edilmiş olabilir. Bu durumda ilgili SQL cümlesinin yazışal biçimde oluşturulması gereklidir. Tabii ileride ele alınacağı gibi parametrik kullanım da tercih edilebilir. Aşağıdaki örnekte klavyeden ismi ve numarası alınan öğrenci veritabanına eklenmiştir:

```
import sqlite3

conn = None
try:
    conn= sqlite3.connect('student.sqlite')
    cur = conn.cursor()
```

```

no = int(input('Eklenecek öğrencinin numarasını giriniz:'))
name = input('Eklenecek öğrencinin adını soyadını giriniz:')

cur.execute("INSERT INTO student(student_no, student_name) VALUES({},'{}')".format(no,
name))
conn.commit()
except sqlite3.Error as e:
    print('Error', e)
finally:
    if conn:
        conn.close()

```

Şimdi de SELECT cümlesi ile kayıtların nasıl elde edileceği üzerinde duralım. SELECT cümlesi cursor nesnesinin execute fonksiyonu ile uygulandıktan sonra Cursor sınıfının fetchall örnek metodu ile tüm kayıtlar bir demet listesi olarak elde edilebilmektedir. Demet listesinin elemanı olan demetler select edilen sütun bilgilerinden oluşmaktadır. Örneğin biz student_info veritabanında öğrenci bilgilerini çekelim:

```

import sqlite3

conn = None
try:
    conn= sqlite3.connect('student.sqlite')
    cur = conn.cursor()

    cur.execute("SELECT * FROM student")
    result = cur.fetchall()
    print(result)

except sqlite3.Error as e:
    print('Error', e)
finally:
    if conn:
        conn.close()

```

Programın çıktı şöyledir:

```
[(245, 'Sacit Bulut'), (467, 'Necati Ergin'), (1345, 'Ali Serçe'), (1789, 'Kaan Aslan')]
```

Tabii biz bu listeyi for döngüsü ile dolaşıp uygun elemanları ugın formatta yazdırabiliriz. Örneğin:

```

import sqlite3

conn = None
try:
    conn= sqlite3.connect('student.sqlite')
    cur = conn.cursor()

    cur.execute("SELECT * FROM student")
    print(f"{'No':<10}{''Adı Soyadı'}\n")
    for no, name in cur.fetchall():
        print(f'{no:<10}{name}')

except sqlite3.Error as e:
    print('Error', e)
finally:
    if conn:
        conn.close()

```

Programın çıktısı şöyledir:

No	Adı Soyadı
245	Sacit Bulut
467	Necati Ergin
1345	Ali Serçe
1789	Kaan Aslan

Aslında cursor nesnesinin kendisi de dolaşılabilir (iterable) bir nesnedir. Dolayıyla biz fetchall yapmadan doğrudan cursor nesnesini de for döngüsü ile dolaşabiliriz. Örneğin:

```
import sqlite3

conn = None
try:
    conn= sqlite3.connect('student.sqlite')
    cur = conn.cursor()

    cur.execute("SELECT * FROM student")
    print(f"{'No':<10}{'Adı Soyadı'}\n")
    for no, name in cur:
        print(f'{no:<10}{name}')

except sqlite3.Error as e:
    print('Error', e)
finally:
    if conn:
        conn.close()
```

Cursor sınıfının fetchone isimli örnek metodu her defasında yalnızca sıradaki kaydı bize demet olarak verir. Kayıt listesinin sonuna gelindiğinde fetchone None değerine geri dönmektedir. Örneğin:

```
import sqlite3

conn = None
try:
    conn= sqlite3.connect('student.sqlite')
    cur = conn.cursor()

    cur.execute("SELECT * FROM student")
    print(f"{'No':<10}{'Adı Soyadı'}\n")
    while True:
        result = cur.fetchone()
        if not result:
            break
        print(f'{result[0]:<10}{result[1]}')


except sqlite3.Error as e:
    print('Error', e)
finally:
    if conn:
        conn.close()
```

Diğer bir fetch metodu da fetchmany isimli metottur. Bu metot en fazla parametreyle girdiğimiz miktarda kaydı bize bir demet listesi olarak verir. Yine kayıt kalmadığında bu metot da None değerine geri dönmektederi. Örneğin:

```
import sqlite3

conn = None
try:
    conn = sqlite3.connect('SampleDatabases/student2.sqlite')
    cur = conn.cursor()
```

```

cur.execute("SELECT * FROM student")
print(f"{'No':<10}{ 'Adı Soyadı'}\n")
while True:
    result = cur.fetchmany(5)
    if not result:
        break
    for no, name in result:
        print(f'{no:<10}{name}')

except sqlite3.Error as e:
    print('Error', e)
finally:
    if conn:
        conn.close()

```

Pekiyi fetchall, fetchone ve fetchmany metodları arasındaki farklılıklar nelerdir? VTYS'lerden sorgulama yapıldığında kütüphaneler VTYS'den kayıtları parça parça çekebilmektedir. İşte fetchall ile tek hamlede bütün kaydın çekilmesi bazı durumlarda hem zaman kaybına hem de büyük miktarda bellek kullanımına yol açabilmektedir. Bu nedenle kayıt sayısı çok yüksekse onların fetchone ile teker teker ya da fetchmany ile parça parça elde edilmesi uygun olabilmektedir.

execute metodunda diğer framework'lerde olduğu gibi yer tutucular kullanılabilmektedir. Yer tutucu olarak ? ya da "isim:" kullanılabilir. Eğer yer tutucu kullanılmışsa buna karşılık execute metodunda SQL cümlesinden sonraki argümandan bu yer tutucularla eşleşen bir demet girmek gereklidir. Örneğin:

```
cur.execute("INSERT INTO student(student_no, student_name) VALUES(?, ?)", (no, name))
```

Burada ilk ? no ile ikinci ? ise name ile eşleşmiştir. Yani adeta bu soru işaretlerinin yerini no ve name nesnelerinin içerisindeki değerler almış gibi bir etki oluşturmaktadır. Örneğin:

```

import sqlite3

conn = None
try:
    conn= sqlite3.connect('student.sqlite')
    cur = conn.cursor()

    no = int(input('Eklenecek öğrencinin numarasını giriniz:'))
    name = input('Eklenecek öğrencinin adını soyadını giriniz:')

    cur.execute("INSERT INTO student(student_no, student_name) VALUES(?, ?)", (no, name))
    conn.commit()
except sqlite3.Error as e:
    print('Error', e)
finally:
    if conn:
        conn.close()

```

Yer tutucu olarak '?' yerine ":isim" de kullanılabilir. Bu durumda bu isimlerin ve ona karşı gelen değerlerin bir sözlük nesnesi ile execute metoduna verilmesi gereklidir. Sözlüğün anahtarı ":isim" yer tutucusundaki "isim", değeri de onun yerine yerleştirilecek değeri belirtir. Örneğin:

```
cur.execute("INSERT INTO student(student_no, student_name) VALUES(:no, :name)", {'no': no, 'name': name})
```

Burada :no yer tutucusu yerine sözlükteki 'no' anahtarının değeri :name yer tutucusu yerine ise sözlükteki 'name' anahtarının değeri yerleştirilecektir. Örneğin:

```

import sqlite3

conn = None
try:
    conn= sqlite3.connect('student.sqlite')
    cur = conn.cursor()

    no = int(input('Eklenenek öğrencinin numarasını giriniz:'))
    name = input('Eklenenek öğrencinin adını soyadını giriniz:')

    cur.execute("INSERT INTO student(student_no, student_name) VALUES(:no, :name)", {'no': no,
    'name': name})
    conn.commit()
except sqlite3.Error as e:
    print('Error', e)
finally:
    if conn:
        conn.close()

```

Biz şimdiye kadar önce connection nesnesinden cursor nesnesini elde ettik ve bu cursor nesnesi ile execute işlemi uyguladık. Aslında doğrudan connection sınıfının da bir execute örnek metodu vardır. Bu metot zaten kendi içerisinde cursor nesnesini elde edip onunla execute işlemi yapar. Connection sınıfının bu bahsettiğimiz execute metodu bize cursor nesnesini vermektedir. Yani biz işlemlere bu execute metodunun yarattığı cursor nesnesi ile devam edebiliriz. Örneğin:

```

import sqlite3

conn = None
try:
    conn= sqlite3.connect('student.sqlite')
    cur = conn.execute("SELECT * FROM student")

    print(f"{'No':<10}{'Adı Soyadı'}\n")
    for no, name in cur:
        print(f'{no:<10}{name}')

except sqlite3.Error as e:
    print('Error', e)
finally:
    if conn:
        conn.close()

```

Şimdi de veritabanı üzerinde çeşitli işlemler yapan küçük bir program yazalım:

```

import sqlite3

def get_option():
    print('1) Kayıt Ekle')
    print('2) Kayıt Listele')
    print('3) Kayıt sil')
    print('4) Çık')
    while True:
        try:
            result = int(input('\nSeçiminiz:'))
            if result >= 1 and result <= 4:
                break
            print('Giriş geçersiz!')
        except:
            print('Giriş geçersiz!')
    return result

def add_record(cur):

```

```

while True:
    try:
        no = int(input('No:'))
        break
    except:
        print('Numara geçersiz! Yeniden Numara giriniz:')

name = input('Adı Soyadı:')
try:
    cur.execute("INSERT INTO student(student_no, student_name) VALUES(?, ?)", (no, name))
    cur.connection.commit()
    print('Kayıt başarıyla eklendi...\n')
except:
    print('Ekleme işlemi başarısız!\n')

def list_record(cur):
    cond = input('Koşul cümlesini giriniz:')
    sql = 'SELECT student_no, student_name FROM student '
    if cond != '':
        sql += 'WHERE {}'.format(cond)
    try:
        cur.execute(sql)
        print('\n-----')
        print(f"{'No':<10}{'Adı Soyadı'}\n")
        for no, name in cur:
            print(f'{no:<10}{name}')
        print('-----\n')
    except:
        print('Listeleme işlemi başarısız!')

def delete_record(cur):
    cond = input('Koşul cümlesini giriniz:')
    sql = 'DELETE FROM student WHERE {}'.format(cond)
    print(sql)

    try:
        cur.execute(sql)
        cur.connection.commit()
        if cur.rowcount > 0:
            print('{} kayıt silindi\n'.format(cur.rowcount))
        else:
            print('Herhangi bir kayıt silinmedi!\n')
    except:
        print('Kayıt silinmedi!..\n')

def main():
    try:
        conn = sqlite3.connect('student.sqlite')
        cur = conn.cursor()

        while True:
            option = get_option()
            if option == 4:
                break
            {1: add_record, 2: list_record, 3: delete_record}[option](cur)

    finally:
        conn.close()

main()

```

Son eklenen kayda ilişkin (yani INSERT INTO komutuyla son eklenen autoincrement alana ilişkin) autoincrement alan değeri Cursor sınıfının lastrowid örnek özniteliği ile elde edilebilir. SQLite'ta PRIMARY KEY sütunu default olarak zaten "autoincrement" durumdadır.

Daha önce de belirttiğimiz gibi her ne kadar SQLite SQL standartlarını destekliyorsa da SQLite'in aslında standart SQL türlerinden daha az sütun türü vardır. Yani bazı türler SQLite'ta aslında başka bir tür olarak ifade edilmektedir.

Örneğin SQLite'ta VARCHAR alanında TEXT alan olarak içsel biçimde ifade edilmektedir. SQLite'taki içsel sütun türleri ile bunun Python karşılıkları şöyledir:

SQLite type	Python type
NULL	None
INTEGER	int
REAL	float
TEXT	depends on text_factory , str by default
BLOB	bytes

Insert, update işlemlerinden sonra yapılan değişikliklerin veritabanına yansıtılması için commit işleminin yapılması gerektiğini daha önce belirtmiştim. Peki bu commit işleminin anlamı nedir? Commit işlemi veritabanlarında transaction oluşturmak için kullanılmaktadır. Transaction terimi veritabanlarında bir grup işlemin kesintisiz tek bir işlememiş gibi işletilmesi anlamına gelir. Eğer transaction kavramı olmasaydı yani yapılan değişiklikler bir bütün olarak değil parça parça yapılsaydı veritabanının tutarlılığı ve bütünlüğü bozulabilirdi. Örneğin bir programın iki ayrı tabloda birbirleriyle ilişkili iki kaydı güncellemek istediğini düşünelim. Birinci tablo için UPDATE sql cümlesi ile güncelleme işlemi yapıldığında daha program ikinci tabloda güncelleme yapamadan başka bir program daha hızlı davranıp bu iki kaydı güncelleyebilir. Bu durumda ilk program diğer kaydı güncellediğinde iki tablodaki kayıt arasında tutarsızlık oluşabilir. Bunu engellemenin tek yolu bu iki işlemin başka bir program araya girmeden sanki tek bir işlem gibi "atomik" bir biçimde yapılmasını sağlamaktır. İşte commit işleminden önce uygulanan SQL cümleleri gerçek anlamda veritabanına yansıtılmamaktadır. Commit işlemiyle en son commit'ten bu yana yapılan işlemler bir bütün olarak atomik bir biçimde veritabanına yansıtılmaktadır. Örneğin:

```
cur.execute('UPDATE .....')
cur.execute('UPDATE .....')
cur.commit()
```

Bazen bir grup işlem yapılırken aradaki bir işlemde sorun çıkabilir. Bu durumda her ne kadar işlemler henüz commit ile tablolara yansıtılmamış olsa da bunların geçersiz hale getirilmeleri gerekmektedir. İşte bu geçersiz hale getirme işlemine veritabanlarında "rollback" denilmektedir. Rollback işlemi Python'da Connection sınıfının rollback metoduyla yapılmaktadır. Örneğin:

```
try:
    ...
    cur.execute('....')
    cur.execute('....')
    cur.execute('....')
    ...
except sqlite3.Error:
    cur.rollback()
```

Biz şimdide kadar SQL komutlarını Cursor sınıfının execute metoduna yaptık. Cursor sınıfının execute metodunda biz yalnızca tek bir SQL cümlesini uygulayabiliyoruz. Halbuki Cursor sınıfının executemany isimli bir metodu daha vardır. İşte executemany birden fazla cümlenin uygulanmasına olanak sağlayan bir metottur. Ancak executemany'de bu işlem aralarına ';' konulmuş ayrı SQL cümleleri ile değil parametrik olarak yapılmaktadır. Yani executemany'de yine tek bir SQL cümlesi bulunur. Ancak bu cümle "?" ya da ":isim" biçiminde yer tutucularla oluşturulmuş olmalıdır. executemany metodunun ikinci parametresi tipik olarak demet listesi olur (aslında biribirini içeren iki dolaşılabilir

nesne de kullanılabilir. executemany bu listedeki demetleri yer tutucularla eşleştirerek birden fazla kez verilen SQL cümlesini çalıştırır. Örneğin:

```
cur.executemany("INSERT INTO student VALUES(?, ?)", [(251, 'Birsen Saban'), (252, 'Vicdan Özer')])
```

Burada iki kez INSERT INTO cümlesi farklı değerlerle çalıştırılacaktır. executemany metodu zaten elimizde toplu halde bir bilgi varsa tercih edilmektedir. Örneğin bir csv dosyasının içeriğini okuyup tek hamlede onu executemany ile veritabanına ekleyebiliriz:

```
result = readcsv('test.csv')
cur.executemany("INSERT INTO student VALUES(?, ?)", result)
```

executemany metodunda yer tutucu olarak ":isim" tekniği de kullanılabilir. Bu durumda ikinci parametre tipik olarak bir sözlük dizisi olmalıdır. Örneğin:

```
cur.executemany("INSERT INTO student VALUES(:no, :name)", [{"no": 253, "name": 'George Harrison'}, {"no": 254, "name": 'Ringo Starr'}])
```

executemany yukarıda gördüğünüz gibi tek bir SQL cümlesinin birden fazla veri grubu için yinelemeli bir biçimde çalıştırılmasını sağlıyordu. Fakat bazen gerçekten birden fazla farklı SQL cümlesinin çalıştırılmasını isteyebiliriz. İşte bunun için executescript metodu kullanılmaktadır. Bu metotta yer tutucular kullanılamaz. Birden fazla SQL cümleleri aralarına ';' getirilerek yazılır. Örneğin:

```
import sqlite3

conn = None
try:
    conn= sqlite3.connect('student.sqlite')
    cur = conn.cursor()

    cur.executescript("""
        INSERT INTO student VALUES(623, 'Sami Ercan');
        INSERT INTO student VALUES(854, 'İhsan Derman');
        INSERT INTO student VALUES(778, 'Necmiye Artar');
    """)
    conn.commit()

except sqlite3.Error as e:
    print('Error', e)
finally:
    if conn:
        conn.close()
```

executescript metodu parametrik biçimde kullanılamamaktadır. executemany ve executescript metodları aynı zamanda Connection sınıflarının içerisine de yerleştirilmiştir. Bunlar yine kendi içinde bir Cursor nesnesi yaratıp aslında işlemleri bu cursor nesnesinin executemany ve executescript metodlarını kullanarak gerçekleştirmektedir. Bu metodlar yine cursor nesneleriyle geri dönmektedir.

Bazen yabancı anahtar eşliğinde birden fazla tabloya kayıt eklemek gerekebilir. Tipik olarak bunun için INSERT INTO komutunda aynı zamanda sorgulama da yapılmaktadır. Örneğin student veri tabanında "student" ve "school" isimli iki tablo olsun. "student" tablosu öğrencilerin bilgilerini, school tablosu ise okulların bilgilerini tutuyor olsun. İki tablo arasındaki bağlantının school_id isimli yabancı anahtarla yapıldığını varsayıyalım:

Student Tablosu

student_no	student_name	school_id
123	Kaan Aslan	1
257	Ayşe Er	2
189	Necati Erhan	1
...

School Tablosu

school_id	school_name
1	Şehrenem Lisesi
2	Üsküdar Lisesi
...	...

Bu tür durumlarda veri girişi yapılırken bunların ayrı ayrı yapılması (ayrı menülerden ya da ayrı pencerelerden) uygun olur. Yani veriyi giren kişi önce okulları oluşturup sonra öğrencileri kaydedebilir. Böylece öğrenci kaydedilirken okullar oluşturulmuş durumda olur. Tabii bu iki işlem birlikte de yapılabilir. Örneğin:

```

import sqlite3

conn = None
try:
    conn = sqlite3.connect('student.sqlite')
    cur = conn.cursor()
    cur.executescript("""
        CREATE TABLE IF NOT EXISTS student(student_no INTEGER PRIMARY KEY, student_name
VARCHAR(64), school_id INTEGER);
        CREATE TABLE IF NOT EXISTS school(school_id INTEGER PRIMARY KEY AUTOINCREMENT,
school_name VARCHAR(64));
    """
)

    conn.commit()

    no = int(input('Öğrencinin numarasını giriniz:'))
    student_name = input('Öğrencinin adını ve soyadını giriniz: ')
    school_name = input('Okulun adını giriniz: ')

    cur.execute("INSERT INTO school(school_name) VALUES(?)", (school_name,))
    cur.execute("INSERT INTO student(student_no, student_name, school_id) VALUES(?, ?, (SELECT
school_id FROM school WHERE school_name = ?))", (no, student_name, school_name))

    conn.commit()

except sqlite3.Error as e:
    print('Error', e)
finally:
    if conn:
        conn.close()

```

Burada ikinci INSERT işleminin VALUES kısmında SELECT işleminin yapıldığına dikkat ediniz. Böylece örnekte önce okul bilgisi school tablosuna insert edilmiş daha sonra da buradaki school_id değeri kullanılarak öğrenci bilgileri student tablosuna insert edilmiştir. Ancak bu örnekte şöyle bir kusur vardır: Burada aynı okul birden fazla kez school tablosuna insert edilebilmektedir. Peki bunu nasıl engelleyebiliriz? İlk akla gelen yöntem önce bir SELECT sorgulaması yapıp okulun daha önce school tablosuna insert edilmiş olup olmadığını bakmak, eğer okulbu tabloya insert edilmemişse gerçekten insert işlemi yapmaktır. Buradaki işlem biraz uzun olduğu için VTYS'lerde değişik biçimlerde bu işlemi içselleştirmiştir. Bazı VTYS'lerde SQL INSERT INTO komutuna WHERE ... NOT EXISTS gibi koşullar getirilebilmektedir. Ancak Sqlite bunun yerine UNIQUE sütun kavramını kullanmaktadır. Sqlite'ta bir sütun UNIQUE yapılrsa eğer INSERT INTO yerine INSERT OR IGNORE TO kullanılrsa zaten ilgili kayıt daha önce varsa yeniden insert işlemi yapılmamaktadır. Yukarıdaki kodu Sqlite için şöyle revize edebiliriz:

```

import sqlite3

conn = None
try:
    conn = sqlite3.connect('student.sqlite')
    cur = conn.cursor()
    cur.executescript("""
        CREATE TABLE IF NOT EXISTS student(student_no INTEGER PRIMARY KEY, student_name
        VARCHAR(64), school_id INTEGER);
        CREATE TABLE IF NOT EXISTS school(school_id INTEGER PRIMARY KEY AUTOINCREMENT,
        school_name VARCHAR(64), UNIQUE(school_name));
    """
)

    conn.commit()

    no = int(input('Öğrencinin numarasını giriniz:'))
    student_name = input('Öğrencinin adını ve soyadını giriniz:')
    school_name = input('Okulun adını giriniz:')

    cur.execute("INSERT OR IGNORE INTO school(school_name) VALUES(?)", (school_name,))
    cur.execute("INSERT INTO student(student_no, student_name, school_id) VALUES(?, ?, (SELECT
    school_id FROM school WHERE school_name = ?))", (no, student_name, school_name))

    conn.commit()

except sqlite3.Error as e:
    print('Error', e)
finally:
    if conn:
        conn.close()

```

Sqlite'ta "bellek veritabanı (memory database)" diye isimlendirilen bir veritabanı çeşidi de vardır. Bellek veritabanları veritabanı ismi için ":memory;" kullanılarak yaratılırlar. Bunlar tamamen ana bellekte oluşturulmaktadır. Dolayısıyla da program sonlandığında bu bilgiler (eğer başka bir yere yazılmamışsa) yok edilirler. Bellek veritabanları size ilk bakışta "gereksiz" gibi gelebilir. Ancak uzun süreli çalışan bazı programlar verileri de zaten saklama niyetleri yoksa bu özelliği kullanabilmektedir. Şüphesiz bellek veritabanları disk tabanlı gerçek veritabanlarına göre çok daha hızlı çalışmaktadır.

Sqlite'ta tarih ve zaman için özel bir veri türü bulunmamıştır. Ama Sqlite SQL standartlarını desteklemek için DATE, TIME ve DATETIME gibi sütun türlerini kabul etmektedir. Sqlite'ta tarih ve zaman bilgileri aslında arka planda tamamen TEXT, INTEGER ya da REAL türleriyle turulmaktadır. Başka bir deyişle biz aslında bir tarih bilgisini "yyyy-aa-gg" biçiminde bir yazı gibi bir sütunda saklayabiliriz. Sonra o sütundan bilgileri alarak yeniden onu tarih bilgisine dönüştürebiliriz. Ya da örneğin tarih ve zaman bilgisini time modülündeki 01/01/1970'ten geçen saniye sayısını olarak REAL bir alanda tutabiliriz. Örneğin:

```

import sqlite3

conn = None
try:
    conn = sqlite3.connect(':memory:')
    cur = conn.cursor()
    cur.execute("CREATE TABLE student(student_no INTEGER, student_name VARCHAR(64),
    student_bdate DATE)")

    cur.execute("INSERT INTO student VALUES(123, 'Hasan Bal', '2005-12-23')")
    cur.execute("INSERT INTO student VALUES(456, 'Sadık Dursun', '2004-11-22')")
    cur.execute("INSERT INTO student VALUES(768, 'Ayşe Er', '2003-08-17')")
    conn.commit()

    cur.execute("SELECT * FROM student")

```

```

for no, name, bdate in cur:
    print(no, name, bdate, sep=', ')

except sqlite3.Error as e:
    print('Error', e)
finally:
    if conn:
        conn.close()

```

Bu örnekte tarih bilgisi veritabanına tamamen yazışal biçimde girilmiştir. Daha önceden de belirttiğimiz gibi aslında Sqlite'in DATE, TIME ve DATETIME biçiminde özel türleri yoktur. Bunlar TEXT türüne karşılık gelmektedir. Tabii biz yazışal olarak sakladığımız tarihi yeniden datetime müdünlükteki date sınıfının fromisoformat metodu ile datetime.date türüne dönüştürebiliriz. Örneğin:

```

...
cur.execute("SELECT * FROM student")

import datetime

days = ['Pazartesi', 'Salı', 'Çarşamba', 'Perşembe', 'Cuma', 'Cumartesi', 'Pazar']

for no, name, bdate in cur:
    date = datetime.date.fromisoformat(bdate)
    print(no, name, date, days[date.weekday()], sep=', ')
...

```

Eğer execute ya da executemany metodlarında yer tutucu kullanıyorsak bu duurmda DATE, TIME, DATETIME alanlar için datetime.date, datetimetime ve datetime.datetime nesnelerini girebiliriz. Örneğin:

```

import sqlite3
import datetime

conn = None
try:
    conn = sqlite3.connect(':memory:')
    cur = conn.cursor()
    cur.execute("CREATE TABLE student(student_no INTEGER, student_name VARCHAR(64), student_bdate DATE)")

    while True:
        no = int(input('Öğrenci numarasını giriniz:'))
        if not no:
            break
        name = input('Öğrencinin adını ve soyadını giriniz:')
        bdate = datetime.date.fromisoformat(input('Öğrencinin doğu tarihini ISO formatında giriniz:'))
        cur.execute("INSERT INTO student VALUES(?, ?, ?)", (no, name, bdate))

    cur.execute("SELECT * FROM student")
    days = ['Pazartesi', 'Salı', 'Çarşamba', 'Perşembe', 'Cuma', 'Cumartesi', 'Pazar']

    for no, name, bdate in cur:
        date = datetime.date.fromisoformat(bdate)
        print(no, name, date, days[date.weekday()], sep=', ')

except sqlite3.Error as e:
    print('Error', e)
finally:
    if conn:
        conn.close()

```

Bazen programcı bu ratih ve zaman bilgileri üzerinde sorgulama da yapmak isteyebilir. Bu durumda Sqlite'ın tarih ve zaman fonksiyonlarından faydalanaılmalıdır. Örneğin 01/01/2010'dan sonra doğmuş kişileri SELECT cümlesiyle şöyle elde edebiliriz:

```
cur.execute("SELECT * FROM student WHERE date(student_bdate) > date('2010-01-01')")
```

Eksikler: text_factory, row_factory, conn.description, blob alanlar

```
import sqlite3

conn = None
try:
    conn = sqlite3.connect('student4.sqlite')
    cur = conn.cursor()

    cur.execute("CREATE TABLE IF NOT EXISTS student(student_no INTEGER PRIMARY KEY,
student_name VARCHAR(64), student_photo BLOB)")

    while True:
        no = int(input('Öğrencinin numarasını giriniz:'))
        if not no:
            break
        name= input('Öğrencinin adını ve soyadını giriniz:')
        path = input('Öğrenci fotoğrafına ilişkin dosyanın yol ifadesini giriniz:')

        with open(path, 'rb') as f:
            photo_data = f.read()
        cur.execute("INSERT INTO student VALUES(?, ?, ?)", (no, name, photo_data))

        conn.commit()

    cur.execute("SELECT * FROM student")

    import PIL
    import io
    import IPython

    for no, name, photo in cur:
        print('{}, {}'.format(name, no))

        bio = io.BytesIO(photo)
        image = photo_data = PIL.Image.open(bio)
        image.thumbnail((200, 300))
        IPython.display.display(image)
        print()

except sqlite3.Error as e:
    print('Error', e)
finally:
    if conn:
        conn.close()
```

Pyton'da MySQL Kullanımı

MySQL oldukça yaygın kullanılan bir VTYS'dir. Açık kaynak kodlu ve ticari uygulamaların büyük çoğunluğu bunu kullanmaktadır. MySQL client-server çalışan gerçek bir VTYS'dir. Yukarıda da belirtildiği gibi SQLite Python'ın belli bir sürümünden sonra standart kütüphaneye eklenmiştir. Ancak MySQL ile işlemler yapan modüller Python'ın standart kütüphanesinde bulunmamaktadır.

Yerel makinemize Windows'ta MySQL kurmak oldukça kolaydır. Kurulum sırasında bizden root kullanıcı için bir parola istenecektir. Server kurulumunun yanı sıra "MySQL Workbench" denilen GUI aracının ve bunun eski biçimi olan "MySQL GUI Tools" araçlarının kurulması tavsiye edilir. Windows için tüm kurulumların Inter adresleri şöyledir:

MySQL Server: <https://dev.mysql.com/downloads/mysql/>

MySQL Workbench: <https://dev.mysql.com/downloads/workbench/>

MySQL GUI Tools (Eski araçlar): <https://downloads.mysql.com/archives/gui/>

Mademki MySQL Python'ın standart kütüphanelerinde bulunmuyor o halde bizim ona ilişkin Python modüllerini kendi makinemize yüklememiz gereklidir. Geleneksel olarak bu yüklemelere veritabanı dünyasında "connector" denilmektedir. Yani bizim "Python için MySQL connector" dosyalarını yüklememiz gereklidir.

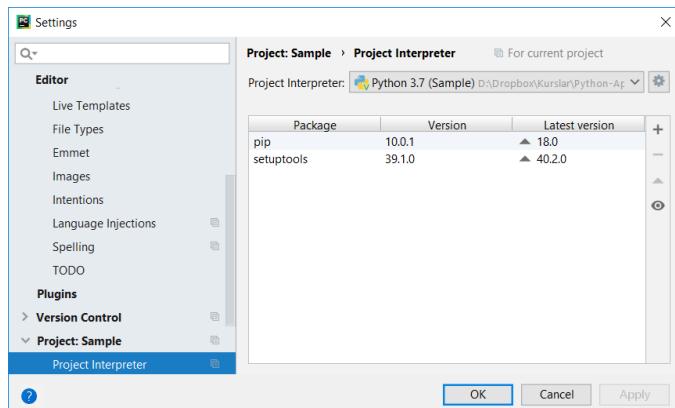
Python MySQL Connector komut satırında "pip" denilen Python yardımcı programıyla Internet'ten yüklenebilir. pip yazılımların yüklü olduğu çeşitli "repository'lere bakarak oradan yüklemeyi yapmaktadır. pip programı Python'ın içerisinde -m seçeneğiyle de çalıştırılabilir. MySQL Connector yüklemesi şöyle yapılabilir:

```
pip install mysql-connector-python
```

Aşağıdaki işlem de eşdeğerdir:

```
python -m pip install mysql-connector-python
```

pip kullanırken dikkat edilmesi gereken en önemli nokta şudur: Sistemde birden fazla Python sürümü yüklü olabilmektedir. Her Python sürümünün ayrı bir pip paket yöneticisi vardır. Biz hangi Python versiyonuna paket yüklemesini yapacağımız o versiyona ilişkin pip ya da python programını çalıştırılmamalıdır. Eğer biz PyCharm IDE'sini kullanırsak Settings/Project/Project Interpreter menüsünden PyCharm'ın o proje kullandığı yorumlayıcıyı görebiliriz:



Tabii yükleme işlemi PyCharm'da aslında doğrudan yukarıdaki menüdeki + simbolünden de yapılabilir. pip ve diğer araçların ayrıntılı kullanımı ayrı bir başlık halinde ele alınacaktır.

Python'da MySQL kullanımını aslında SQLite kullanımını ile aynıdır. Tabii bazı detaylar vardır. MySQL'de connect fonksiyonu ile VTYs'ye bağlanırken SQLite'ta olduğu gibi dosya ismi belirtilmez. MySQL client-server çalışan gerçek bir VTYs olduğu için "host name", "user-name", "password" bilgileri de bağlantı için gerekmektedir. Kullanım hakkında genel bir tutorial W3C sayfasında aşağıdaki adreste bulunmaktadır:

https://www.w3schools.com/python/python_mysql_getstarted.asp

connect işlemi sırasında connect fonksiyonuna host, user, passwd ve database parametreleri girilmelidir. Örneğin:

```
import mysql.connector as mysql
```

```
try:
```

```

conn = mysql.connect(host='localhost', user='root', passwd='csd1993', database='world')
print('Ok')
except:
    print('Bağlantı başarısız!..')

```

Bağlantıdan sonraki temel çalışma biçimini SQLite ile aynıdır. Yani yine bir cursor nesnesi elde edilir ve onun üzerinde execute işlemi uygulanır. Ancak detaylarda farklılık vardır. Örneğin:

```

import mysql.connector as mysql

try:
    conn = mysql.connect(host='localhost', user='root', passwd='csd1993', database='world')
    cur = conn.cursor()
    cur.execute("SELECT Name, CountryCode FROM city")
    for record in cur:
        print('{}, {}'.format(record[0], record[1]))
except:
    print('DB işlemi başarısız!..')
finally:
    conn.close()

```

brew install msodbcsql@13.1.9.2 mssql-tools@14.0.6.0

Python'da Anahtar-Değer Temelli DBM Veritabanının Kullanılması

DBM anahtar değer tutan bir veritabanıdır. Biz DBM veritabanını sözlüklerin diskte oluşturulmuş ve kalıcı yapılmış bir biçimde düşünürebiliriz. DBM veritabanı ilk kez 1979 yılında Ken Thompson tarafından geliştirilmiştir. Daha sonra bu veritabanının pek çok farklı versiyonu oluşturulmuştur. Örneğin ilk kez BSD'lerde kullanılan Berkeley DB veritabanı DBM benzeri bir veritabanıdır. Benzer biçimde GNU projesinde de bu veritabanı ayrıca gerçekleştirılmıştır. Ayrıca Oracle'ın mülkiyetinde olan ndbm veritabanı da tamamen benzer bir işlev sunmaktadır.

Python'daki DBM veritabanı aslında bir arayüz gibidir. Yani yukarıda sözü edilen bazı veritabanlarını (sisteme bağlı olursa) aynı arayüzle kullanabilmektedir.

DBM veritabanı arayüzü şöyle kullanılmaktadır:

1) Önce veritabanı dbm modülündeki open fonksiyonuyla açılır. open fonksiyonunun parametrik yapısı şöyledir:

```
dbm.open(file, flag='r', mode=0o666)
```

Fonksiyonun birinci parametresi veritabanı dosyasının ismini belirtir. Bazı sistemler burada belirtilen isme ilişkin iki dosya oluştururken bazıları tek dosya oluşturmaktadır. İkinci parametre veritabanını açış modunu belirtir. Bu mod aşağıdakilerden biri olabilir:

Açış Modu	Anlamı
'r'	Bu modda veritabanından yalnızca okuma yapabiliyoruz. Yeni bir anahtar-değer çifti ekleyemeyiz. Veritabanı yoksa exception olur.
'w'	Bu modda veritabanından hem okuma hem de ona yazma yapabiliyoruz. Veritabanı yoksa exception olur.
'c'	Bu modda veritabanından hem okuma hem de ona yazma yapabiliyoruz. Bu modda veritabanı yoksa yaratılır.
'n'	Veritabanı yoksa yaratır. Okuma yazma yapabiliyor. Veritabanı zaten varsa

	exception oluştur.
--	--------------------

dbm.open fonksiyonunun son parametresi UNIX/Linux sistemlerindeki dosyanın erişim haklarını belirtmektedir. Bu parametre default olarak geçilebilir. Fonksiyon başarı durumunda bize işlem yapmak için kullanacağımız bir DBM nesnesi verir. Örneğin:

```
>>> import dbm  
>>> d = dbm.open('test', 'c')  
>>> type(d)  
<class 'dbm.dumb._Database'>
```

2) Veritabanı yaratıldıktan ya da açıldıktan sonra artık anahtar-değer çiftleriyle işlem yapabiliriz. Örneğin:

```
import dbm  
  
d = dbm.open('testdbm', 'c')  
print(type(d))  
  
d['ali'] = '100'  
d['veli'] = '200'  
d['selami'] = '300'  
d['ayşe'] = '400'  
d['fatma'] = '500'  
  
print("d['ayşe'] = {}".format(d['ayşe']))  
print("d['fatma'] = {}".format(d['fatma']))
```

Göründüğü gibi veritabanına yeni bir anahtar-değer çifti eklemek için tek yapılacak şey sözlüklerde olduğu gibi köşeli parantez operatörüyle atama yapmaktadır. Değer yine köşeli parantez operatörüyle anahtar verilerek elde edilir.

3) dbm.open fonksiyonunun geri döndürdüğü dbm nesnesinin keys isimli örnek metodu bize veritabanındaki tüm anahtarları vermektedir. Örneğin:

```
import dbm  
  
d = dbm.open('testdbm', 'c')  
print(type(d))  
  
for key in d.keys():  
    print('{} => {}'.format(str(key), str(d[key]), encoding='utf-8'))
```

Ancak values biçiminde bir metod yoktur. keys metodunun dolaşılabilir bir nesne geri verdiğine dikkat ediniz. Örneğin:

```
import dbm  
  
d = dbm.open('testdbm', 'c')  
print(type(d))  
  
keys = list(d.keys())  
print(keys)
```

dbm veritabanı içsel olarak anahtar ve değeri her zaman bytes nesnesi biçiminde tutup bize vermektedir. Ancak arayüz anahtar ve değer olarak bytes nesnesinin yanı sıra bizden str de kabul edebilmektedir. Bu durumda bizden alınan string default encode metodunu ile UTF-8 formatına dönüştürüllerken bytes nesnesi oluşturulur. dbm köşeli parantez ile anahtarları verdigimizde değeri bize her zaman bytes nesnesi olarak verir. Örneğin:

```
>>> d['ayşe']  
b'400'
```

```
>>>
```

Anahtar ya da değer str ya da bytes dışında bir tür olamaz. Örneğin:

```
>>> d['ayşe'] = 500
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    d['ayşe'] = 500
  File "C:\Python37\lib\dbm\dumb.py", line 204, in __setitem__
    raise TypeError("values must be bytes or strings")
TypeError: values must be bytes or strings
```

Örneğin:

```
>>> d[100] = 'ayşe'
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    d[100] = 'ayşe'
  File "C:\Python37\lib\dbm\dumb.py", line 200, in __setitem__
    raise TypeError("keys must be bytes or strings")
TypeError: keys must be bytes or strings
```

Eğer biz veritabanında anahtar ve değer olarak yazı tutacaksak UTF-8 encoding'i ile bytes nesnesini yeniden yazıya dönüştürmek zorunda kalabiliriz. Örneğin:

```
>>> d['ağrı'] = 'dağ'
>>> d['ağrı']
b'da\xc4\x9f'
>>> d['ağrı'].decode(encoding='utf-8')
'dağ'
```

decode metodunun default UTF-8 encoding'ine göre dönüştürme yaptığını anımsayınız:

```
>>> d['ağrı'].decode()
'dağ'
DBM veritabanında yine del ve in operatörleri sözlüklerde olduğu gibi çalışmaktadır. Örneğin:
```

```
>>> 'veli' in d
True
```

Örneğin:

```
>>> del d['veli']
>>> 'veli' in d
False
```

len metodu da sözlükteki toplam eleman sayısını bize verir. Örneğin:

```
>>> len(d)
6
```

DBM veritabanında herhangi bir uygunsuz durumla karşılaşıldığında dbm.error isimli bir sınıfla raise işlemi yapılmıştır. Tabii biz exception'ları parametresiz except bloğu ile ya da taban sınıf olan Exception parametreli exception bloğu ile yakalayabiliriz. Örneğin:

```
import dbm
try:
```

```

d = dbm.open('testdbm', 'c')
v = d['xxx']
print(v)
except:
    print('anahtar bulunamadı!')

```

Ancak bu veritanında bazı hatalar dbm.error ile değil standart KeyError, TypeError gibi exception sınıflarıyla da ifade edilmektedir. Örneğin bir anahtar bulunamadığında KeyError, anahtara atanın değer str ya da bytes olmadığıda TypeError ile raise işlemi yapılmaktadır.

Yukarıda da belirtildiği gibi asında dbm bir arayüzdür. Bu arayüz farklı DBM gerçekleştirimlerini kullanabilmektedir. dbm arayüzü sırasıyla şu gerçekleştirimlerin sisteme yüklü olup olmadığına bakarak ilk bulduğu gerçekleştirmi kullanmaktadır:

```

dbm.dbmgnu
dbm.ndbm
dbm.dumb

```

Listenin sonunda dbm.dumb gerçekleştirimini görürsünüz. Bu gerçekleştirim eğer diğerleri yoksa devreye girmektedir. İsminin "dumb" olması bu gerçekleştirimin "biraz yavaş" olabileceğini ima etmektedir. Windows'taki Python kurulumunda diğer gerçekleştirimler olmadığı için "dumb" gerçekleştirimini görebilirsiniz. Tabii programcı isterse DBM arayüzü yerine alt paketteki belli bir gerçekleştirmi de -eğer varsa tabii- kullanabilir.

Her DBM gerçekleştiriminin dosya formatı farklı olduğu için veritabanı dbm.open fonksiyonuyla açılırken bu uyuma da bakılmaktadır. eğer söz konusu dosyanın formatına uygun bir dbm gerçekleştirmi yükleyse yukarıdaki listede arka sırada olsa bile o gerçekleştirim kullanılmaktadır. Bir veritabanı dosyasının hangi gerçekleştirimle oluşturulduğu ayrıca istenirse hiç open uygulanmadan dbm.whichdb fonksiyonu ile sorgulanabilir. Örneğin:

```

>>> dbm.whichdb('testdbm')
'dbm.dumb'

```

Nesnelerin pickle Modülü ile Seri Hale Getirilmesi

Nesnelerin seri hale getirilmesi (object serialization) pek çok framework ve kütüphanede bulunan bir modüldür. Bir nesne tüm elemanlarıyla bir dosyada ya da başka bir ortamda saklanabilir. Buna nesnenin "seri hale getirilmesi (serialization)" denilmektedir. Sonra seri hale getirilmiş bu bilgilerden yeniden orijinal nesne elde edilebilir. Buna da "seri hale getirilmiş nesnenin geri alınması (deserialization)" denilmektedir. İşte Python'da bu işlemler pickle modülündeki fonksiyonlar ve sınıflarla yapılmaktadır. Seri hale getirme ve geri alma işlemi kabaca şöyle yapılmaktadır:

1) Nesne pickle modülünün dump isimli fonksiyonu ile seri hale getirilebilir. Aslında dump içerisinde Pickler isimli sınıfı kullanmaktadır. Bu sınıf daha sonra ele alınacaktır. pickle.dump fonksiyonunun parametrik yapısı şöyledir:

```

pickle.dump(obj, file, protocol=None, *, fix_imports=True)

```

Fonksiyonun bitinci parametresi seri hale getirilecek nesneyi belirtir. ikinci parametre built-in open fonksiyonundan elde edilen dosya nesnesini belirtmektedir. Biz istersek seri hale getirme işlemi için önceden belirlenmiş bir protokol kullanabiliriz. Bu konu ileride ele alınacaktır. Default protokol "binary" olduğu için dosyanın da binary modda açılması gerekmektedir. Örneğin:

```

import pickle

l = [1, 2, 'Ali', ['Kırmızı', 'Mavi', 'Yeşil'], 4.5]

try:
    with open('serialize.dat', 'wb') as file:
        pickle.dump(l, file)

```

```
except Exception as err:  
    print(err)
```

2) Nesneyi geri almak için pickle.load fonksiyonu kullanılmaktadır. Bu fonksiyonun parametrik yapısı şöyledir:

```
pickle.load(file, *, fix_imports=True, encoding="ASCII", errors="strict")
```

Fonksiyonun birinci parametresi ilgili dosyaya ilişkin dosya nesnesini alır. Diğer parametreler default değerlerle geçilebilir. Fonksiyon bize ilgili nesneyi yaratıp onun adresini vermektedir. Orijinal nesne hangi türdense load bize o türden bir nesne verir. Şüphesiz load dosyadan (ya da ilgili kaynaktan) okumayı dosya göstericisinin gösterdiği yerden yapar. Yani nesneyi geri almadan önce dosya göstericisinin tam uygun konumda olması gereklidir. Örneğin:

```
import pickle  
  
try:  
    with open('serialize.dat', 'rb') as file:  
        l = pickle.load(file)  
        print(l)  
except Exception as err:  
    print(err)
```

Aslında pickle modülündeki global dump fonksiyonu kendi içerisinde Pickler isimli bir sınıf nesnesi yaratıp o sınıf sınıfın dump metodunu çağrırmaktadır. Yani biz dump işlemini şöyle de yapabildik:

```
import pickle  
  
l = [1, 2, 'Ali', ['Kırmızı', 'Mavi', 'Yeşil'], 4.5]  
  
try:  
    with open('serialize.dat', 'wb') as file:  
        pickler = pickle.Pickler(file)  
        pickler.dump(l)  
except Exception as err:  
    print(err)
```

Burada:

```
pickler = pickle.Pickler(file)  
pickler.dump(l)
```

işleminin eşdeğerinin:

```
pickle.dump(l, file)
```

birimde olduğuna dikkat ediniz. Benzer biçimde load işlemi de aslında arka planda pickle modülünün Unpickler isimli sınıfı ile yapılmaktadır. Yani biz load işlemini global load fonksiyonuyla değil aşağıdaki gibi de yapabildik:

```
import pickle  
  
try:  
    with open('serialize.dat', 'rb') as file:  
        unpickler = pickle.Unpickler(file)  
        l = unpickler.load()  
        print(l)  
except Exception as err:  
    print(err)
```

Burada:

```
unpickler = pickle.Unpickler(file)
l = unpickler.load()
```

işleminin eşdeğerinin,

```
l = pickle.load(file)
```

biçiminde olduğuna dikkat ediniz.

Aslında biz kendi sınıflarımızı da -belirli koşulların sağlanması durumunda- seri hale getirip geri alabiliriz. Örneğin:

```
import pickle

class Person:
    def __init__(self, name, no):
        self.name = name
        self.no = no

    def __str__(self):
        return '{}, {}'.format(self.name, self.no)

try:
    with open('serialize.dat', 'wb') as file:
        person = Person('Kaan Aslan', 123)
        pickle.dump(person, file)
except Exception as err:
    print(err)
```

Seri hale getirdiğimiz bu nesneyi aşağıdaki gibi geri alabiliriz:

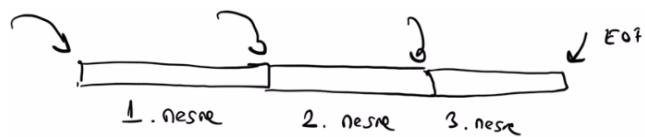
```
import pickle

class Person:
    def __init__(self, name, no):
        self.name = name
        self.no = no

    def __str__(self):
        return '{}, {}'.format(self.name, self.no)

try:
    with open('serialize.dat', 'rb') as file:
        person = pickle.load(file)
        print(person)
except Exception as err:
    print(err)
```

Nesneleri seri hale getirip dosya yazdıktan sonra bunları geri alırken dosya göstericisinin tam o noktada olması gereklidir. Örneğin biz üç nesneyi bu biçimde seri hale getirmış olalım:



Biz bu nesneleri aynı sırada alabiliriz. Ancak bunlardan herhangi birini almak istiyorsak dosya göstericisini o noktaya getirmemiz gereklidir. Örneğin:

```
import pickle
```

```
a = [1, 2, 3, 4, 5]
b = ['Ali', 'Veli', 'Selami']
```

```

c = {'name': 'Kaan Aslan', 'no': 123}

try:
    with open('serialize.dat', 'w+b') as file:
        pickle.dump(a, file)
        pickle.dump(b, file)
        pickle.dump(c, file)

        file.seek(0, 0)
        x = pickle.load(file)
        y = pickle.load(file)
        z = pickle.load(file)

    print(x, y, z)
except Exception as err:
    print(err)

```

Tabii biz yazarken dosya offset'ini kaydedersek tam o noktaya dosya göstericisini konumlandırarak geri alma işlemini yapabiliriz. Örneğin aşağıda ikinci nesnenin başlangıç offset'i kaydedilip sonra yalnızca bu ikinci nesne alınmıştır.

```

import pickle

a = [1, 2, 3, 4, 5]
b = ['Ali', 'Veli', 'Selami']
c = {'name': 'Kaan Aslan', 'no': 123}

try:
    with open('serialize.dat', 'w+b') as file:
        pickle.dump(a, file)
        pos = file.tell()
        pickle.dump(b, file)
        pickle.dump(c, file)

        file.seek(pos, 0)
        y = pickle.load(file)
        print(y)
except Exception as err:
    print(err)

```

pickle modülündeki dumps (sonundaki s'ye dikkat ediniz) fonksiyonu seri hale getirme işlemini bir dosyaya yapmaz. Seri hale getirilmiş byte'ları bize bytes nesnesi olarak verir. Örneğin:

```

import pickle

a = [1, 2, 3, 4, 5]
try:
    b = pickle.dumps(a)
    print(b)
except Exception as err:
    print(err)

```

Programın ekran çıktısı şöyle olacaktır:

```
b'\x80\x03]q\x00(K\x01K\x02K\x03K\x04K\x05e.'
```

Benzer biçimde loads fonksiyonu da bizden seri hale getirilmiş byte nesnesini alarak onu açar. Örneğin:

```

import pickle

a = [1, 2, 3, 4, 5]

```

```

try:
    b = pickle.dumps(a)
    print(b)
    x = pickle.loads(b)
    print(x)
except Exception as err:
    print(err)

```

dump ve dumps fonksiyonları ayrıca bir de protocol parametresi alabilmektedir. Bu protocol parametresi seri hale getirme işleminin kodlama biçimini belirtmektedir. Örneğin prtocol 0 kodlamadan ASCII karakterleriyle yapılmak üzere anlamına gelmektedir. Bu kodlama biçiminde seri hale getirilmiş olan nesne nispeten okunabilir durumdadır.

Örneğin:

```

import pickle

a = [1, 2, 3, 4, 5]
try:
    with open('serialize.dat', 'wb') as file:
        b = pickle.dump(a, file, protocol=0)
except Exception as err:
    print(err)

```

load sırasında protocol numarası belirtilmez. Çünkü zaten protokol numarası seri hale getirilen dosyaya yazılmaktadır. Örneğin:

```

import pickle

try:
    with open('serialize.dat', 'rb') as file:
        b = pickle.load(file)
        print(b)
    file.close()

except Exception as err:
    print(err)

```

Shelve Kullanımı

Shelve modülü avramı aslında dbm ile pickle modülünün birleştirilmiş bir hali gibidir. Anımsanacağı üzere dbm nesneleriyle biz anahtar ve değeri str ya da bytes olan nesneleri dosyaya kaydedip alabiliyoruz. Shelve tamamen dbm gibi çalışmaktadır. Shelve nesnelerinin dbm nesnelerinden en önemli farkı değer olarak string ya da bytes zorunluluğunun olmamasıdır. Yani shelve arka planda değeri önce pickle modülü ile seri hale getirip dbm ile yazar. Benzer biçimde geri alım sırasında da değeri önce bytes biiminde dbm kullanarak geri almaktır ve onu pickle modülü ile yeniden orijinal türde dönüştürmektedir. Örneğin:

```

import shelve

try:
    sh = shelve.open('testshelve')
    sh['Ali'] = 123
    sh['Veli'] = 456
    sh['Selami'] = 623

    for key in sh.keys():
        print('{0} => {1}'.format(key, sh[key]))

    sh.close()
except Exception as err:
    print(err)

```

Göründüğü gibi kullanım tamamen DBM'e benzemektedir. Ancak değer olarak biz artık herhangi bir türü (kendi sınıflarımız da dahil olmak üzere) kullanabiliriz. shelve sınıfı da "kaynak yönetim protokolünü (resource management protocol)" desteklediği için with deyi̇miyle kullanabilir. with çıkışında shelve dosyaları otomatik olarak kapatılacaktır. Örneğin:

```
import shelve

try:
    with shelve.open('testshelve.dat') as sh:
        sh['ali'] = 123
        sh['Veli'] = 456
        sh['Selami'] = 623

        for key in sh.keys():
            print('{} => {}'.format(key, sh[key]))
except Exception as err:
    print(err)
```

shelve modülündeki open fonksiyonunun ikinci parametresi dbm ile aynıdır. Burada default açış modu yine 'c' (yani dosya yoksa yarat varsa olanı aç anlamında) biçimindedir.

shelve nesnesi de len fonksiyonuna işlemeye sokulabilir. Yine del operatörü ile belli elemanı silebiliriz. in operatörü ile de belli bir anahtar bulunup bulunmadığını test edebiliriz. Örneğin:

```
import shelve

try:
    with shelve.open('testshelve', 'w') as sh:
        for key in sh.keys():
            print('{} => {}'.format(key, sh[key]))

        print('Toplam eleman sayısı: {}'.format(len(sh)))
        print('Anahtar Var' if 'Veli' in sh else 'Anahtar Yok')
        del sh['Veli']
        print('Anahtar Var' if 'Veli' in sh else 'Anahtar Yok')

except Exception as err:
    print(err)
```

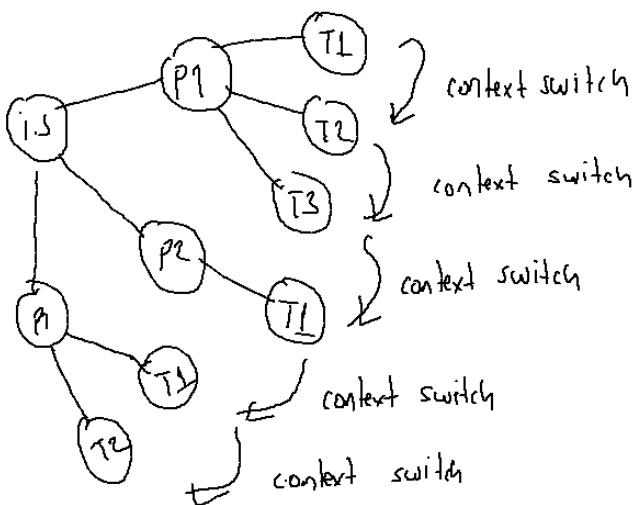
Python'da Thread Uygulamaları

Thread'ler bir programın işletim sistemi tarafından bağımsız çizelgelenen akışlarıdır. Bir program tek bir akışa sahip olmak zorunda değildir. Şimdiye kadar bizim programlarımıza tek bir akış vardı. Oysa programlarda birden fazla akış aynı zamanda işletilebilmektedir.

İşletim sistemleri terminolojisinde çalışmakta olan programlara "process" denilmektedir. Thread proseslerin akışlarını belirtir. Proses kavramı akışın dışında çalışmakta olan programın tüm özelliklerini betimlemektedir. Örneğin onun yetki derecesini, açmış olduğu dosyaları çalışma dizinini vs.

İşletim sistemleri proseslerin thread'lerini zaman paylaşımı olarak çalışmaktadır. Yani tipik olarak işletim sistemi bir prosesin bir thread'ini CPU ya da çekirdeğe atar. O thread belli süre çalıştırılır. Sonra işletim sistemi o thread'i durdurur. CPU ya da çekirdeğe diğer thread'i atar. Böyle böyle aslında programlar zaman paylaşımı biçimde çalıştırılmaktadır. CPU'lar ya da çekirdekler aynı anda birden fazla programı çalıştırılamazlar. "Biraz ondan, biraz bundan" biçiminde zaman paylaşımı bir çalışma uygulanmaktadır. Bir thread'in CPU ya da çekirdeğe atandığında parçalı çalışma süresine "quanta süresi" ya da "quantum" denilmektedir. İşletim sistemlerinin kullandığı tipik quanta süreleri 20 ms., 60 ms gibi değerlerdir. Quanta sürelerinin çok yüksek olması interaktiviteyi azaltmaktadır. Quanta süresinin çok kısa olması ise "birim zamanda yapılan iş miktarının (throughput)" azalmasına yol açar. Çünkü thread'ler

arası geçişin (buna "context switch" de denilmektedir) belli bir zaman maliyeti vardır. Böylece bir akışın belli bir noktadan belli bir noktaya gelmesi için gereken mutlak zaman sistemin yüküne göre değişebilmektedir.



Pekiyi sistemimizde birden fazla CPU ya da çekirdeğin (core) olması durumunda ne olacaktır? Aslında bu durumda prensipte değişen hiçbir şey yoktur. Yine zaman paylaşımı bir çalışma uygulanır. Bu durumda her CPU ya da çekirdeğin ayrı bir çizelge kuyruğu olacaktır. Tabii thread'ler birden fazla CPU ya da çekirdeğin kuyruğuna atanacaklarından toplamda çalışma süreleri hızlanacaktır. Ancak yine zaman paylaşımı bir çalışma söz konusudur. Bizim programımızın farklı thread'leri farklı CPU ya da çekirdeklerde atandığında onlar aynı anda çalışıyor olabilirler. Ancak bir thread aynı anda farklı CPU ya da çekirdeklerde çalışmamaktadır.

Thread sözcüğü etimolojik olarak "iplik" sözcüğünden gelmektedir. Akışlar ipliklere benzetilerek bu sözcük uydurulmuştur. Thread'ler bir prosesin bağımsız çizelgelenen akışlarını belirtir. Proses çalışmakta olan programın tamamını kavramsal olarak anlatmaktadır. Thread ise yalnızca bir akış belirtir. Dolayısıyla thread'ler proses kavramının içerisinde yer alır. Thread'lerin ilk ciddi denemeleri 80'li yıllarda yapılmıştır. Fakat 90'lı yıllarda işletim sistemlerine gerçek anlamda sokulmuştur. Örneğin DOS'ta thread yoktu. Windows 3.1 sistemleri de thread'li sistemler değildir. Microsoft'un ilk thread'li sistemi Windows NT (1993) ve sonra Windows 95 (1995)'tir. Linux'un ilk versiyonlarında thread'ler yoktu. 2.0'dan itibaren thread'li çalışma Linux sistemlerine sokulmuştur.

Cok thread'li işletim sistemlerinde proses çalışmaya bir thread'le başlar. Yani proses yaratıldığından bir thread de yaratılmış durumdadır. Buna prosesin ana thread'i (main thread) denir. Diğer thread'ler işletim sisteminin sistem fonksiyonlarıyla (yani Windows'ta API fonksiyonlarıyla, UNIX/Linux sistemlerinde POSIX fonksiyonlarıyla) yaratılırlar. Biz Python'da thread'ler için Python'ın standart kütüphanesindeki sınıfları ve metodları kullanacağız. Ancak bunlar aslında işletim sisteminin sistem fonksiyonlarıyla thread'leri oluşturulmaktadır.

Python'da thread işlemleri threading isimli standart bir modülle yapılmaktadır.

Thread'lere Neden Gereksinim Duyulmaktadır?

Thread'lere neden gereksinim duyulmaktadır? Bu gereksinim birkaç maddeyle özetlenebilir:

1) Thread'ler arka plan olayları izlemek için iyi bir araç oluşturmaktadır. Örneğin hem bir işi yaparken hem de ekranın sağ üst köşesine saat basmak isteyelim. Saati ne zaman basacağımız. Her işlemin arasında saat basmamız gereklidir. Peki bu durumda klavye ya da disk işlemleri yapıldığında ne olacak? Ya da hem bir işi yaparken hem de arka planda dışsal bir olay (örneğin seri portu, ya da bir termometreyi) izleyeceğimiz olalım. Eskiden bu tür işlemleri yapmak için tüm programın organizasyonunu değiştirmek gerekiyordu. Yani bu tür işlemler çok zor yapılabiliyordu. Halbuki thread'li sistemlerde bir thread yaratıp bu arka plan olayı bu thread' devredebiliriz. Böylece diğer thread'ler kendi işlemini yapabilir. Artık bu thread'ler bloke olsa bile arka plan olaylar izlenmeye devam edecektir.

2) Thread'ler bir programı hızlandırmak için kullanılabilir. Yani biz programımızda çok thread kullanırsak toplamda daha fazla CPU zamanı çekeriz.

3) Thread'ler blokeli IO işlemlerinde yoğun olarak kullanılmaktadır. Yani bir IO işlemi başlattığımızda (örneğin boru ya da soket gibi) belli bir süre bloke oluruz. Bu durumda gerekli olan başka şeyleri yapamayız. İşte IO işlemleri thread'lere yaptırılırsa blokeden yalnızca o thread etkilenir.

4) Thread'ler paralel programlama için mecburen kullanılmaktadır. Paralel programları bir işi parçalara ayırarak onu aynı anda birden fazla işlemci ya da çekirdeğe atayarak gerçekleştirmeye sürecine denilmektedir.

5) Thread'ler GUI programlama modelinde bazen mecburen kullanılmak zorundadır. Örneğin bir mesaj geldiğinde bir işi uzatırsak kuyrukta sıradaki mesajları işleyemeyiz. İşte uzun sürebilecek işlemler thread'lere havale edilebilir.

Thread'lerin Yaratılması

Python'da thread'ler iki biçimde yaratılabilir. Aslında iki yaratım biçimi birbirinin aynısıdır fakat kod organizasyonu bakımından farklılıkları vardır. threading modülündeki Thread sınıfı thread'i temsil etmektedir. Bu sınıf türünden bir nesne yaratılıp sınıfın start isimli örnek metodu çağrılırsa thread akışı çalışmaya başlar. Thread sınıfının başlangıç metodunun parametrik yapısı şöyledir:

```
Thread(group=None, target=None, name=None, args=(), kwargs={}, *, daemon=None)
```

Metodun birinci parametresi geleceğe yönelik saklı tutulmuştur. Bu parametre None olarak (zaten default değerinin None olduğunu dikkat ediniz) girilmelidir. target parametresi thread akışının başlatılacağı fonksiyonu ya da metodu belirtmektedir. Thread'lere birer isim verilebilir. name parametresi thread'e verilecek ismi belirtir. args ve kwargs parametreleri thread metoduna geçirilecek argümanları belirtmektedir. args parametresi bir demet biçiminde oluşturulmalıdır. kwargs parametresi ise bir sözlük biçiminde olmalıdır. daemon parametresi thread'in arka plan mı ön plan mı olduğunu belirtir. Örneğin:

```
import threading
import time

def main():
    thread = threading.Thread(target=threadProc)
    thread.start()
    for i in range(10):
        print('Main thread: {}'.format(i))
        time.sleep(1)

def threadProc():
    for i in range(10):
        print('Other thread: {}'.format(i))
        time.sleep(1)

main()
```

Burada hem ana thread hem de yarattığımız thread 0'dan 9'a kadar sayıları birer saniye aralıklarla ekrana yazdırmaktadır. Ekrana yazdırma sırasında thread'ler bir arada çalıştırırları için bir iç içe geçme olasıdır. Bunu önemsemeyiniz. time modülündeki sellep isimli fonksiyon parametresiyle belirtilen saniye kadar thread akışını bekletir. sleep fonksiyonun parametresi float türdendir. Dolayısıyla 0.1 saniye gibi değerler de girilebilmektedir.

Thread fonksiyonu çalışmaya başladığında ona parametre aktarılabilmektedir. Bunun için Thread sınıfının başlangıç metodundaki args parametresi kullanılır. Bu parametrenin bir demet olması gerekmektedir. Bu demetteki elemanlar thread fonksiyonuna argüman olarak geçirilmektedir. Örneğin:

```
import threading
import time
```

```

def main():
    thread = threading.Thread(target=threadProc, args=('Other Thread',))
    thread.start()
    for i in range(10):
        print('Main thread: {}'.format(i))
        time.sleep(1)

def threadProc(arg):
    for i in range(10):
        print('{}: {}'.format(arg, i))
        time.sleep(1)

main()

```

args argümanında tek bir eleman girilecekse parantezin içerisinde son ',' atomunu unutmayın.

Şüphesiz thread fonksiyonu bir sınıfın örnek metodudur. Örnek metodlarının ilgili sınıf türünden referansla ifade edildiğini anımsayınız. Örneğin:

```

import threading
import time

class Sample:
    def __init__(self):
        pass
    def threadProc(self, arg):
        for i in range(10):
            print('{}: {}'.format(arg, i))
            time.sleep(1)

def main():
    s = Sample()
    thread = threading.Thread(target=s.threadProc, args=('Other Thread',))
    thread.start()
    for i in range(10):
        print('Main thread: {}'.format(i))
        time.sleep(1)

main()

```

Yani thread'i temsil eden fonksiyonun global bir fonksiyon olması gerekmemektedir. Bir sınıfın örnek metodu, static metod ya da sınıf metodudur olabilir.

Birden fazla thread aynı thread fonksiyonundan çalışmaya başlayabilir. Örneğin biz bir döngü içerisinde birden fazla thread yaratabiliriz:

```

import threading
import time

threads = []

def main():
    for i in range(10):
        thread = threading.Thread(target=threadProc, args=(f'Thread No {i + 1}',))
        threads.append(thread)
        thread.start()

    for i in range(10):
        print('Main thread: {}'.format(i))
        time.sleep(1)

```

```

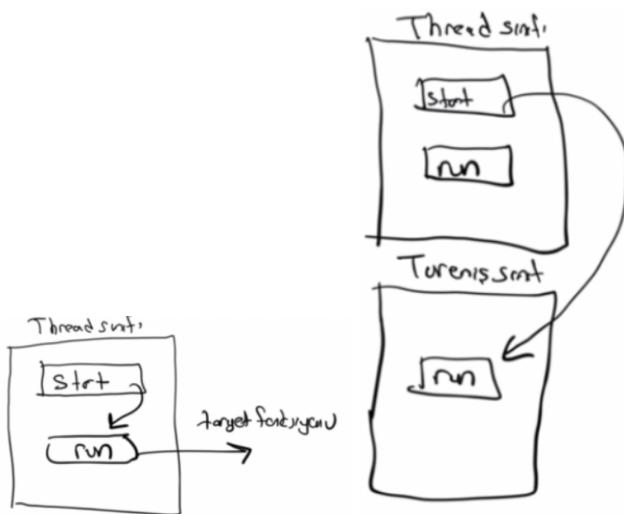
def threadProc(arg):
    for i in range(10):
        print('{0}: {1}'.format(arg, i))
        time.sleep(1)

main()

```

Bu örnekte 10 farklı thread akış `threadProc` isimli fonksiyondan başlayacak biçimde yaratılmıştır. Thread nesnelerinin global bir dizide saklandığına dikkat ediniz.

Thread'lerin yaratılmasının ikinci yolu `threading.Thread` sınıfından türetme yapmaktadır. Aslında `Threda` sınıfının `start` metodunu kendi içerisinde `Thread` sınıfının `run` metodunu çağrıır. Biz `Thread` sınıfından türetme yapıp bu `run` metodunu override edersek bizim `run` metodumuz çalıştırılacaktır. İşte bu `run` metodunu thread akışının başladığı metot olabilir. Normal olarak `Thread` sınıfının `run` metodunu (yani default `run` metodu) sınıfın başlangıç metodunda `target` parametresiyle belirtilen fonksiyonu çağrırmaktadır.



Örneğin:

```

import threading
import time

class MyThread(threading.Thread):
    def __init__(self, arg):
        super(MyThread, self).__init__()
        self.arg = arg

    def run(self):
        for i in range(10):
            print('{0}: {1}'.format(self.arg, i))
            time.sleep(1)

def main():
    mt = MyThread('Other thread')
    mt.start()
    for i in range(10):
        print('Main thread: {}'.format(i))
        time.sleep(1)

main()

```

Burada `MyThread` isimli sınıf `Thread` isimli sınıfından türetilmiştir. Dolayısıyla `Threda` sınıfıyla yapabileceğimiz her şeyi `MyThread` sınıfıyla da yapabilirim. Örneğin `MyThread` sınıfı türündne bir nesne yaratıp `start` metodunu çağrırlabilirim. `MyThread` nesnesiyle `start` metodunu çağrıldığında bu metot kendi içerisinde `run` metodunu çağrırmaktadır. `run`

metodunu da override ettiğimizden dolayı artık MyThread sınıfındaki run metodunu çalıştırılacaktır. Böylece bizim thread'imizin akışı MyThread sınıfının run metodundan başlamış gibi olacakatır.

Türetme yöntemiyle thread yaratmanın bir avantajı thread'in bir sınıfla temsil edilmesidir. Bu da nesne yönelimli teknik için daha uygun olabilmektedir. Ancak kursumuzdaki uygulamalarda diğer yöntemi daha fazla kullanılcagız.

Thread'lerin Sonlanması

Bir thread akışının sonlanması en normal yolu thread'in başlangıç fonksiyonunun doğal biçimde sonlanmasıdır. Örneklerimizde target parametresiyle belirttiğimiz fonksiyon ya da run metodunu bittiğinde thread'imiz de sonlanacaktır. Maalesef Python'ın threading modülünde bir thread'in kendi kendini sonlandıran (örneğin exit gibi) bir metot bulunmamıştır. Oysa hem işletim sistemlerinde hem de diğer pek çok ortamda (framework) bu işi yapacak fonksiyonlar bulunmaktadır. Ancak Python'da bir thread'de bir exception oluşursa bu exception yalnızca o thread'i sonlandırmaktadır. Thread'lerin herhangi bir noktada sonlandırılması bu yöntemle yapılabilir. Örneğin:

```
import threading
import time

def main():
    thread = threading.Thread(target=threadProc, args=('Other Thread',))
    thread.start()
    for i in range(10):
        print('Main thread: {}'.format(i))
        time.sleep(1)

def threadProc(arg):
    try:
        for i in range(10):
            print('{}: {}'.format(arg, i))
            foo(i)
            time.sleep(1)
    except:
        pass

def foo(i):
    if i == 5:
        raise Exception

main()
```

Python'da da tıpkı .NETe olduğu gibi thread'ler daemon (şeytan anlamına gelmekteir) thread ve normal thread olmak üzere ikiye ayrılmaktadır. Default durumda thread daemon değildir. Thread'i daemon yapmak için thread sınıfının daemon isimli elemanına True değerinin yerleştirilmesi gereklidir. Örneğin:

```
thread.daemon = True
```

Thread yaratılırken de Thread sınıfının başlangıç metodunda daemon parametresi True geçilebilir. Örneğin:

```
thread = threading.Thread(target=threadProc, args=('Other Thread',), daemon=True)
```

Daemon thread ile normal threda arasındaki tek fark şudur: Sistemdeki son daemon olmayan thread sonlandığında bütün daemon thread'ler otomatik olarak sonlandırılıp program sonlandırılmaktadır. Ana threaea normal bir thread'tir. Biz ana thread'i sonlandırırsak başka bir normal thread sonlanmaz. Son normal thread sonlandığında tüm daemon thread'ler sonlanmaktadır. Aşağıdaki örnekte ana thread içerisinde bir normal threda yaratılmıştır. Sonra ana thread sonlandırılmıştır. Fakat yaratılmış olan normal thread çalışmaya devam edecektir:

```
import threading
import time
```

```

def main():
    thread = threading.Thread(target=threadProc, args=( 'Other Thread',))
    thread.start()

    for i in range(10):
        print('Main thread: {}'.format(i))
        if i == 5:
            break
        time.sleep(1)

def threadProc(arg):
    for i in range(10):
        print('{}: {}'.format(arg, i))
        time.sleep(1)

main()

```

Şimdi ana yarattığımız thread'i daemon thread yapalım. Artık ana thread sonlandığında daemon thread'lerin hepsi sonlandırılacaktır:

```

import threading
import time

def main():
    thread = threading.Thread(target=threadProc, args=( 'Other Thread',))
    thread.daemon = True
    thread.start()

    for i in range(10):
        print('Main thread: {}'.format(i))
        if i == 5:
            break
        time.sleep(1)

def threadProc(arg):
    for i in range(10):
        print('{}: {}'.format(arg, i))
        time.sleep(1)

main()

```

Bu örnekte ana thread'in sayacı 5'e geldiğinde ana thread sonlandırılmaktadır. Artık yarattığımız thread çalışmaya devam etmeyecektir. Örneğin uygulamalarda yalnızca ana thread normal yapılıp diğerleri daemon yapılrsa ana thread sonlandığında diğer tüm thread'ler sonlandırılacaktır.

Her durumda sistemdeki son normal thread'in sonlandığında programın sonlanacağına dikkat ediniz.

Thread'in Sonlanması Beklenmesi

Bazen bir thread yaratılır ve diğer bir thread o thread sonlana kadar belli bir noktada beklemek isteyebilir. Çok thread'lu uygulamalarda bu tür durumlarla sık karşılaşılmaktadır. Bunun için thread sınıfının join isimli metodu kullanılır. Örneğin:

```

import threading
import time

def main():
    thread = threading.Thread(target=threadProc, args=( 'Other Thread',))
    thread.start()
    thread.join()
    print('ok')

```

```

def threadProc(arg):
    for i in range(10):
        print('{0}: {1}'.format(arg, i))
        time.sleep(1)

main()

```

İstenirse join metoduna bir zaman aşımı değeri verilebilir. Bu durumda join eğer thread hala sonlanmamışsa en fazla o kadar bekler.

Thread Sınıfının Diğer Önemli Elemanları

threading modülündeki global current_thread metodu bize kendi thread'imizin (yani bu metodu çağırın thread'in) thread nesnesini verir. Thread sınıfının name özeniteliği ise thread'e verdığımız ismi temsil etmektedir. Örneğin:

```

import threading
import time

def main():
    thread = threading.Thread(target=threadProc, name='Other Thread')
    thread.start()
    thread.join()
    print('ok')

def threadProc():
    for i in range(10):
        print('{0}: {1}'.format(threading.current_thread().name, i))
        time.sleep(1)

main()

```

Therad sınıfının is_alive isimli metodu thread sonlanmışsa False, sonlanmamışsa True değerine geri dönmektedir. Örneğin:

```
print('thread çalışıyor' if thread.is_alive() else 'thread sonlanmış')
```

Thread'lerin Stack'lerinin Birbirlerinden Ayrılmış Olması

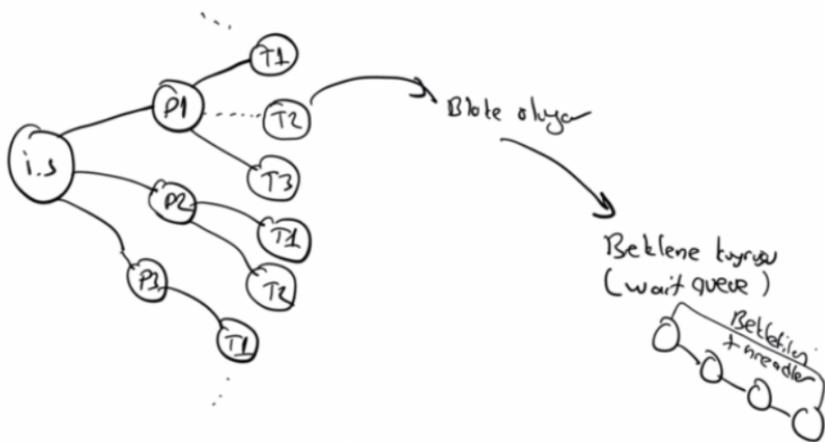
Metotların parametre değişkenleri ve yerel değişkenleri "stack" denilen bir bölümde yaratılmaktadır. Thread'lerin de her sistemde olduğu gibi Python'da da stack'leri birbirlerinden ayrılmıştır. Bu nedenle iki thread akışı aynı fonksiyon ya da metot üzerinde ilerlerken o fonksiyon ya da metodun yerel değişkenlerinin farklı kopyalarını kullanıyor durumdadırlar. Yani bir thread fonksiyondaki yerel değişkeni değiştirdiğinde diğer thread onu değişmiş olarak görmez. Thread'in yerel değişkenlerinin her thread için ayrı bir kopyası vardır. Halbuki global değişkenlerin toplamda tek bir kopyası vardır. Yani thread'lerden biri bir global değişkenin değerini değiştirirse diğerini değişmiş görür.

CPython Dağıtımındaki GIL (Global Interpreter Lock)

CPython yorumlayıcısı maalesef GIL (Global Interpreter Lock) denilen ana bir kilide sahiptir. Bu nednele bu dağıtımda aynı anda prosesin iki thread'i çalışamamaktadır. Örneğin sistemimizde 4 çekirdek olsun. Bizim CPython dağıtımında yarattığımız thread'ler farklı çekirdeklerde atanmış olabilirler. Ancak ne olursa olsun bunlar aynı anda çalışmazlar. Bu nedenle bu dağıtımdaki default thread kütüphanesi ile "paralel programlama" yapılamamaktadır. Tabii aslında bu durum Python dilinden değil gerçekleştirildiğinden kaynaklanmaktadır. Örneğin JPython ve IronPython gerçekleştirmelerinde bu kısıt yoktur. CPython'daki bu kısıt yorumlayıcının genel performansını artırmak için konulmuştur.

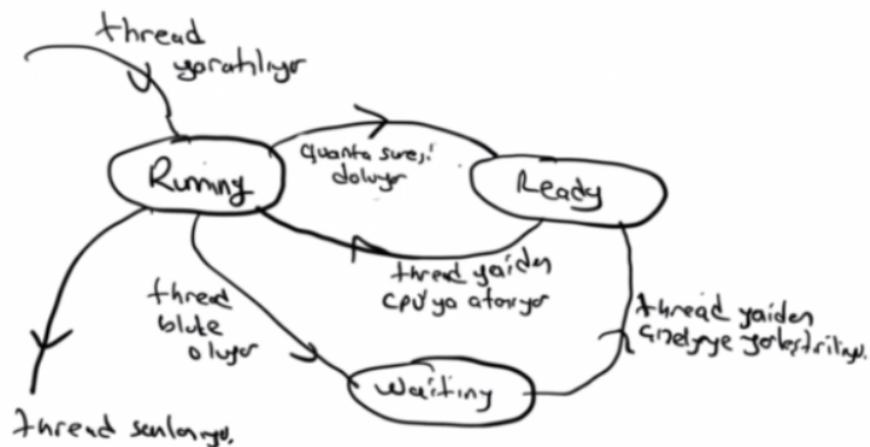
Bloke Kavramı

Bir thread çalışırken dışsal bir olayı başlattığında (örneğin disk işlemi, klavye okuması, soket okuması vs. gibi) işletim sistemi thread'i CPU zamanı harcanmasın diye geçici olarak çizelge dışına çıkartır ve olayı kendisi arka planda kesme (interrupt) tekniğiyle izler. Bu sırada sanki thread hiç çizelgede değilmiş gibi bekletilir. İşlem bittiğinde işletim sistemi yeniden thread'i çizelgeye yerleştirir. Sonuçta thread yine olay bitene kadar beklemiş olur fakat boşuna CPU zamanı harcanmamıştır. İşte bir thread'in bir işlem bitene kadar ya da gerçekleşene kadar çizelge dışına çıkarılarak bekletilmesine thread'in bloke olması (blocking) denilmektedir. Örneğin programlama dillerindeki sleep gibi fonksiyonlar da aslında mesgul bir döngüde sürekli bekleme yapmazlar. Bunlar da bloke yol açarak threadi bekletirler. Böylece bir sistemde yüzlerce thread olabilir fakat bunların çoğu belli bir olayı bekler durumdadır. Yani çok az thread aktif olarak belli bir anda CPU'yu kullanma eğilimindedir.



Aslında thread'in bir quanta süresinin tamamını harcaması çok nadirdir. Örneğin thread'in quanta süresi 20 ms. olsun. Pek çok thread daha birkaç milisaniye içerisinde bir IO olayına girer ve uzun süre bekler.

Bir thread'in yaşam döngüsü tipik olarak şöyledir:



Burada Running thread'in o anda CPU'ya atanmış olduğunu gösteriyor. Thread quanta süresini normal olarak doldurduğunda çizelgede bekletilir. Bu durum şekilde "Ready" ile temsil edilmiştir. Thread çalışırken bloke olursa çizelgeden çıkarılır. Bu durum da şekilde "Waiting" ile belirtilmiştir.

Bir proses bloke olduğunda işletim sistemi onu ismine "wait kuyruğu (wait queue)" denilen bir kuyrukta bekletir. Sonra olay gerçekleşince oradan onu alarak yeniden çizelge kuyruğuna koyar. Genellikle işletim sistemleri her olay için ayrı birer wait kuruğu oluşturmaktadır.

Pekiyi bir thread ne kadar zaman CPU harcayıp ne kadar zaman wait kuyruğunda bekler? Tabii bu thread'inden thread'ine değişir. Genel olarak çok CPU kullanan fakat az IO yapan thread'lere "CPU yoğun (CPU bound)" thread'ler, çok IO yapıp az CPU kullanan thread'lere de "IO yoğun (IO bound)" thread'ler denilmektedir. Genellikle thread'ler IO yoğun olma eğilimindedir. Örneğin matematiksel hesaplamalar yapan bir thread CPU yoğun, veritabanı işlemi yapan

bir thread IO yoğundur. O halde bir sistemde yüzlerce thread olsa da aslında bunların çoğu uykuda (yani wait kuyruğunda bekliyor) durumdadır.

Bir thread'in CPU kullanım oranından bahsedilebilir. Peki bu nasıl hesaplanmaktadır? Değişik hesaplama yöntemleri söz konusu olabilir. Bir thread'e verilen quanta süresinin o thread'in ortalama ne kadarını kullandığı iyi bir ölçüt olabilir. Örneğin thread'in toplam CPU'da harcığı zaman ile wait kuyruklarında harcadığı zamanın toplamı ile de bir oran belirlenebilir.

Thread Senkronizasyonu

Thread'ler konusunun en önemli bölümünü thread senkronizasyonu oluşturmaktadır. Thread'ler bir arada çalışırken birbirlerini beklemek zorunda kalabilirler.

Kritik Kod (CriticalSection) Kavramı

Başından sonuna kadar tek bir akış tarafından çalıştırılması gereken kod parçalarına kritik kod (critical section) denilmektedir. Pek çok durumda thread'ler ortak bir kaynak üzerinde bir arada çalışma yapıyor olabilirler. Bu ortak kaynak bir veri yapısı olabileceği gibi dış dünyadaki donanımsal bir aygıt da olabilir. İşte bir thread ortak bir kaynak üzerinde ilerlerken o sırada thread'ler arası geçiş olursa o kaynak kararsız bir durumda kalır. Diğer bir thread kaynağı kullanmak istedığında sorun çıkar. Örneğin iki thread'ığın aynı global değişkeni artırdığını düşünelim. Artırma yapan thread tam artırmanın ortasında kesilirse ve diğer thread artırma yapmaya çalışırsa bu işlem umulduğu gibi gerçekleşmeyebilir. Bu tür ortak kaynak kullanan thread'lerin işin tamamı bitene kadar birbirlerini beklemesi gerekmektedir. Örneğin:

```
import threading

x = 0

def main():
    thread1 = threading.Thread(target=threadProc1, args=( 'Thread-1', ))
    thread2 = threading.Thread(target=threadProc2, args=( 'Thread-1',))

    thread1.start()
    thread2.start()

    thread1.join()
    thread2.join()

def threadProc1(args):
    global x

    for i in range(10_000_000):
        x += 1

def threadProc2(args):
    global x

    for i in range(10_000_000):
        x += 1

main()

print(x)      # 20000000 çıktı mı?
```

Burada bizim işlemi tek bir `+=` operatörüyle yapmış olmamız bunun atomik olacağı anlamına gelmez. Makine komutları atomiktir. Yani bir makine komutu çalıştırılırken zaten thread'ler arası geçiş oluşamaz. Ancak iki makine komutu arasında thread'ler arası geçiş olabilir. İşte Python yorumlayıcısı `x += 1` ifadesini tek bir makine komutuyla yapmak zorunda değildir. Örneğin yorumlayıcı bu işlemi aşağıdaki gibi üç makine komutuyla yapabilirler:

```

MOV    reg, x
INC    reg
MOV    x, reg

```

Gerçekten de yukarıdaki program çalışırsa muhtemelen yitmi milyon değeri görülmeyecektir. İşte bu örnekteki artırım işlemi kritik bir kodu temsil eder. Bu artırım başından sonuna kadar tek bir thread akışı tarafından yapılmalıdır.

Kritik kodlar manuel olarak oluşturulamazlar. Örneğin aşağıdaki gibi bir kodla kritik kod oluşturamayız:

```

flag=False
while flag:
    pass
    flag=True
}
flag=False

```

Bu kodun iki sorunu vardır: Birincisi bekleyen thread meşgul bir döngüde (busy loop) bekleme yapar. İkincisi burada ok ile gösterilen noktada thread'ler arası geçiş oluşursa birden fazla thread kritik koda girebilir.

Pekiyi bu işlem güvenli bir biçimde nasıl yapılabilir? İşte bu işlem ismine "senkronizasyon nesneleri denilen" bir grup nesneyle yapılmaktadır. Python'da bu amaçla Lock nesneleri (mutex nesneleri) kullanılmaktadır.

Python'da Lock (Mutex) Kullanımı

Lock nesneleri şöyle kullanılmaktadır:

1) Lock nesnesi global düzeyde yaratılır. Ya da yerel düzeyde yaratılıp thread fonksiyonlarına parametre yoluyla aktarılır. Lock nesneleri threading modülündeki Lock isimli sınıfı temsil edilmektedir:

```

def main():
    lock = threading.Lock()
    thread1 = threading.Thread(target=threadProc1, args=(lock, ))
    thread2 = threading.Thread(target=threadProc2, args=(lock, ))

    thread1.start()
    thread2.start()

    thread1.join()
    thread2.join()

```

2) Kritik kod Lock sınıfının acquire ve release metodları arasına alınır. acquire kilidi elde etmek için (yani kilitlemek için) release ise serbest bırakmak için (yani açmak için) kullanılmaktadır.

```

lock.acquire()
}
lock.release()

```

Thread'lerden biri acquire metodu ile kilidi aldığı zaman artık diğer thread'ler kilidi alan thread release metodu ile kilidi bırakana kadar acquire metodunda blokede bekler. Tabii bekleme meşgul bir döngüde değil çizelge dışına çıkılarak yapılmaktadır. Böylece kritik kod başından sonuna kadar tek bir akış tarafından çalıştırılmış olur.

```
import threading

x = 0

def main():
    lock = threading.Lock()
    thread1 = threading.Thread(target=threadProc1, args=(lock, ))
    thread2 = threading.Thread(target=threadProc2, args=(lock, ))

    thread1.start()
    thread2.start()

    thread1.join()
    thread2.join()

def threadProc1(lock):
    global x

    for i in range(10_000_000):
        lock.acquire()
        x += 1
        lock.release()

def threadProc2(lock):
    global x

    for i in range(10_000_000):
        lock.acquire()
        x += 1
        lock.release()

main()

print(x)      # 20000000 çıkışacak
```

Ayrıca lock deyimi "kaynak yönetim protokolünü (resource management protocol)" desteklemektedir. Yani biz lock nesnelerini with ile kullanabiliriz. Bu durumda with deyimine girişte nesne otomatik olarak kilitlenip, çıkışlığında açılmaktadır. Yukarıdaki kodu biz daha sade biçimde şöyle de yazabilirdik:

```
def threadProc1(lock):
    global x

    for i in range(10_000_000):
        with lock:
            x += 1

def threadProc2(lock):
    global x

    for i in range(10_000_000):
        with lock:
            x += 1
```

Anımsanacağı gibi iki thread aynı anda ekrana birşeyler yazmak istediginde yazım biçimsel olarak bozulabiliyordu. İşte Lock nesneleri ile biz yazının bozulmamasını sağlayabiliyoruz. Örneğin:

```

import threading
import time

def main():
    lock = threading.Lock()
    thread1 = threading.Thread(target=threadProc1, args=( 'thread-1', lock, ))
    thread2 = threading.Thread(target=threadProc2, args=( 'thread-2', lock, ))

    thread1.start()
    thread2.start()

    thread1.join()
    thread2.join()

def threadProc1(arg, lock):
    for i in range(10):
        with lock:
            print(' {}: {}'.format(arg, i))
        time.sleep(1)

def threadProc2(arg, lock):
    for i in range(10):
        with lock:
            print(' {}: {}'.format(arg, i))
        time.sleep(1)

main()

```

Aşağıdaki örnekte kritik kod etkisini daha açık görebiliriz:

```

import threading
import time
import random

def main():
    lock = threading.Lock()
    thread1 = threading.Thread(target=threadProc1, args=( 'thread-1', lock, ))
    thread2 = threading.Thread(target=threadProc2, args=( 'thread-2', lock, ))

    thread1.start()
    thread2.start()

    thread1.join()
    thread2.join()

def proc(arg, lock):
    for i in range(10):
        with lock:
            print('-----')
            print(' {} Step-1: '.format(arg))
            time.sleep(random.random())
            print(' {} Step-2: '.format(arg))
            time.sleep(random.random())
            print(' {} Step-3: '.format(arg))
            time.sleep(random.random())
            print(' {} Step-4: '.format(arg))
            time.sleep(random.random())
            print(' {} Step-5: '.format(arg))
            time.sleep(random.random())

```

```

def threadProc1(arg, lock):
    proc(arg, lock)

def threadProc2(arg, lock):
    proc(arg, lock)

main()

```

Pekiyi birdne fazla thread acquire metodunda kilidin açılmasını beklerken kilit açıldığından hangi bekleyen thread kilidi alır? Genel olarak işletim sistemleri bu konuda herhangi bir garanti vermemektedir. (Yani kilide ilk gelen thread'in kilidi alacağı yönünde bir garanti yoktur.)

Lock sınıfının acquire isimli metodu biraz daha geniş kullanıma sahiptir. Metodun parametrik yapısı şöyledir:

```
acquire(blocking=True, timeout=-1)
```

blocking parametresi default olarak True biçimdedir. Eğer bu parametre False yapılrsa acquire kilidi alamadığı durumda blokeye yol açmaz. Bu durumda metot False değerine geri döner. Örneğin:

```

result = lock.acquire()
if result:
    
else:
    

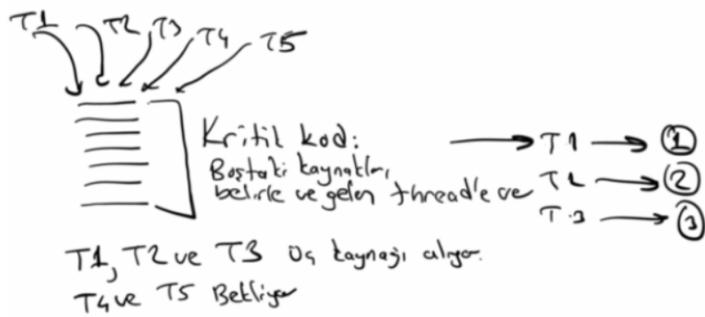
```

Eğer kilit başka bir thread tarafından alınmamışsa blokesiz modda acquire True ile geri dönmektedir. acquire metodunun timeout parametresi zaman aşımını belirtir. Thread en fazla burada belirlenen saniye kadar blokede kalmaktadır. Halbuki default durumda thread kilit açılana kadar blokede kalır. acquire metodu eğer zaman aşımından dolayı sonlanmışsa False değerine, kilit açık olduğundan dolayı sonlanmışsa True değerine geri döner.

Lock nesnelerini hangi thread kilitlemişse o thread açmak zorundadır. Yani başka bir thread eğer kilidi almamışsa relese metodu ile kilidi açamaz.

Semaphore Nesneleri

Semaphore'lar sayıçlı senkronizasyon nesneleridir. Bir kritik koda en fazla n tane akışın girmesini sağlamak için kullanılırlar. Semaphore'lar sayesinde üretici-tüketici problemi gibi algoritmik problemler çözülebilmektedir. Semaphore'lar özellikle n tane kaynağın bölüştürülmesi gerekiği durumlarda tercih edilmektedir. Örneğin elimizde üç tane kaynak olsun biz bu kaynakları n tane thread'in olduğu bir ortamda paylaşmak isteyelim. İlk gelen üç thread bu kaynakları elde edecektir. Ancak daha sonra gelenler kaynakları elden eden thread'lerin en az biri kaynağı bırakana kadar blokede bekleyecektir.

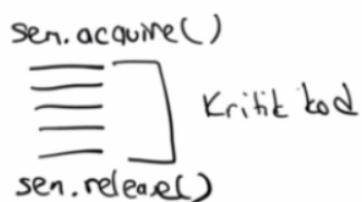


Semaphore kullanımı şöyledir:

1) Semaphore nesnesi bir sayaç sayısı verilerek yaratılır. Bu sayı paylaştırılacak kaynağı temsil etmektedir. Örneğin:

```
sem = threading.Semaphore(3)
```

2) Kritik kod şöyle oluşturulur:



Semaphore sınıfının acquire metodu eğer semaphore sayacı sıfırdan büyüğe geçişe izin verir. Ancak sayaç otomatik olarak bir eksiltilir. Eğer semaphore sayacı 0 ise acquire metodu blokeye yol açarak thread'i bekletir. Ta ki sayaç sıfırdan büyük olana kadar, release metodu semaphore sayacını bir artırmaktadır. Böylece artık kritik koda girmek için bekleyen bir thread giriş yapabilir. Ancak kritik koda girmek için birden fazl thread bekliyorsa hangi thread'in kritik koda gireceği hakkında bir garanti verilmemektedir.

Eğer bir semaphore yaratılırken semaphore sayacı 1 ise böyle semaphore'lara "binary semaphore'lar" denilmektedir. Binary semaphore'lar Lock (yani mutex) nesnelerine benzemektedir. Ancak semaphore nesnelerine başka bir thread tarafından release uygulanabilmektedir. Bu bakımdan binary semaphore'lar yine de Lock (mutex) nesnelerindne ayrılmaktadır. Semaphore'lar en fazla üretici-tüketicili tarzı problemlerin çözümü için kullanılmaktadır. Burada önce üretici-tüketicili problemini ele alacağız. Aslında Python'da zaten üretici-tüketicili problemi queue isimli sınıfla gerökleştirilmiştir. Yani programcının aşağıda anlatılacağı biçimde bu problemi semaphore'larla çözmesine gerek yoktur.

Üretici-Tüketicili Problemi (Producer-Consumer Problem)

Üretici-Tüketicili problemi programlamada en fazla karşılaşılan senkronizasyon problemlerinden biridir. Bu problemde thread'lerden biri bir döngü içerisinde bir değer elde eder. O değeri paylaştırılan bir alana yerleştirir. Diğer thread'de onu alarak kullanır. Değeri bulup paylaştırılan alana yerleştiriren thread'e "ürütici thread", bunu kullanan thread'e "tüketicili thread" denilmektedir. Burada sorun şudur: Üretici thread ile tüketici thread asenkron çalışmaktadır. Üretici thread daha tüketici thread eski değeri almadan yeni bir değeri paylaştırılan alana koyarak eski değeri ezmemelidir. Benzer biçimde tüketici thread de aynı değeri birden fazla kez almamalıdır. İşte burada özel bir senkronizasyonun uygulanması gereklidir. Aşağıda senkronizasyon uygulanmamış böyle bir sistem simülasyonunu görüyorsunuz:

```
import threading
import time
import random

shared = 0

def main():
    pass
```

```

thread_producer = threading.Thread(target=producer)
thread_consumer = threading.Thread(target=consumer)

thread_producer.start()
thread_consumer.start()

thread_producer.join()
thread_consumer.join()

def producer():
    global shared

    i = 0
    while True:
        time.sleep(random.random())
        shared = i
        i += 1
        if i == 100:
            break

def consumer():
    while True:
        val = shared
        time.sleep(random.random())
        print(val, end=' ', flush=True)
        if val == 99:
            break
    print()

main()

```

Çıkan değerler şuna benzerdir:

```

0 0 0 1 1 1 3 4 4 5 6 7 11 12 12 14 14 15 16 17 18 18 19 21 22 24 24 25 25 26 27 27 28 30 30 31
32 32 36 36 37 38 38 40 41 42 43 43...

```

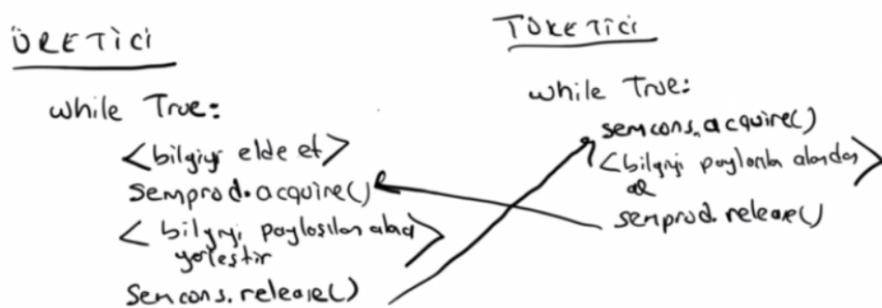
Üretici-Tüketici problemi tipik olarak semaphore nesneleriyle çözülmektedir. Çözüm için iki semaphore alınır. Üretici ve tüketici birbirlerini bu semaphore'larla beklerler. Yani üretici henüz tüketici değeri almadna yeni bir değer yerleştirmez. Tüketici de üretici yeni bir değer yerleştirmeden eski değeri yeniden almaz.

Üretici-Tüketici probleminin tipik çözümü şöyledir:

```

Semprod = threading.Semaphore(1)
Semcons = threading.Semaphore(0)

```



Burada iki semaphore oluşturulmuştur. Üretici semaphore'un başlangıç değeri 1, tüketici semaphore'un 0'dır. Üretici tüketicinin semaphore syacını, tüketici de üreticinin semaphore sayacını 1 artırır. Böylece birbirlerini kurtarırlar. Python kodu şöyle oluşturulabilir:

```

import threading
import time
import random

semprod = threading.Semaphore(1)
semcons = threading.Semaphore(0)
shared = 0

def main():
    thread_producer = threading.Thread(target=producer)
    thread_consumer = threading.Thread(target=consumer)

    thread_producer.start()
    thread_consumer.start()

    thread_producer.join()
    thread_consumer.join()

def producer():
    global shared

    i = 0
    while True:
        time.sleep(random.random())

        semprod.acquire()
        shared = i
        semcons.release()

        i += 1
        if i == 100:
            break

def consumer():
    while True:
        semcons.acquire()
        val = shared
        semprod.release()

        time.sleep(random.random())
        print(val, end=' ', flush=True)
        if val == 99:
            break
    print()

main()

```

Pekiyi üretici-tüketici probleminde neden tek bir thread hem üretim hem de tüketimi kendisi yapmıyor da bunun için iki farklı thread kullanılıyor? Bunun nedeni hız kazancı sağlamaktır. Bilgiyi tek bir thread'in alıp işlediğini düşünelim:

```

while True:
    <bilgiyi aldet>
    <bilgiyi işe>

```

Şimdi de bu işi iki farklı thread'in yaptığıni düşünelim:

*while True:
 <bilgiyi elde et>
 <bilgiyi diğerine ver>*

*while True:
 <bilgiyi al>
 <bilgiyi işle>*

Hangisi daha hızlı olur?

Üretici-Tüketici probleminde paylaşılan alan bir kuyruk sistemi olabilir. Bu durumda üretici yalnızca kuyruk dolu olunca tüketici de yalnızca kuyruk boş olunca diğerini bekler. Toplamda bekleme süresi çok daha azılır. Python'da bunu uygulamak için üreticinin semaphore sayacı kuruk uzunluğuna tüketicinin ise sıfır kurulur. Aşağıdaki uygulamada kuyruk sistemini bir listeleyle döngüsel olarak oluşturacağız.

```
import threading
import time
import random

shared = [0] * 10
semprod = threading.Semaphore(len(shared))
semcons = threading.Semaphore(0)

head = 0
tail = 0

def main():
    thread_producer = threading.Thread(target=producer)
    thread_consumer = threading.Thread(target=consumer)

    thread_producer.start()
    thread_consumer.start()

    thread_producer.join()
    thread_consumer.join()

def producer():
    global shared, head

    i = 0
    while True:
        time.sleep(random.random())

        semprod.acquire()
        shared[head] = i
        head += 1
        head %= len(shared)
        semcons.release()

        i += 1
        if i == 100:
            break

def consumer():
    global tail

    while True:
        semcons.acquire()
        val = shared[tail]
        tail += 1
        tail %= len(shared)
        semprod.release()

        time.sleep(random.random())
```

```

print(val, end=' ', flush=True)
if val == 99:
    break
print()

main()

```

Üretici tüketici probleminin birden fazla üretici ve birden fazla tüketici olduğu genel biçimleri de vardır. Temelde bunların gerçekleştirilmesi benzer biçimdedir. Kursumuzda bu gerçekleştirimler üzerinde durulmayacaktır.

Senkronize Kuyruk Nesneleri

Python'da çok üreticili ve çok tüketicili üretici-tüketici probleminin kuyruklu versiyonu hazır biçimde oluşturulmuştur. Yani programcının aslında yukarıdaki gibi bir kuyruk sistemi oluşturmasına bir gerek yoktur. queue modülündeki Queue isimli sınıf zaten kendi içerisinde semaphore kontrolü ile üretici-tüketici problemini gerçekleştirmektedir.

Bir Queue nesnesi kuyruk uzunluğu verilerek yaratılır. Sonra put metodu ile kuyruğa yerleştirme yapılır, get metodu ile kuyruktan bilgi alınır. Queue sınıfı FIFO kuyruk oluşturmaktadır. Bu sınıfın farklı birkaç değişik biçimde vardır. Bu durumda yukarıdaki üretici-tüketici problemini aşağıdaki gibi queue nesnesiyle gerçekleştirebiliriz:

```

import threading
import time
import random
import queue

q = queue.Queue(10)

def main():
    thread_producer = threading.Thread(target=producer)
    thread_consumer = threading.Thread(target=consumer)

    thread_producer.start()
    thread_consumer.start()

    thread_producer.join()
    thread_consumer.join()

def producer():
    i = 0
    while True:
        time.sleep(random.random())

        q.put(i)
        i += 1
        if i == 100:
            break

def consumer():
    while True:
        val = q.get()

        time.sleep(random.random())
        print(val, end=' ', flush=True)
        if val == 99:
            break
    print()

main()

```

Queue sınıfının emty isimli metodu kuyruk boşsa True değerine, doluya False değerine geri dönmektedir. İsterse programcı get ve put metodlarına zaman aşımı verebilmektedir. Bu metodların parametrik yapısı şöyledir:

```
Queue.put(item, block=True, timeout=None)
Queue.get(block=True, timeout=None)
```

Eğer get metodu metodlar zaman aşımından dolayı sonlanmışlarsa queue.Empty isimli exception'i put metodu ise queue.Full isimli exception'i oluşturmaktadır. Queue sınıfının ayrıca blokesiz biçimde işlem yapan get_nowait ve put_nowait metodları da vardır. (Aslında bu metodlar yerine get ve put metodlarında block=False da geçilebilir.) get_nowait metodunu kuyruk boşsa blokede beklemez bunun yerine queue.Empty isimli exception'i oluşturmaktadır. Benzer biçimde put_nowait metodunu da eğer kuyruk doluya beklemeyip doğrudan queue.Full exception oluşturmaktadır.

queue modülündeki PriorityQueue senkronize üretici-tüketici öncelik kuyruğu oluşturmaktadır. Bu kuyrukta elemanlar kuyruğa eklenirken onlara bir öncelik derecesi de verilmektedir. Böylece kuyruktan eleman alınırken alan thread önceliği en yüksek olan (sayısal olarak en düşük olan) elemanı önce alır. Eğer eşit öncelikli birden fazla eleman kuyruktan varsa FIFO sırası işletilmektedir.

Programın Komut Satırı Argümanları

Bir programı komut satırında çalıştırırken program isminden sonra yazılan yazılar "komut satırı argümanları (command line arguments) denilmektedir. Tabii Python'da biz programı Python yorumlayıcısı ile çalıştmaktayız. Ancak UNIX/Linux sistemlerinde script dosyalarının doğrudan çalıştırılabilğini anımsayınız. Örneğin:

```
D:\Dropbox\Kurslar\Python-App-August-2018\Src\CmdLine>python sample.py ali veli selami
```

Burada "ali veli selami" yazısı programın komut satırı argümanlarıdır. Python'da programın komut satırı argümanları sys modülündeki argv isimli değişkenle elde edilebilmektedir. Modüldeki argv değişkeni bir str listesidir. Listenin her elemanı bir komut satırı argümanını tutmaktadır. Listenin ilk elemanı her zaman çalıştırılan programın yol ifadesini belirtir. Örneğin:

```
# sample.py

import sys

for arg in sys.argv:
    print(arg)
```

Programı komut satırından çalıştıralım:

```
D:\Dropbox\Kurslar\Python-App-August-2018\Src\CmdLine>python sample.py ali veli selami
sample.py
ali
veli
selami
```

Bilindiği gibi eğer boşluk karakterleriyle ayrılmış olan yazıların tek bir argüman olarak aktarılması isteniyorsa çift tırnak içerisine alınmalıdır. Örneğin:

```
D:\Dropbox\Kurslar\Python-App-August-2018\Src\CmdLine>python sample.py "ali veli" selami
sample.py
"ali veli"
selami
```

Anımsanacağı gibi UNIX/Linux sistemlerinde komut satırında bu amaçla çift tırnak yerine tek tırnak da kullanılabilmektedir. (Tabii bu sistemlerde çift tırnakla tek tırnak arasında bazı küçük farklılıklar vardır.)

Komut satırı argümanlarının sys.argv dizisi içerisinde string biçiminde bulunduğuna dikkat ediniz. Aşağıda komut satırı argümanlarıyla girilen değerlerin toplamını ekrana yazdırın program görüyorsunuz:

```

# sample.py

import sys

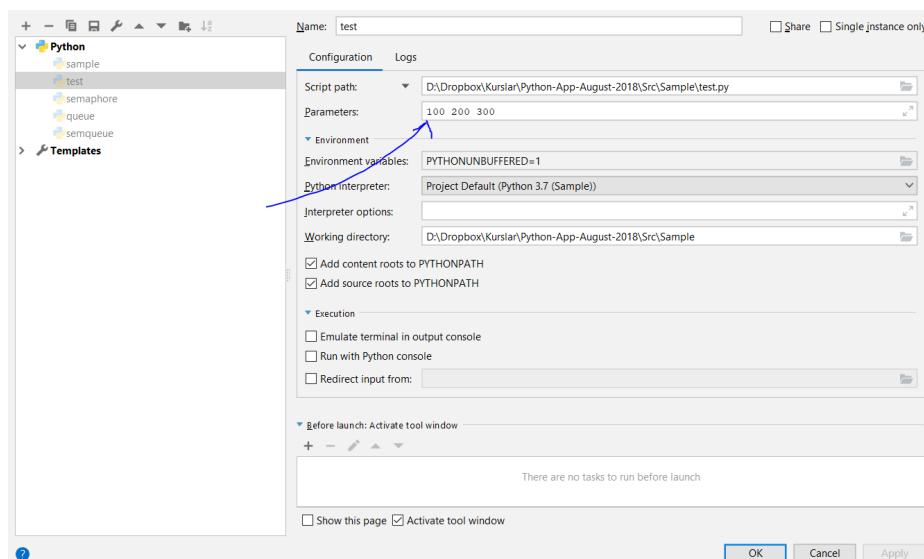
total = 0
try:
    for i in range(1, len(sys.argv)):
        total += float(sys.argv[i])
    print(total)
except:
    print('girilen argümanlardan en az biri sayı belirtmiyor!')

```

Programı komut satırında şöyle çalıştırabiliriz:

```
D:\Dropbox\Kurslar\Python-App-August-2018\Src\CmdLine>python sample.py 10 20 30
60.0
```

PyCharm IDE'sinde programı çalıştırduğumızda aslında programı IDE'nin kendisi python yorumlayıcısını kullanarak çalışmaktadır. İşte biz PyCharm IDE'sinde komut satırı argümanlarını Run/Edit Configuration/Parameters kısmında belirtebiliriz. Örneğin:



Python'da GUI Uygulamaları

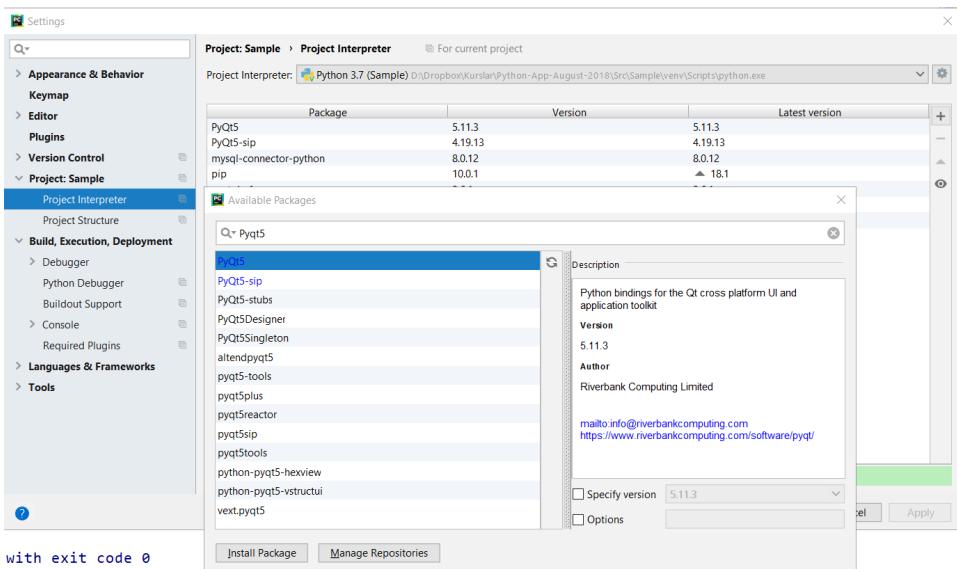
Python'da GUI uygulamaları için pek çok framework kullanılabilmektedir. Python'in kendisi "tkinter" denilen kütüphaneyi desteklemektedir. Yani Python'ın standart kütüphanesinde "tkinter" denilen GUI framework kullanıma hazır halde zaten bulunmaktadır. ("tkinter" "ti key intır" ya da "tikintir" biçiminde okunabilmektedir.) "Tkinter" aslında TCL denilen dilde ağırlıklı kullanılan "TK" isimli kütüphanenin Python'dan kullanılan biçimidir. Python'da "tkinter" kütüphanesinin en önemli alternatifi PyQt'dir. PyQt aslında bir Qt isimli C++ GUI framework'ünün Python'dan kullanılan biçimidir. Ancak Python'da kullanılabilmektedir. Yetenek olarak TK kütüphanesinden (yani "ttkinter"dan) oldukça iyidir. Tabii Python'da GUI işlemler için başka seçenekler de vardır. Örneğin wxWidget, Gtk+ diğer önemli seçeneklerdir.

Python'da PyQt Kullanımı

PyQt Python'ın standart kütüphanesi içerisinde olmadığı için bunun önce pip yoluyla kurulması gerekmektedir. Kurulum komut satırından şöyle yapılabilir:

```
D:\Dropbox\Kurslar\Python-App-August-2018\Src\CmdLine>python -m pip install pyqt5
```

Ya da PyCharm IDE'sinde File/Settings/Project/Project Interpreter/de kurulum yapılabilir:



PyQt projesi sourceforge.net sitesinde barındırılmaktadır. PyQt'nin genel dokümantasyonuna aşağıdaki adresen ulaşabilirsiniz:

<https://www.riverbankcomputing.com/software/pyqt/intro>

Sınıf dokümantasyonu ise aşağıdaki adreste bulunmaktadır:

<http://pyqt.sourceforge.net/Docs/PyQt4/classes.html>

Ancak buradaki sınıf dokümantasyonunda sınıfların metodları tek tek açıklanmamaktadır. Bu nedenle ayrıntılı açıklama için Qt'nin aşağıdaki orijinal dokümantasyonuna başvurabilirsiniz:

<http://doc.qt.io/qt-5/classes.html>

GUI Ortamlarında Mesaj Tabanlı Çalışma Modeli

Mesaj tabanlı programlama modelinde klavye ve fare gibi aygıtlarda oluşan girdileri programcı kendisi almaya çalışmaz. Fare gibi, klavye gibi girdi aygıtlarını işletim sisteminin (ya da GUI alt sistemin) kendisi izler. Oluşan girdi olayı hangi pencereye ilişkinse işletim sistemi ya da GUI alt sistem, bu girdi olayını “mesaj” adı altında bir yapıya dönüştürerek o pencerenin ilişkin olduğu (yani o pencereyi yaratan) programın “mesaj kuyruğu (message queue)” denilen bir kuyuk sistemine yerleştirir. Mesaj kuyruğu içerisinde mesajların bulunduğu FIFO prensibiyle çalışan bir kuyruk veri yapısıdır. Sistemin daha iyi anlaşılması için süreci maddeler halinde özetlemek istiyoruz:

1. Her programın (ya da thread'in) “mesaj kuyruğu” denilen bir kuyruk veri yapısı vardır. Mesaj kuyruğu mesajlardan oluşmaktadır.
2. İşletim sistemi ya da GUI alt sistem gerçekleşen girdi olaylarını “mesaj (message)” adı altında bir yapı formatına dönüştürmekte ve bunu pencerenin ilişkin olduğu programın (ya da thread'in) mesaj kuyruğuna eklemektedir.
3. Mesajlar ilgili olayı betimleyen ve ona ilişkin bazı bilgileri barındıran yapı (structure) nesneleridir. Örneğin Windows'ta mesajlar MSG isimli bir yapıyla temsil edilmişleridir. Bu yapının elemanlarında mesajın ne mesajı olduğu (yani neden gönderildiği) ve olaya ilişkin bazı bilgiler bulunur.

Göründüğü gibi GUI programlama modelinde girdileri programcı elde etmeye çalışmamaktadır. Girdileri bizzat işletim sisteminin kendisi ya da GUI alt sistemi elde edip programcıya mesaj adı altında iletmektedir.

GUI programlama modelinde işletim sisteminin (ya da GUI alt sistemin) oluşan mesajı ilgili programın (ya da thread'in) mesaj kuyruğuna eklemenin dışında başka bir sorumluluğu yoktur. Mesajların kuyruktan alınarak işlenmesi

ilgili programın sorumluluğundadır. Böylece GUI programcısının mesaj kuyruğuna bakarak sıradaki mesajı alması ve ne olmuşsa ona uygun işlemleri yapması gereklidir. Bu modelde programcı kodunu şöyle düzenler: Bir döngü içerisinde sıradaki mesajı kuyruktan al, onun neden gönderildiğini belirle, uygun işlemleri yap, kuyrukta mesaj yoksa da blokede bekle". İşte GUI programlarındaki mesaj kuyruğundan mesajı alıp işleyen döngüye mesaj döngüsü (message loop) denilmektedir.

Bir GUI programının işleyişini tipik akışı aşağıdaki gibi bir kodla temsil edebiliriz:

```

int main()
{
    <ana pencereyi başlat>
    for(;;)
        <sıradaki mesajı al>
        <mesajı işle>
        <x tuşa basılmışsa
          döngüde git>
    }
    return 0;
}

```

Bu temsili koddan da görüldüğü gibi tipik bir GUI programında programcı bir döngü içerisinde mesaj kuyruğundan sıradaki mesajı alır ve onu işler. Mesajın işlenmesi ise "ne olmuş ve ben buna karşı ne yapmalıyım?" biçiminde oluşturulmuş olan kodlarla yapılmaktadır.

Peki bir GUI programı nasıl sonlanmaktadır? İşte pencerenin sağındaki (bazı sistemlerde solundaki) X simgesine kullanıcı tıkladığında işletim sistemi ya da GUI alt sistem bunu da bir mesaj olarak o pencerenin ilişkin olduğu prosesin (ya da thread'in) mesaj kuyruğuna bırakır. Programcı da kuyruktan bu mesajı alarak mesaj döngüsünden çıkar ve program sonlanır.

GUI ortamımız ister .NET, ister Java, ister MFC olsun, isterse PyQt olsun, işletim sisteminin ya da GUI alt sistemin çalışması hep burada ele alındığı gibidir. Yani örneğin biz .NET'te ya da Java'da işlemlerin sanki başka biçimlerde yapıldığını sanabiliriz. Aslında işlemler bu ortamlar tarafından aşağı seviyede yine burada anlatıldığı gibi yapılmaktadır. Bu ortamlar (frameworks) ya da kütüphaneler çeşitli yükleri üzerinden alarak bize daha rahat bir çalışma modeli sunarlar. Ayrıca şunu da belirtmek istiyoruz: GUI programlama modeli özellikle nesne yönelimli programlama modeline çok uygun düşmektedir. Bu nedenle bu konuda kullanılan kütüphanelerin büyük bölümünü sınıflar biçiminde nesne yönelimli diller için oluşturulmuş durumdadır.

Şimdi GUI programlama modelindeki mesaj kavramını biraz daha açalımlım. Yukarıda da belirttiğimiz gibi bu modelde programcıyı ilgilendiren çeşitli oylara "mesaj" denilmektedir. Örneğin klavyeden bir tuşa basılması, pencere üzerinde fare ile tıklanması, pencere içerisinde farenin hareket ettirilmesi gibi oylar hep birer mesaj oluşturmaktadır. İşletim sistemleri ya da GUI alt sistemler mesajları birbirinden ayırmak için onlara birer numara karşılık getirirler. Örneğin Windows'ta mesaj numaraları WM_XXX biçiminde sembolik sabitlerle kodlanmıştır. Programcılar da konuşurken ya da kod yazarken mesaj numaralarını değil, bu sembolik sabitleri kullanırlar. (Örneğin WM_LBUTTONDOWN, WM_MOUSEMOVE, WM_KEYDOWN gibi) Mesajların numaraları yalnızca gerçekleşen olayın türünü belirtmektedir. Oysa bazı oylarda gerçekleşen olaya ilişkin bazı bilgiler de söz konusudur. İşte bir mesaja ilişkin o mesaja özgü bazı parametrik bilgiler de işletim sistemi ya da GUI alt sistem tarafından mesajın bir parçası olarak mesajın içerişine kodlanmaktadır. Örneğin Windows'ta biz klavyeden bir tuşa bastığımızda Windows WM_KEYDOWN isimli mesajı programın mesaj kuyruğuna bırakır. Bu mesajı kuyruktan alan programcı mesaj numarasına bakarak klavyenin bir tuşuna basılmış olduğunu anlar. Fakat hangi tuşa basılmıştır? İşte Windows basılan tuşun bilgisini de ayrıca bu mesajın içerişine kodlamaktadır. Örneğin WM_LBUTTONDOWN mesajını Windows farenin sol tuşuna tıklandığında kuyruğa bırakır. Ancak ayrıca basım koordinatını da mesaja ekler. Yani bir mesaj oluştuğunda

yalnızca o mesajın hangi tür bir olay yüzündenoluştuğu bilgisini değil aynı zamanda o olayla ilgili bazı bilgileri de kuyruktaki mesajın içerisindeñden alabilmektedir.

GUI programlama modelinde bir mesaj oluþtuðunda o mesajın bir an evvel işlenmesi ve akışın çok bekletilmemesi gerekir. Aksi takdirde programcı kuyruktaki diğer mesajları işleyemez bu da “program donmuş etkisi” yaratmaktadır. Eğer bir mesaj alındığında uzun süren bir işlem yapılmak isteniyorsa bir thread oluşturulup o işi o thread'e devretmek ve böylece mesaj döngüsünün işlemesini sağlamak gerekir.

GUI programlama modellerinde genel olarak mesaj kavramı pencere kavramıyla ilişkilendirilmiştir. Yani bir pencere yaratılmadıktan sonra bir mesajın oluşma durumu da yoktur. Bu nedenle mesaj döngüsüne girmeden önce programcının en az bir pencere (tipik olarak programın ana penceresi) yaratmış olması gerekir.

Windows gibi bazı sistemlerde thread'lerle ilişkilendirilmiştir. Bu sistemlerde prosesin tek bir mesaj kuyruğu yoktur. Her thread'in ayrı bir mesaj kuyruğu vardır. Bu durumda işletim sistemi ya da GUI alt sistem bir pencereye ilişkin bir işlem gerçekleştiðinde o pencerenin hangi prosesin hangi thread'i tarafından yaratılmış olduğunu belirler ve mesajı o thread'in mesaj kuyruğuna bırakır. Böylece biz bir thread oluşturup o thread'te de bir pencere yaratmışsa artık bizim de o thread'te o pencerenin mesajlarını işlemek için mesaj döngüsü oluşturmamız gereklidir. Tabii eğer thread'imizde biz hiçbir pencere oluşturmamışsa böyle bir mesaj döngüsünü oluşturmamıza da gerek yoktur. (Örneğin Microsoft eğer bir thread bir pencere yaratmışsa böyle thread'lere “GUI thread’ler” yaratmamışsa “worker thread’ler” demektedir).

İskelet PyQt Programı

Ekrana boş bir pencere çıkartan basit PyQt programı şöyle yazılabılır:

```
#generic.py

import sys
from PyQt5.QtWidgets import *

class MainWindow(QWidget):
    def __init__(self):
        super().__init__()

app = QApplication(sys.argv)
mainWindow = MainWindow()
mainWindow.show()
app.exec()
```

İskelet programda PyQt5.QtWidgets modülünün içerisindeki tüm isimlerin dışarıya aktarıldığına dikkat ediniz. Bu sayede biz PyQt sınıflarını hiç niteliklendirme yapmadan doğrudan kullanabilmekteyiz. PyQt5 bir paket (package) biçiminde organize edilmiştir. PyQt5 paketinin içerisinde çeşitli modüller vardır. En önemli modüllerden biri QtWidgets isimli modüldür. PyQt kütüphanesinde bütün sınıflar Q hatfiyle başlatılarak isimlendirilmiştir. Ancak programcı kendi sınıflarını Q harfi olmadan isimlendirmelidir. QAyrıca PyQt'de sınıfların metodları "deve notasyonu (camel casting)" isimlendirilmiştir.

Bir PyQt programında uygulanmanın tamamı QApplication isimli bir sınıfta temsil edilmektedir. QApplication sınıfı bizden komut satırı argümanlarını parametre olarak almaktadır. QApplication nesnesi yaratıldıkten sonra sıra programın ana penceresinin yaratılmasına gelmiştir. Ana pencere QWidget sınıfı ile oluşturulabilir. Ancak biz QWidget nesnesine eklemeler yapacaðımız için ana pencerenin bu QWidget sınıfından türetilmiş olan bir sınıfı oluşturulması daha uygundur. Yani aslında iskelet program aşağıdaki gibi de oluşturulabilirdi:

```
#generic.py

import sys
from PyQt5.QtWidgets import *
```

```

app = QApplication(sys.argv)
mainWindow = QWidget()
mainWindow.show()
app.exec()

```

Ancak bu durumda biz QWidget üzerinde eklemeler yapamazdık. İşte tipik olarak PyQt uygulamalarında ana pencere doğrudan QWidget sınıfı ile değil QWidget sınıfından türetilmiş bir sınıf ile oluşturulmaktadır. İskelet programda MainWindow isimli sınıf QWidget sınıfından türetilmiştir. MainWindow sınıfının `__init__` metodunda taban sınıf olan QWidget sınıfının `__init__` metodunun çağrıldığına dikkat ediniz. Tabii aslında biz MainWindow sınıfının `__init__` metodunu boş bırakmayacağız. Eğer boş bırakacak olsaydık doğrudan sınıfı pass anahtar sözcüğüyle de kapatabilidik.

İskelet programda ana pencere nesnesi yaratıldıktan sonra QWidget sınıfından geelen show isimli metot ile bu pencere görünür yapılmıştır. Bir pencerenin yaratılmasıyla görünür yapılması farklı adımlarla gerçekleştirilmektedir.

İskelet programda QApplication sınıfının exec isimli metodu mesaj döngüsü (message loop) oluşturmak için kullanılmaktadır. Ana pencere kapatıldığında bu fonksiyon sonlanmış olur. Program yaşamamını QApplication sınıfının exec metodunda geçirir.

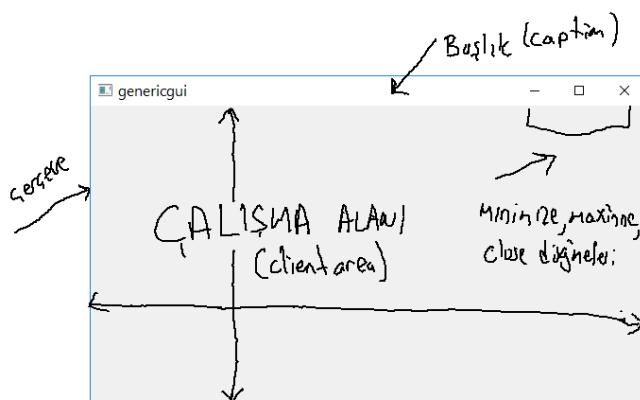
PyQt'de Program Organizasyonu

PyQt framework'ünde mesaj döngüsü QApplication sınıfının exec metoduyla oluşturulmaktadır. Bu metot bir döngü içerisinde mesaj kuyruğundaki sıradaki mesajı alır ve programcının belirlediği fonksiyonları ya da metodları çağırır. Programcı da GUI programını "şu düğmeye tıklandıysa şu metod çağırılsın" biçiminde organize eder. Bu çağrıma işlemini tamamen PyQt framework'ü arka planda organize etmektedir.

GUI Sistemlerinde Pencere Terminolojisi

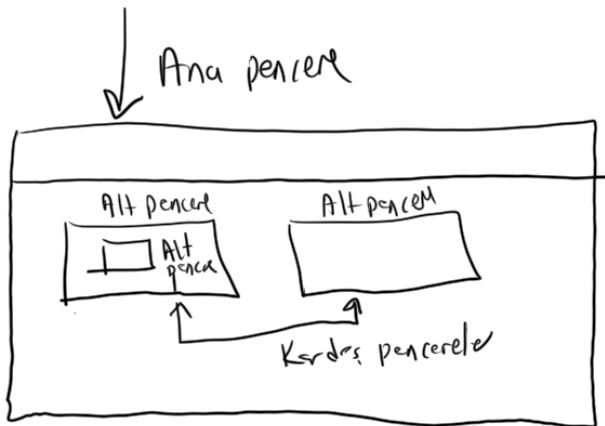
GUI tabanlı sistemlerde ekranda bağımsız olarak kontrol edilebilen dikdörtgensel bölgelere pencere (window) denilmektedir. Konumlarına ve işlevselliklerine göre pencereler birkaç gruba ayrılmaktadır. Doğrudan masaüstüne açılan pencerelere "ana pencereler (top level windows)" denilmektedir. Pek çok GUI alt sisteminde görünür durumda olan ana pencereler bir çubukta gösterilmektedir (örneğin Windows'taki task bar). Her ne kadar zorunlu değilse de ana pencerelerin genellikle bir çerçevesi (frame), bir başlık kısmı (caption) ve başlık kısmının üzerinde bazı simgeleri bulunur.

Pencerenin sınır çizgilerinin ve başlığının altında kalan alana "çalışma alanı (client area)" denilmektedir. Çalışma alanı bizim aktif çizim yapabileceğimiz, başka alt pencereleri yerleştirebileceğimiz (yani bizim için ayrılmış) olan alandır. Tabii bir pencerenin toplam genişliği ve yüksekliği çalışma alanının genişliği ve yüksekliği ile aynı olmak zorunda değildir. Eğer pencerenin başlığı ve sınır çizgileri yoksa bu durumda pencerenin kendi genişliği ve yüksekliği çalışma alanının genişliği ve yüksekliği ile aynı olur.



Bir pencerenin içerisinde görüntülenen, onun dışına çıkamayan pencerelere "alt pencereler (child windows)" denilmektedir. Alt pencerelerde genellikle başlık kısmının ve sınır çizgilerinin bulunması istenmez. (Ancak tabii alt pencereler de istenirse başlık kısmına ve sınır sizgilerine sahip olabilirler.) Alt pencerelerin alt pencereleri de söz

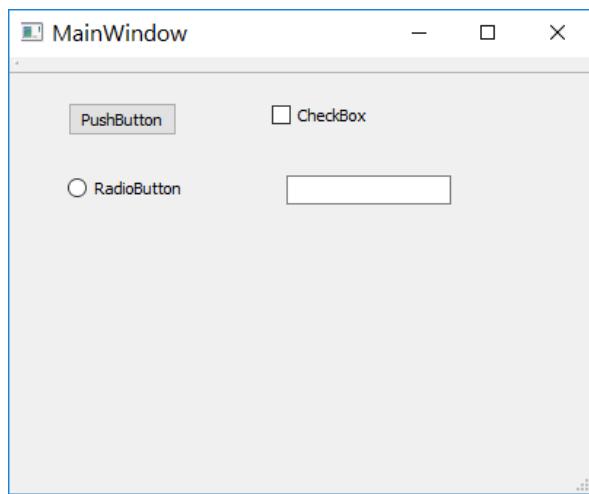
konusu olabilir. Her alt pencerenin bir üst penceresi (parent window) vardır. Aynı üst pencereye sahip olan pencerelere kardeş pencereler (sibling windows) denilmektedir.



Ayrıca bir de ismine “sahiplenilmiş (owned)” pencere denilen bir pencere türü daha vardır. Tipik olarak diyalog pencereleri bu türdendir. Sahiplenilmiş pencereler hem bir çeşit alt pencere gibi hem de ana pencere gibi davranışlarılar. Bunlar her zaman üst pencerelerinin yukarısında görüntülenir. Bunların üst pencereleri minimize edildiğinde bunlar da görünmez olurlar.

GUI Elemanları ve Alt Pencereler

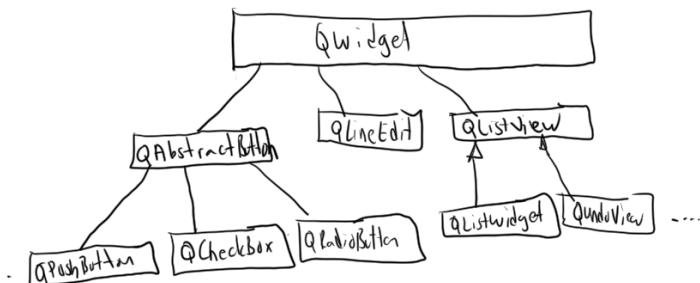
Programların GUI arayüzlerinde kullanılan düğmeler, edit alanları, listeleme kutuları, menüler vs. hep aslında birer alt pencerelerdir. Örneğin:



Aslında yukarıdaki ana pencerede gördüğünüz düğme gibi, seçenek kutusu gibi, radyo düğmesi gibi görsel öğeler içi boş pencereler üzerinde çizim işlemleri yapılarak oluşturulmuşlardır. Biz de sıfırdan içi boş bir pencereden hareketle kendi görsel öğelerimizi oluşturabiliriz. Ancak Qt gibi ortamlarda çeşitli görsel elemanlar alt pencere biçiminde zaten önceden oluşturulmuş durumdadır. Bu görsel öğeler birer sınıfla temsil edilmiştir. Böylece programcı hangi görsel öğeyi kullanacaksa o sınıf türünden bir nesne yaratır, sonra o nesnenin üye fonksiyonlarını çağırarak nesnenin tam olarak istediği biçimde gözükmescini sağlar. GUI ortamlarında kullanıcı arayüzüleri hep böyle oluşturulmaktadır. Ancak bir noktaya dikkat çekmek istiyoruz: Her ne kadar Qt’de -diğer ortamlarda olduğu gibi- pek çok görsel öğe alt pencere biçiminde hazır bulundurulmuş olsa da bunlar programının isteklerini tam olarak karşılamıyor olabilir. Bu durumda programcı arzu ettiği görsel öğeleri sıfırdan içi boş alt pencerelerden hareketle oluşturmak ya da başkaları tarafından oluşturulmuş olanları satın alıp kullanmak isteyebilir. Neyse ki gereksinimlerin büyük çoğunluğu mevcut alt pencere sınıflarıyla karşılanabilmektedir.

PyQt'de Pencere Sınıfları

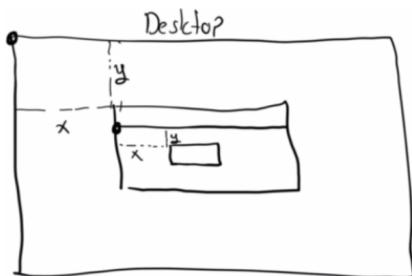
PyQt'de görsel arayüzü oluşturan alt pencere sınıflarının bütün ortak özelliklerini QWidget isimli bir sınıfı toplanmıştır. Tüm pencere sınıfları QWidget sınıfından türetilmiş durumdadır. Yani QWidget sınıfı tüm pencerelerin ortak özelliklerini barındıran en temel sınıfıdır. İskelet PyQt programında da ana pencerenin QWidget sınıfıyla yaratıldığına dikkat ediniz. Diğer tüm görsel öğeler çeşitli biçimlerde çeşitli sınıflardan türetilmiş durumdadır. Örneğin:



Yukarıdaki şekilde QPushButton, QCheckBox ve QRadioButton sınıflarının ortak elemanlarının QAbstractButton sınıfında toplandığına dikkat ediniz. PyQt'de bu biçimde oluşturulmuş geniş bir türetme şeması vardır.

PyQt'de Pencere Koordinatları

GUI sistemlerde ekranada görüntülenecek en küçük birime pixel (picture element) denilmektedir. Her türlü görüntü (yazılıar, resimler, şekiller vs.) aslında pixellerin bir araya gelmesiyle oluşturulmaktadır. Her pixel 16 milyon renkten bir tanesiyle renklendirilebilmektedir. Ekran bir pixel matrisi olarak düşünülebilir. Örneğin 1200X800 çözünürlük demekle aslında yatayda 1200, düşeyde 800 pixel'in olduğu bir matris anlaşılmaktadır. İşte PyQt'de bir ögenin yerini belirlemek için pixel koordinat sistemi kullanılmaktadır. Alt pencereler için orijin noktası o alt pencerenin üst penceresinin çalışma alanının sol-üst köşesidir. Üst pencereler için ise orijin noktası masaüstünen sol üst köşesidir.



PyQt'de Alt Pencerelerin Oluşturulması

Alt pencereler ilgili alt pencere sınıfı türünden nesne yaratılarak oluşturulabilirler. Alt pencereler oluşturulurken bunların başlangıç metodlarında (`__init__` metodlarında) yaratılacak olan alt pencerenin üst penceresi belirtilir. Genel olarak pek çok alt pencere sınıfının başlangıç metodlarının birinci parametresi başlık yazısı, ikinci parametresi üst pencere nesnesidir. Örneğin:

```

import sys
from PyQt5.QtWidgets import *

class MainWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.pushButtonOk = QPushButton('Ok', self)

def main():
    app = QApplication(sys.argv)
    mainWindow = MainWindow()
    mainWindow.show()
    app.exec()
  
```

```
main()
```

Burada QPushButton isimli alt pencere yaratılmış ve onun referansı sınıfın pushButtonOk isimli örnek özniteligidde saklanmıştır.

```
self.pushButtonOk = QPushButton('Ok', self)
```

Başlangıç metodunun birinci parametresinin düğme üzerinde ikacak yazısı, ikinci parametresinin ise üst pencere nesnesini belirttiğine dikkat ediniz. Buradaki self QMainWindow nesnesini temsil emektedir.

Pencerelerin Konumlandırılması

Bir pencere (QWidget) yaratıldığında başlangıç konumu (0, 0) biçimindedir. Programcının yarattığı pencelerleri konumlandırması gerekebilir. Aslında konumlandırma birtakım "layout" nesneleriyle otomatik de yapılmaktadır. Bu konu ileride ele alınacaktır. Pencerelerin konumlandırılması taban sınıf olan QWidget sınıfının metotları ile yapılmaktadır.

QWidget sınıfının pos isimli metodu pencerenin sol-üst köşesinin koordinatını bize QPoint türünden bir nesne olarak verir. QPoint bir noktayı temsil eden genel bir sınıfır. Aslında doğrudan QWidget sınıfının x ve y isimli metotları bize ayrı ayrı pencerennin konumunu QPoint olarka değil int türden vermektedir.

QWidget sınıfının move isimli metodu pencereyi konumlandırmak için kullanılır. move bizden pencerenin sol-üst köşesinin konumunu x ve y olarak almaktadır. Örneğin:

```
self.pushButtonOk = QPushButton('Ok', self)
self.pushButtonOk.move(100, 100)
```

QWidget sınıfının rect isimli örnek metodu bize pencerenin konumunu QRect isimli bir sınıf türünden vermektedir. Ancak bu metodun verdiği konum yine kendi çalışma alanı orijinlidir. Dolayısıyla bu metottan elde edeceğimiz x ve y değerleri her zaman (0, 0) olur. QRect dikdörtgensel bir bölgeyi temsil eden genel bir sınıfır.

size metodu bize pencerenin genişlik ve yüksekliğini QSize isimli bir sınıf türünden vermektedir. Pencerenin genişlik ve yüksekliğini resize metodu ile değiştirebiliriz. Örneğin:

```
class MainWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.pushButtonOk = QPushButton('Ok', self)
        self.pushButtonOk.move(100, 100);
        self.pushButtonOk.resize(200, 200)
```

QWidget sınıfının geometry isimli metodu bize oencerenin konumunu QRect olarak verir. Ancak bunun pos metodundan farkı orijin olarak üst pencereyi almasıdır. QWidget sınıfının setGeometry metoduyla da biz tek hamlede hem konumlandırma hem de boyutlandırma yapabiliriz. Bu metot bizden sırasıyla x, y, width ve height değerlerini istemektedir. Örneğin:

```
self.pushButtonOk = QPushButton('Ok', self)
self.pushButtonOk.setGeometry(100, 100, 200, 200)
```

Pencerenin genişliğini QWidget sınıfının width örnek metodu ile, yüksekliğini de height örnek metodu ile doğrudan alabiliriz.

Tabii ana pencere de QWidget sınıfın türetildiğine göre biz yukarıdaki metotları ana pencere için de kullanabiliriz. Ancak ana pencerede geometry, size, width ve height gibi metotlar pencere başlığını ve sınır çizgilerini dahil etmezler. Biz ana pencere için pencere başlığı ve sınır çizgileri dahil olmak üzere konum almak istiyorsak frameGeometry metodunu kullanmalıyız. Örneğin:

```

class MainWindow(QWidget):
    def __init__(self):
        super().__init__()
        print(self.geometry())
        print(self.frameGeometry())

```

Burada ana pencerenin konumu hem geometry metodu ile hem de frameGeometry metodu ile elde edilmiştir. Her iki metot da ana pencereler için x = 0, ve y = 0 değeri verirler. Ancak genişlik ve yükseklik geometry metodunda sınır çizgileri ve başlık kısım dahil olmayacağı biçimde (yani yalnızca çalışma alanı dahil olacak biçimde) frameGeomtry metodunda bunlar da dahil olacak biçimde verilir.

Ana pencereler için en büyük ve en küçük genişlik ve yükseklik değerleri QWidget sınıfının setMaximumWidth, setMinimumWidth, setMaximumHeight, setMinimumHeight metodlarıyla ayarlanabilmektedir. Bu metodlardaki değerler yine çalışma alanı ile ilgilidir. Yani pencerenin başlık kısımı ve sınır çizgileri bu değerlerde dikkate alınmamaktadır.

Konumlandırmaya ilgili akla takılan bazı sorular ve onların yanıtları aşağıda verilmiştir:

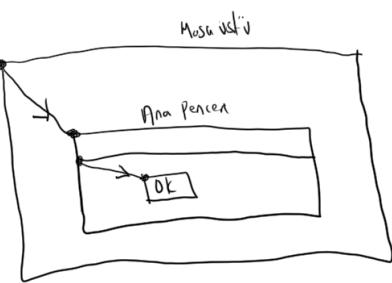
Soru: size fonksiyonu ile frameSize arasındaki farklılık nedir?

Cevap: size bize ilgili pencerenin çalışma alanının genişlik ve yüksekliğini, frameSize ise sınır çizgileri ve başlık kısımı dahil olmak üzere tüm pencerenin genişlik ve yüksekliğini vermektedir. size ile width ve height fonksiyonları her zaman aynı değeri verir. Fakat genellikle alt pencerelerin balık kısımları ve sınır çizgileri olmadığı için alt pencereler söz konusu olduğunda size ile frameSize aynı olacaktır.



Soru: pos, x ve y ve mov fonksiyonlarındaki sol üst köşe koordinatları neresidir ve orijini nereye göredir?

Cevap: Bu fonksiyonlardaki sol üst köşe her zaman pencerenin tamamının sol üst köşesidir. Buradaki sol üst köşe koordinatları alt pencere için üst pencerenin çalışma alanının sol üst kölesi orjinli, ana pencereler için masa üstünün sol üst kölesi orjinlidir.

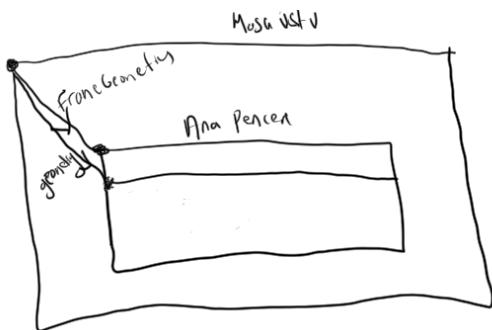


Soru: geometry ile rect fonksiyonları arasında ne fark vardır?

Cevap: Her iki fonksiyon da çalışma alanının genişlik ve yüksekliğini verir. Ancak rect sol üst köşeyi kendisi orjinli vermektedir. Halbuki geometry üst pencere orjinli olarak (ana pencere söz konusuya masa üstü masa üstü orjinli olarak) verir.

Soru: geometry ile frameGeometry fonksiyonları arasındaki fark nedir?

Cevap: geometry bize çalışma alanın genişlik ve yüksekliğini verir. Oysa frameGeometry bize tüm pencerenin (sınıf çizgileri ve başlık kısımları da dahil olmak üzere) genişlik ve yüksekliğini vermektedir. Ayrıca Ayrıca geometry bize sol üst köşe koordinatı olarak çalışma alanın koordinatını verir. Halbuki frameGeometry tüm pencerenin sol üst köşesinin koordinatını vermektedir. Yani frameGeometry ile verilen x ve y değerleri pos fonksiyonu ile verilenle aynıdır.



Soru: Mademki rect bize her zaman sol üst köşe koordinatı sıfır olan bir QRect veriyor, bunun uygulamada bir anlamı olabilir mi? Yani rect ile elde edilen bilgi ile size ile elde edilen bilgi aynı olmuyor mu?

Cevap: Evet teorik olarak böyle ancak bazen çalışma alanı konumunun QRect olarak ve çalışma alanı orijinli alınması gerekebilmiştir.

Pencere Başlık Yazısının Alınması ve Değiştirilmesi

Tüm pencerelerin bir yazısı vardır. Örneğin QPushButton penceresinin yazısı düğme üzerindeki yazıdır. QTextBox penceresinin yazısı edit alanının içerisindeki yazıdır. Ana pencerenin yazısı pencere başlığındaki (caption) yazıdır. PyQt'de ana pencerenin üzerinde başlık yazısı QWidget sınıfının windowTitle isimli metodu ile alınabilir, setWindowTitle metodu ile set edilebilir. Örneğin:

```
class MainWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle('Generic Window')
```

Herhangi bir alt pencerenin yazısı ise QWidget sınıfından gelen text metoduna alınıp setText metoduyla set edilebilir.

Mesaj Pencerelerinin Kullanımı

GUI uygulamalarında mesajları konsol ekranına yazmak iyi bir teknik değildir. Bu ancak debug amaçlı uygulanabilecek bir tekniktir. Programcı mesajları doğrudan bir diyalog penceresi içerisinde kullanıcıya gösterir. İşte Qt'de bu diyalog penceresine MessageBox denilmektedir. Messagebox PyQt'de QMessageBox sınıfıyla temsil edilmiştir. MessageBox oluşturmak için programcı önce message box penceresini yaratır. Sonra pencerenin başlık yazısını setWindowTitle metodu ile, içerisinde görüntülenecek yazıyı da setText metodu ile oluşturur. Sonra da sınıfın exec metodunu çağırır. Örneğin:

```
msg = QMessageBox(self)
msg.setWindowTitle('Error')
msg.setText('File not found!')
msg.exec()
```

Bu biçimde çıkartılacak mesaj penceresinde default olarak Ok düğmesi vardır. Ancak biz QMErrorMessage sınıfının setStandardButtons metodu ile birkaç düğme arasından bir takım seçebiliriz.

```
msg = QMessageBox(self)
msg.setWindowTitle('Error')
msg.setText('File not found!')
msg.setStandardButtons(QMessageBox.Abort | QMessageBox.Retry | QMessageBox.Ignore)
msg.exec()
```

Tabii bu tür durumlarda bizim mesaj penceresini hangi düğmeyle kapattığımız bilmemiz gerekir. exec metodunun geri dönüş değeri hangi düğmeyle mesaj penceresinin kapatıldığını belirtmektedir. Örneğin:

```
msg = QMessageBox(self)
msg.setWindowTitle('Error')
msg.setText('File not found!')
msg.setStandardButtons(QMessageBox.Abort|QMessageBox.Retry|QMessageBox.Ignore)
result = msg.exec()
if result == QMessageBox.Abort:
    print('abort')
elif result == QMessageBox.Retry:
    print('retry')
else:
    print('ignore')
```

Mesaj pencerelerine birkجاç simge görüntüsünden biri yerleştirilebilir. Bunun QMessageBox sınıfının setIcon metodu kullanılmaktadır. Örneğin:

```
msg = QMessageBox(self)
msg.setWindowTitle('Error')
msg.setText('File not found!')
msg.setStandardButtons(QMessageBox.Abort|QMessageBox.Retry|QMessageBox.Ignore)
msg.setIcon(QMessageBox.Information)
msg.exec()
```

Kullanılacak simge listesi şunlardır:

```
QMessageBox.NoIcon
QMessageBox.Question
QMessageBox.Information
QMessageBox.Warning
QMessageBox.Critical
```

Aslında biz doğrudan pencereyi QMessageBox sınıfının question, warning, information, critical isimli static metodlarıyla da oluşturabiliriz. Bu metodlar sırasıyla bizden üst pencere referansını, balık yazısını ve pencere içerisindeki yazıyı almaktadır. Örneğin:

```
QMessageBox.information(self, 'Example', 'This is a test')
```

Bu metodlarda istenirse dördüncü parametreyle tuş takımı da belirtilebilir:

```
QMessageBox.information(self, 'Example', 'This is a test',
QMessageBox.Abort|QMessageBox.Retry|QMessageBox.Ignore)
```

PyQt'de Sinyal Slot Kavramı

Qt'de gerçekleşen oylara sinyal (signal), bu olaylar sonucunda çağrılacak metodlara da slot (slot) denilmektedir. Biz sinyallere slot bağlayarak belli bir olay gerçekleştiğinde bir metodumuzun çağrılması sağlanabilmektedir. Sinyallere yalnızca slot'lar değil, sinyaller de bağlanabilmektedir. Örneğin biz A sinyaline B sinyalini de bağlayabiliyoruz. Bu durumda A sinyali oluştuğunda (Qt terminolojisinde "emit" edildiğinde deniyor) bu durum B sinyalinin oluşmasına yol açmaktadır. Bir sinyale tek bir slot ya da sinyal bağlanmak zorunda değildir. İstenildiği kadar çok sinyal ve slot bağlanabilir.

Sınıfların sinyallerinin neler olduğu Qt dokümanlarda belirtilmektedir. Türemiş sınıf taban sınıfın sinyallerini de içermektedir. Örneğin QPushButton sınıfının tüm sinyal kümesi QPushButton, QAbstractButton ve QWidget sınıflarının sinyallerinden oluşmaktadır.

Örneğin biz bir düğmeye basıldığında bir kodumuzu çalıştırmak isteyelim. İşte yapmamız gereken şey QPushButton sınıfının clicked isimli sinyaline bir slot bağlamaktadır. Sinyal sınıflarının connect isimli metotlarıyla bu bağlantı yapılabilmektedir. Örneğin:

```
import sys
from PyQt5.QtWidgets import *

class MainWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle('Generic Window')

        self.pushButtonOk = QPushButton('Ok', self)
        self.pushButtonOk.setGeometry(100, 100, 100, 100)
        self.pushButtonOk.clicked.connect(self.buttonClickedHandler)

    def buttonClickedHandler(self):
        QMessageBox.information(self, 'Message', 'Ok')

def main():
    app = QApplication(sys.argv)
    mainWindow = MainWindow()
    mainWindow.show()
    app.exec()

main()
```

Bu örnekte sınıfın örnek clicked elemanı bir sinyaldir. Bu sinyale buttonClickHandler isimli slot bağlanmıştır. Böylece düğmeye tıkladığımızda bu metot çağrılr. Sinyaller pyqtBoundSignal isimli bir sınıf türündendir. conenct metodu da aslında bu sınıfın bir örnek metodudur. Tabii aslında slot global bir fonksiyon da olabilmektedir. Örneğin:

```
import sys
from PyQt5.QtWidgets import *

class MainWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle('Generic Window')

        self.pushButtonOk = QPushButton('Ok', self)
        self.pushButtonOk.setGeometry(100, 100, 100, 100)
        self.pushButtonOk.clicked.connect(buttonClickedHandler)

    def buttonClickedHandler():
        QMessageBox.information(None, 'Message', 'Ok')

def main():
    app = QApplication(sys.argv)
    mainWindow = MainWindow()
    mainWindow.show()
    app.exec()

main()
```

Tabii istersek slot yerine doğrudan bir lambda ifadesi de kullanabiliriz. Örneğin:

```
self.pushButtonOk = QPushButton('Ok', self)
self.pushButtonOk.setGeometry(100, 100, 100, 100)
self.pushButtonOk.clicked.connect(lambda: QMessageBox.information(None, 'Message', 'Ok'))
```

Penecelerin Kapatılması

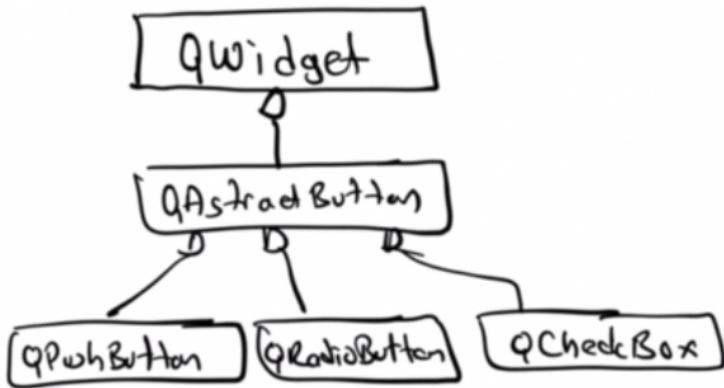
Bir pencere QWidget sınıfından gelen close metoduyla kapatılabilir. Söz konusu pencere ana pencereye close ana pencerenin kapanmasına ve dolayısıyla da mesaj döngüsünden (yani exec metodundan) çıkışmasına yol çamaktadır. O halde bizim GUI programı sonlandırmak için tek yapacağımız şey ana pencereyi self.close() metoduyla kapatmaktadır.

PyQt'de Temel GUI Elemenlar

Bu bölümde Qt'deki temel alt pencere sınıfları ele alınacaktır. GUI uygulamalar bu sınıflar türündne nesneler yaratılarak gerçekleştirilmektedir.

QPushButton Sınıfı

PushButton en çok kullanılan standart GUI elemanıdır. Belli bir olayı başlatmak ya da bitirmek için kullanılır. QPushButton sınıfı QAbstractButton sınıfından türetilmiştir. Aslında QPushButton sınıfının QCheckBox ve QRadioButton sınıfları ile ortak elemanları vardır. Bu ortak elemanlar QAbstractButton sınıfında toplanmıştır. QAbstractButton sınıfı da QWidget sınıfından türetilmiş durumdadır.



QPushButton nesnesi bir düğme ve üzerinde bir yazıyla karakterize edilmiştir. Bu düğmeye tıklanıp el farden çakıldığından sınıfın QAbstractButton sınıfından gelen clicked isimli sinyali tetiklenir.

QCheckBox Sınıfı

Bu sınıf bir kutu ve onun yanında bir yazdan oluşan bir alt pencereyi temsil etmektedir. Bu pencereye tıklandığında kutu çarpılanır, bir daha tıklandığında çarpı kalındırılır. Programcı check box nesnesinden onun çarpılı olup olmadığı sonucunu elde etmek ister. Kontrolün çarpılı olup olmadığı QCheckBox sınıfının QAbstractButton sınıfından gelen isChecked metodu ile elde edilebilir. Örneğin:

```
import sys
from PyQt5.QtWidgets import *

class MainWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle('Generic Window')

        self.pushButtonOk = QPushButton('Ok', self)
        self.pushButtonOk.move(10, 10)
        self.pushButtonOk.clicked.connect(self.buttonClickHandler)

        self.checkBox = QCheckBox('test', self)
        self.checkBox.move(100, 10)

    def buttonClickHandler(self):
        QMessageBox.information(self, 'Message', 'checked' if self.checkBox.isChecked() else
'unchecked')

def main():
    pass
```

```

app = QApplication(sys.argv)
mainWindow = MainWindow()
mainWindow.show()
app.exec()

main()

```

QCheckBox sınıfının QAbstractButton sınıfından gelen setChecked metodu ile biz seçenek kutularını programlama yoluyla da çarpılayabiliriz ya da çarpısını kaldırabiliriz. Örneğin:

```

self.checkBox = QCheckBox('test', self)
self.checkBox.move(100, 10)
self.checkBox.setChecked(True)

```

QCheckBox sınıfının setTristate metodu True olarak girilirse üç konumlu seçenek kutusu oluşturulmuş olur. Üç konumlu seçenek kutularında kontrolün konumu checkState metodu ile alınmaktadır. Benzer biçimde set etme de setCheckState metodu ile yapılmaktadır. Bu metodlar şu üç değeri almaktadır:

0	Unchecked
1	PartiallyChecked
2	Checked

Örneğin:

```

self.checkBox = QCheckBox('test', self)
self.checkBox.move(100, 10)
self.checkBox.setTristate(True)
self.checkBox.setCheckState(1)

```

Üç konumlu seçenek kutusunun değeri de şöyle yazdırılabilir:

```

def buttonClickHandler(self):
    QMessageBox.information(self, 'Message', ['Unchecked', 'Indeterminate',
'Checked'][self.checkBox.checkState()])

```

Radyo Düğmeleri (Radio Buttons)

Radyo Düğmeleri bir grup düğmenin yalnızca birinin seçilebildiği bir durum oluşturmak için kullanılır. Aynı pencerenin alt pencerelerindeki radyo düğmeleri bir grup oluşturmaktadır. Bir dayo düğmesine tıklandığında daha önce çarpılanmiş olan düğme yerine tıklanan çarpılanır. Tabii radyo düğmelerinin bir tane yaratılması anlamsızdır. Radyo düğmeleri PyQt'de QRadioButton sınıfıyla temsil edilmiştir. Radyo düğmelerindne programcı nihai olarak hangi düğmenin seçilmiş olduğu bilgisini alır. Bunun maalesef çok pratik bir yolu yoktur. Programcı tek tek bu düğmelere bakarak isChecked metodu ile kontrol yapar. Örneğin:

```

import sys
from PyQt5.QtWidgets import *

class MainWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle('Generic Window')

        self.pushButtonOk = QPushButton('Ok', self)
        self.pushButtonOk.move(10, 10)
        self.pushButtonOk.clicked.connect(self.buttonClickHandler)

        self.radioButtonA = QRadioButton('A', self)
        self.radioButtonA.move(150, 10)

```

```

self.radioButtonB = QRadioButton('B', self)
self.radioButtonB.move(150, 30)

self.radioButtonC = QRadioButton('C', self)
self.radioButtonC.move(150, 50)

self.radioButtonD = QRadioButton('D', self)
self.radioButtonD.move(150, 70)

self.radioButtonE = QRadioButton('E', self)
self.radioButtonE.move(150, 90)

def buttonClickHandler(self):
    if self.radioButtonA.isChecked():
        result = 'A Checked'
    elif self.radioButtonB.isChecked():
        result = 'B Checked'
    elif self.radioButtonC.isChecked():
        result = 'C Checked'
    elif self.radioButtonD.isChecked():
        result = 'D Checked'
    elif self.radioButtonE.isChecked():
        result = 'E Checked'
    else:
        result = 'Nothing Checked'

    QMessageBox.information(self, 'Message', result)

def main():
    app = QApplication(sys.argv)
    mainWindow = MainWindow()
    mainWindow.show()
    app.exec()

main()

```

Hangi radyo düğmesinin seçildiğini başka yöntemlerle de anlayabiliriz. Örneğin radyo düğmelerini bir listeye yerleştirip bir döngü içerisinde hangisinin seçili olduğuna bakılabilir:

```

import sys
from PyQt5.QtWidgets import *

class MainWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle('Generic Window')
        self.radioList = []

        self.pushButtonOk = QPushButton('Ok', self)
        self.pushButtonOk.move(10, 10)
        self.pushButtonOk.clicked.connect(self.buttonClickHandler)

        rb = QRadioButton('A', self)
        rb.move(150, 10)
        self.radioList.append(rb)

        rb = QRadioButton('B', self)
        rb.move(150, 30)
        self.radioList.append(rb)

        rb = QRadioButton('C', self)
        rb.move(150, 50)
        self.radioList.append(rb)

```

```

rb = QRadioButton('D', self)
rb.move(150, 70)
self.radioList.append(rb)

rb = QRadioButton('E', self)
rb.move(150, 90)
self.radioList.append(rb)

def buttonClickHandler(self):
    for rb in self.radioList:
        if rb.isChecked():
            break
    else:
        rb = None

    QMessageBox.information(self, 'Message', rb.text() + ' Checked' if rb else 'Nothing Checked')

def main():
    app = QApplication(sys.argv)
    mainWindow = MainWindow()
    mainWindow.show()
    app.exec()

main()

```

Bir pencerenin tüm radyo düğmeleri aynı grubu oluşturmaktadır. Pekiyi aynı pencerede birden fazla grup oluşturabiliyoruz? İşte bu durumda radyo düğmelerini ana pencerenin altındaki başka pencerelere yerleştirmek gereklidir. Bunun tipik olarak QGroupBox pencereleri bulundurulmuştur. Örneğin:

```

import sys
from PyQt5.QtWidgets import *

class MainWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle('Generic Window')
        self.resize(640, 300)

        self.pushButtonOk = QPushButton('Ok', self)
        self.pushButtonOk.move(10, 10)
        self.pushButtonOk.clicked.connect(self.buttonClickHandler)

        self groupBox1 = QGroupBox(self)
        self.groupBox1.setGeometry(10, 50, 200, 80)

        self groupBox2 = QGroupBox(self)
        self.groupBox2.setGeometry(270, 50, 200, 80)

        self.radioButtonA = QRadioButton('A', self.groupBox1)
        self.radioButtonA.move(10, 10)

        self.radioButtonB = QRadioButton('B', self.groupBox1)
        self.radioButtonB.move(10, 30)

        self.radioButtonC = QRadioButton('C', self.groupBox1)
        self.radioButtonC.move(10, 50)

        self.radioButtonD = QRadioButton('D', self.groupBox2)
        self.radioButtonD.move(10, 10)

```

```

self.radioButtonE = QRadioButton('E', self.groupBox2)
self.radioButtonE.move(10, 30)

self.radioButtonF = QRadioButton('F', self.groupBox2)
self.radioButtonF.move(10, 50)

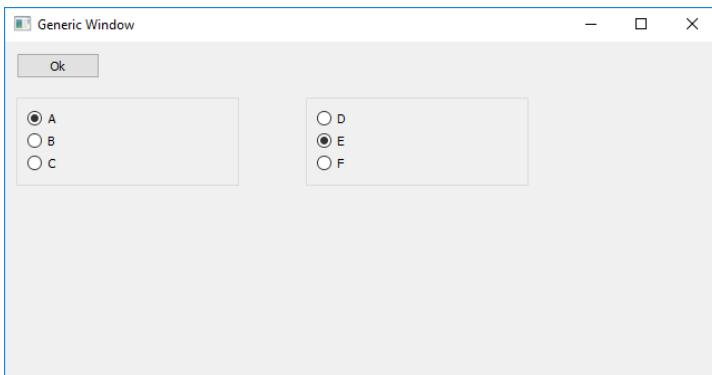
def buttonClickHandler(self):
    pass

def main():
    app = QApplication(sys.argv)
    mainWindow = MainWindow()
    mainWindow.show()
    app.exec()

main()

```

Programın ekran görüntüsü şöyledir:



QLabel Kullanımı

İsmine QLabel denilen amacı yalnızca alt pencere içerisinde yazı göstermek olan basit bir kontrol vardır. Bu kontrol bizden bir yazıyi alır, transparan bir zemine bu yazıyı yazar. Dolayısıyla GUI ekranlarda sabit yazılar bu sınıfla oluşturulmaktadır. Sınıfın QWidget sınıfından gelen text ve setText metotları ilgili yazıyı alıp set etmek için kullanılmaktadır. Örneğin:

```

import sys
from PyQt5.QtWidgets import *

class MainWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle('Generic Window')
        self.resize(300, 200)

        self.labelMsg = QLabel('Bu bir denemedir', self)
        self.labelMsg.move(10, 10)

def main():
    app = QApplication(sys.argv)
    mainWindow = MainWindow()
    mainWindow.show()
    app.exec()

main()

```

QLabel içeriisndeki yazıyı istediğimiz bir fontla da yazdırabiliriz. Bunun için bir QFont nesnesi oluşturup QLabel sınıfının setFont metodu çağrılmalıdır. Örneğin:

```
class MainWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle('Generic Window')
        self.resize(300, 200)

        self.labelMsg = QLabel('Bu bir denemedir', self)
        self.labelMsg.move(10, 10)
        self.labelMsg.setFont(QFont('Times New Roman', 20))
```

Aslında mevcut fontu font metodu ile alıp yalnızca onun boyutunu da değiştirebiliriz. Örneğin:

```
import sys
from PyQt5.QtWidgets import *
from PyQt5.QtGui import *

class MainWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle('Generic Window')
        self.resize(300, 200)

        self.labelMsg = QLabel('Bu bir denemedir', self)
        self.labelMsg.move(10, 10)
        font = self.labelMsg.font()
        font.setPointSize(20)
        self.labelMsg.setFont(font)

def main():
    app = QApplication(sys.argv)
    mainWindow = MainWindow()
    mainWindow.show()
    app.exec()

main()
```

Tek Satırlı Edit Alanlarının Oluşturulması

Kullanıcıdan tek satırı bilgi almak için QLineEdit isimli sınıf kullanılmaktadır. Bir QLineEdit nesnesi diğer GUI öğeler gibi sınıfın başlangıç metoduna yaratılır. Edit alanındaki yazı yine QWidget sınıfından gelen text metoduna elde edilmektedir. Örneğin:

```
import sys
from PyQt5.QtWidgets import *
from PyQt5.QtGui import *

class MainWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle('Generic Window')
        self.resize(300, 150)

        self.labelMsg = QLabel('Adı Soyadı', self)
        self.labelMsg.move(10, 10)

        self.lineEditName = QLineEdit(self)
        self.lineEditName.move(10, 24)
        self.lineEditName.resize(200, 20)
```

```

self.labelNo = QLabel('No', self)
self.labelNo.move(10, 50)

self.lineEditNo = QLineEdit(self)
self.lineEditNo.move(10, 64)
self.lineEditNo.resize(200, 20)

self.pushButtonOk = QPushButton('Ok', self)
self.pushButtonOk.move(120, 110)
self.pushButtonOk.clicked.connect(self.buttonOkClickedHandler)

self.pushButtonQuit = QPushButton('Quit', self)
self.pushButtonQuit.move(200, 110)
self.pushButtonQuit.clicked.connect(self.buttonQuitClickedHandler)

def buttonOkClickedHandler(self):
    msg = self.lineEditName.text() + ', ' + self.lineEditNo.text()
    QMessageBox.information(self, 'Message', msg)

def buttonQuitClickedHandler(self):
    self.close()

def main():
    app = QApplication(sys.argv)
    mainWindow = MainWindow()
    mainWindow.show()
    app.exec()

main()

```

QLineEdit sınıfının setMaxLength isimli metodu edit alanındaki karakter sayısını kısıtlamak için kullanılmaktadır. Sınıfın setAlignment metodu ile biz hizalama sağlayabiliriz. Hizalama değerleri şunlardır:

```

Qt.AlignLeft
Qt.AlignRight
Qt.AlignHCenter
Qt.AlignJustify

```

Örneğin:

```
self.lineEditName.setAlignment(Qt.AlignRight)
```

QLineEdit sınıfının setReadOnly metotları edit kontrolünün yalnızca okunabilir yapmaktadır.

Çok Satırlı Edit Alanlarının Oluşturulması

Çok satırlı edit alanları adeta notepad benzeri bir editör gibidir. PyQt'de bu edit alanları QTextEdit sınıfıyla temsil edilmiştir. Örneğin:

```

import sys
from PyQt5.QtWidgets import *
from PyQt5.QtGui import *

class MainWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle('Generic Window')
        self.resize(600, 350)

        self.textEdit = QTextEdit(self)

```

```

    self.textEdit.setGeometry(10, 10, 580, 330)
    self.textEdit.setFont(QFont('Arial', 14))

def main():
    app = QApplication(sys.argv)
    mainWindow = MainWindow()
    mainWindow.show()
    app.exec()

main()

```

Yine çok satırlı edit alanından tüm yazı text metodu ile alınıp setText metodu ile geri set edilebilir. Örneğin bir dosyanın içeriğini QTextEdit nesnesine yerleştirebiliriz:

```

import sys
from PyQt5.QtWidgets import *
from PyQt5.QtGui import *

class MainWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle('Generic Window')
        self.resize(600, 350)

        self.textEdit = QTextEdit(self)
        self.textEdit.setGeometry(10, 10, 580, 330)
        self.textEdit.setFont(QFont('Consolas', 14))

    try:
        f = open('generic.py')
        s = f.read()
        self.textEdit.setText(s)

    except:
        QMessageBox.warning(self, 'Error', 'file error!')


def main():
    app = QApplication(sys.argv)
    mainWindow = MainWindow()
    mainWindow.show()
    app.exec()

main()

```

QListWidget Kullanımı

Bu kontrole pek çok framework'te "listbox" da denilmektedir. QListWidget ismi üzerinde bir grup öğeyi tutan bir liste oluşturur ve kullanıcının bu listeden seçim yapabilmesini sağlar. Kontrolün kullanımı şöyledir:

1) Önce QListWidget türünden nesne yaratılır ve pencere konumlandırılır. Örneğin:

```

self.listWidgetCities = QListWidget(self)
self.listWidgetCities.move(10, 10)
self.listWidgetCities.resize(200, 300)

```

2) Sıra öğelerin listeye eklenmesine gelmiştir. Aslında QListWidget içeresine QListWidgetItem nesneleri eklenir. tipik olarak önce QListWidgetItem nesnesi yarato sonra QListWidget sınıfının addItem metodu ile ekleme yapılabilir. Ancak doğrudan string alan bir addItem metodu da vardır. Bu metot kendi içerisinde QListWidget nesnesi yaratıp eklemeyi yapmaktadır. Örneğin:

```
self.listViewCities.addItem('Adana')
self.listViewCities.addItem('Adiyaman')
self.listViewCities.addItem('Eskişehir')
```

Aslında aynı işlem sınıfın addItems() metoduyla da tek hamlede yapılabilmektedir. AddItems dolaşılabilir bir string'lerden oluşan bir nesneyi parametre olarak almaktadır. Örneğin:

```
self.listViewCities.addItems(['Adana', 'Adiyaman', 'Eskişehir', 'Balıkesir', 'Ağrı', 'Muğla',
                             'Sakarya', 'Siirt', 'Konya', 'Kocaeli'])
```

3) QListWidget'ta seçili olan elemanlar sınıfın selectedItems() metodu ile alınabilir. Ancak bu metod bize seçili olan tüm nesneleri bir QListWidgetItem olarak vermektedir. Eğer tek seçim yapılmışsa programcı bu listenin ilk elemanını alabilir. Örneğin:

```
def buttonOkClickedHandler(self):
    QMessageBox.information(self, 'Selected Items', self.listViewCities.selectedItems()[0].text())
```

Tabii listede hiçbir eleman seçili de olmayabilir. Bu durmda selectedItems() bize boş bir liste verir. İşte programcının bunu kontrol etmesi gerekebilmektedir. Örneğin:

```
def buttonOkClickedHandler(self):
    if len(self.listViewCities.selectedItems()) != 0:
        QMessageBox.information(self, 'Selected Items', self.listViewCities.selectedItems()[0].text())
```

Aslında seçilen aktif elemanın indeks currentRow() metoduyla alınabilir. Zaten hiçbir eleman seçilmemişse bu metod bize -1 vermektedir. Biz QListWidget nesnesi oluşturulduğunda sınıfın setCurrentRow() isimli metoduyla beli bir eleman seçili duruma da getirilebilmektedir.

Listeleme kutularında bir eleman üzerine çift tıklama çok karşılaşılan bir eylemdir. Bir eleman üzerine çift tıklandığında itemDoubleClicked() isimli sinyal tetiklenmektedir. Bu sinyal için çağrılacak slotun QListWidgetItem türünden seçilen elemanı belirten bir parametresi vardır. Örneğin:

```
self.listViewCities.itemDoubleClicked.connect(self.onListItemSelected)
...
def onListItemSelected(self, item):
    QMessageBox.information(self, 'Selected Item', item.text())
```

Seçili eleman değiştiğinde de currentRowChanged() isimli sinyal tetiklenmektedir. Bu sinyalin parametresi int türden seçilen elemanın indeksini belirtmektedir. Böylece programcı eleman değiştiğinde yeni eleman hakkında bilgileri gösterebilmektedir. Örneğin QListWidget içerisinde kişilerin isimleri olabilir. Kullanıcı bir kişiyi seçtiğinde onun fotoğrafı otomatik olarak gesterilmek istenebilir. Seçili eleman değiştiğinde tetiklenen diğer bir sinyal de currentItemChanged() isimli sinyaldır. Bu sinyalin eski seçilen eleman ve yeni seçilen eleman olmak üzere QListWidgetItem türünden iki parametresi vardır. Örneğin:

```
self.listViewCities.currentItemChanged.connect(self.onCurrentItemChanged)
...
def onCurrentItemChanged(self, old, new):
    print(new.text())
```

QListWidget içerisindeki nesnelerin hepsini silmek için clear() isimli metod, liste içerisindeki elemanların sayısını almak için ise count() isimli metod kullanılır.

QListWidget sınıfının diğer elemanları hakkında bilgi dokümanlardan elde edilebilir.

resize Sinyali

Bir pencerenin boyutu değiştirildiğinde resizeEvent isimli metot çağrılmaktadır. Böylece programcı pencere boyutunun değiştiğini otomatik olarak anlayabilir. Bu işlemin sinyal-slot mekanizmasıyla değil resize isimli metodun yazılmışıyla yapıldığına dikkat ediniz. Örneğin:

```
import sys
from PyQt5.QtWidgets import *
from PyQt5.QtGui import *

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.resize(700, 500)

    def resizeEvent(self, *_args):
        print(".", end="")

def main():
    app = QApplication(sys.argv)
    mainWindow = MainWindow()
    mainWindow.show()
    app.exec()

main()
```

Programcı resizeEvent metodunda pencerenin yeni genişlik ve yüksekliğini alabilir.

PyQt'de Resimlerin Görüntülenmesi

PyQt'de resimler yaz çizim yoluyla ya da QLbale yoluyla görüntülenebilmektedir. Maalesef PyQt'de diğer framework'lerde olduğu gibi "picturebox" benzeri bir kontol yoktur.

Bir QLabel nesnesi yazının yanı sıra aslında zemininde bir resim de gösterebilmektedir. Bunun tek yapılacak şey sınıfın setPixmap metoduyla resim dosyasını belirtmektir. Bu metot bir QPixmap nesnesini parametre olarak alır. QPixmap nesnesi de resmin yolk ifadesi verilerek oluşturulabilmektedir. QPixmap temel pek çok dosya formatını desteklemektedir.

QLabel nesnesinin zeminine setPixmap metodu ile bir resim yerleştirildiğinde label otomatik olarak resim boyutuna getirilir. Aşağıdaki örnekte önce QPixmap ile resim diskten yüklenerek kullanıma hazır hale getirilmiştir. Sonra ana pencere tam bu resmin boyutuna ayarlanmıştır. Böylece ana resmin içerisinde bütün resm görüntülenmektedir. Örneğin:

```
import sys
from PyQt5.QtWidgets import *
from PyQt5.QtGui import *

class MainWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle('Generic Window')

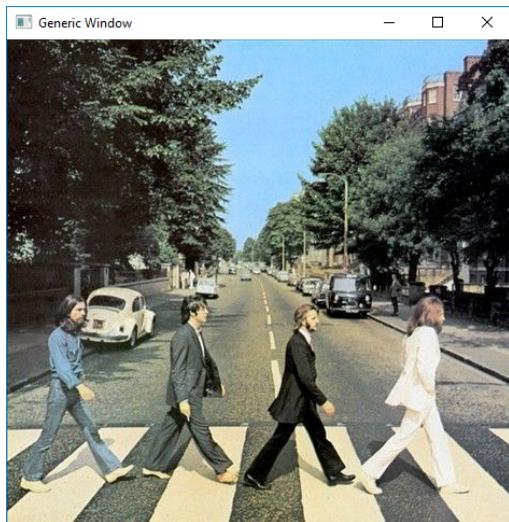
        pixmap = QPixmap('AbbeyRoad.jpg')
        self.resize(pixmap.size())

        self.labelAbbeyRoad = QLabel(self)
        self.labelAbbeyRoad.setPixmap(pixmap)

def main():
    app = QApplication(sys.argv)
    mainWindow = MainWindow()
    mainWindow.show()
```

```
app.exec()  
main()
```

Görüntü aşağıdaki gibi olacaktır:



Tabii resmin boyutlandırılarak görüntülenmesi de önemlidir. Maalesef QLabel bu kadar esnek değildir. Bu tür durumlarda resmi çizim metotlarıyla çizmek çok daha fazla olnaklar sunmaktadır.

QComboBox Kullanımı

Combox kontrolü listbox (yani Qt'teki QListWidget) kontrolünün farklı bir biçimidir. Bir liste içerisindeki tek bir elemanı seçmek kullanılır. Combobox'ın list alanı kontrole tıklandığında açılmaktadır . Combobox kullanımı listbox'a oldukça benzerdir:

1) QCombox türünden nesne yaratılır ve konumlandırılır. Örneğin:

```
self.comboBoxCities = QComboBox(self)  
self.comboBoxCities.move(10, 10)  
self.comboBoxCities.resize(200, 30)
```

Combox'ın boyutu yalnızca başlık kısmının boyutuyla belirtilmektedir.

2) Combobox'a nesneler sınıfın addItem ya da addItems metotlarıyla eklenirler. Örneğin:

```
self.comboBoxCities.addItem('Ankara')  
self.comboBoxCities.addItem('İzmir')  
self.comboBoxCities.addItem('Adana')  
self.comboBoxCities.addItem('Eskişehir')  
self.comboBoxCities.addItem('Kütahya')  
self.comboBoxCities.addItem('Antalya')
```

3) Seçilen elemanın kendisi current isimli metotla, onun yazısı da currentText isimli metotla elde edilebilir. Örneğin:

```
self.pushButtonOk.clicked.connect(self.buttonOkClickedHandler)  
...  
def buttonOkClickedHandler(self):  
    print(self.comboBoxCities.currentText())
```

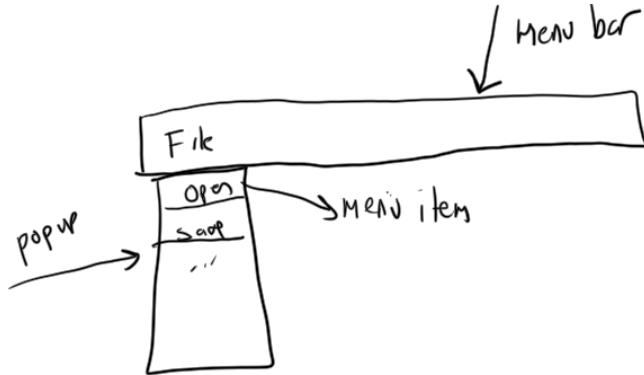
Combox'ın başlığındaki yazı kullanıcı tarafından değiştirilebilsin isteniyorsa setEditable metodu True ile çağrılmalıdır.

Combox'ta yeni bir eleman seçildiğinde currentIndexChanged isimli sinyal tetiklenir. Böylece biz yeni bir eleman seçildiğinde otomatik bazı işlemeleri yapabiliriz.

Menülerin Oluşturulması

Menüler bir GUI programının en önemli elemanlarındadır. Menüler Windows gibi bazı işletim sistemlerinde ve pencere yöneticilerinde pencereye özgüdür ve pencere içerisinde görüntülenir. Mac OS X gibi bazılarında ise masaüstünde tek bir menü bulunmaktadır. Aktif uygulama değişikçe bu menü o uygulamanın menüsü olur.

Menülerdeki ana çubuğa menü çubuğu (menu bar) denilmektedir. Menü çubuğuna bağlı olan menü pencerelerine ise popup denir. Popup pencereler üzerinde menü elemanları vardır:



Menülerin ve araç çubuklarının QWidget penceresinde bulundurulması mümkün olsa da oldukça zahmetlidir. Bunun için ana pencere görevi yapacak QMainWindow sınıfı düşünülmüştür. Yani menü ve araç çubuğu içeren uygulamaların ana pencereleri için QWidget sınıfı değil, QMainWindow sınıfı kullanılmalıdır.

Menü çubuğu Qt'de QMenuBar sınıfıyla, Popup pencereleri ise QMenu sınıfıyla temsil edilmektedir. Menü elemanları ise QAction sınıfıyla temsil edilmiştir. QAction nesneleri yalnızca menü elemanları olarak değil aynı zamanda araç çubuğu (toolbar) elemanları olarak da kullanılmaktadır. Menu elemanı seçildiğinde ya da araç çubuğu elemanına tıklandığında emit edilecek sinyaller QAction sınıfının içerisindeindedir.

QMainWindow nesnesi yaratıldığında zaten bir QMenuBar nesnesi de yaratılmaktadır. Bu nesneneye biz QMainWindow sınıfının menuBar metodu ile erişebiliriz. Yani bizim ayrıca QMenuBar nesnesi yaratmamıza gerek yoktur.

Bir Popup QMenu sınıfı ile yaratılır ve QMenuBar sınıfının addMenu metoduyla menü çubuğuna eklenir. Örneğin:

```
filePopup = QMenu('&File')
self.menuBar().addMenu(filePopup)
```

Ya da popup eklemek için doğrudan QMenuBar sınıfının string parametreli AddMenu metodu da kullanılabilir:

```
filePopup = self.menuBar().addMenu('&File')
```

Burada aslında addMenu kendisi bir QMenu nesnesi yaratıp yine onu eklemektedir. Ancak PyQt5'te muhtemel bir bug yüzünden QMenu nesnesinin kendisi addMenu ile eklendiğinde sorun oluşmaktadır. Bu nedenle PyQt5'te eklemeyi string parametreli addMenu metoduyla aşağıdaki gibi yapınız:

```
filePopup = self.menuBar().addMenu('&File')
```

Bu biçimde addMenu kendi içerisinde yarattığı QMenu nesnesini bize geri dönüş değeri olarak vermektedir.

Popup pencereler menü çubuğu eklendikten sonra sıra QAction nesnelerinin (yani menü elemanlarının) popup pencerelere eklenmesine gelmiştir. Bunun için QMenu sınıfının addAction metotları kullanılır. addAction metotları tek parametreli ya da iki parametreli biçimde kullanılabilir:

```
QAction addAction (QString text)
QAction addAction (QIcon icon, QString text)
```

Tek parametreli kullanımda verilen yazı menü elemanında görüntülenmektedir. İki parametreli kullanımda hem yazı hem de simge metoda verilmektedir. addAction metotları geri dönüş değeri olarak yaratlan QAction nesnesini vermektedir. Simgiler QIcon sınıfıyla temsil edilmişlerdir. Simgeler tipik olarak 16x16 ya da 32x32 .ico, .png ya da .bmp dosya formatlarına ilişkin olabilir. Bir QIcon nesnesi dosyanın yol ifadesi belirtilerek QIcon('open.png') biçiminde yaratılabilir. Örneğin:

```
import sys
from PyQt5.QtWidgets import *

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle('Generic Window')
        self.resize(600, 400)

        filePopup = self.menuBar().addMenu('&File')
        openAction = filePopup.addAction('&Open')
        closeAction = filePopup.addAction('&Close')

    def main():
        app = QApplication(sys.argv)
        mainWindow = MainWindow()
        mainWindow.show()
        app.exec()

main()
```

Şimdi de menü elemanlarının (yani QAction nesnelerine) aynı zamanda bir simge de ekleyelim. Bunun için öncelikle Open ve Close elemanları için 16x16'lık birer simge (icon) bulmamız gereklidir. Bedava simge bulunduran pek çok site vardır. Biz kursumuzda bunun için iconfinder.com sitesinden faydalanaçğız. Simgeleri .png formatında indirmek daha uygundur. Örneğin:

```
filePopup = self.menuBar().addMenu('&File')
openAction = filePopup.addAction(QIcon('Icons/open.png'), '&Open')
closeAction = filePopup.addAction(QIcon('Icons/close.png'), '&Close')
```

Aslında önce QAction nesneleri yaratıp sonra QMenu sınıfının addAction metotlarına bunları da verebiliriz. Ancak PyQt5 yine burada böcekler içermektedir. Bu nedenle PyQt5'te QAction nesnelerini ayrı yaratıp eklemek yerine doğrudan addAction metodu ile yaratınız.

Süphesiz menü eklemenin en önemli amacı menü elemanı seçildiğinde birşeyler yapmaktır. İşte QAction sınıfının triggered isimli sinyalleri menü elemanı seçildiğinde bir metodun çağrılmasını sağlamaktadır. Örneğin:

```
import sys
from PyQt5.QtWidgets import *
from PyQt5.QtGui import *

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle('Generic Window')
```

```

self.resize(600, 400)

filePopup = self.menuBar().addMenu('&File')

openAction = filePopup.addAction(QIcon('Icons/open.png'), '&Open')
closeAction = filePopup.addAction(QIcon('Icons/close.png'), '&Close')

openAction.triggered.connect(self.onOpenTriggered)
closeAction.triggered.connect(self.onCloseTriggered)

def onOpenTriggered(self, sender):
    QMessageBox.information(self, 'Mesaj', 'Open elemanı seçildi')

def onCloseTriggered(self, sender):
    QMessageBox.information(self, 'Mesaj', 'Close elemanı seçildi')

def main():
    app = QApplication(sys.argv)
    mainWindow = MainWindow()
    mainWindow.show()
    app.exec()

main()

```

İç içe popup menüler olabilir. Bu durumda QMenu elemanına addAction değil addMenu uygulamak gereklidir. Örneğin:

```

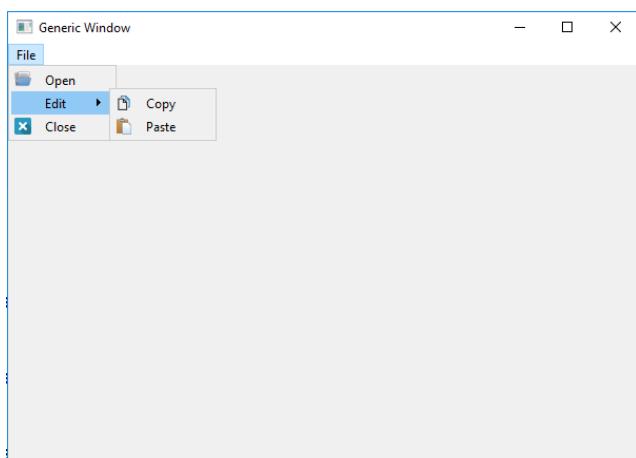
filePopup = self.menuBar().addMenu('&File')

openAction = filePopup.addAction(QIcon('Icons/open.png'), '&Open')
openAction.triggered.connect(self.onOpenTriggered)

editPopupAction = filePopup.addMenu('&Edit')
copyAction = editPopupAction.addAction(QIcon('Icons/copy.png'), 'Copy')
pasteAction = editPopupAction.addAction(QIcon('Icons/paste.png'), 'Paste')

closeAction = filePopup.addAction(QIcon('Icons/close.png'), '&Close')
closeAction.triggered.connect(self.onCloseTriggered)

```



QAction nesnelerine birer kısa yol tuşu da atayabiliriz. Bunun için QAction sınıfının setShortcut isimli metodu kullanılmaktadır. Bu metot bizden QKeySequence türünden bir değer alır. Bu sınıfın standart bazı kısayol tuş elemanları vardır. Örneğin:

```
filePopup = self.menuBar().addMenu('&File')

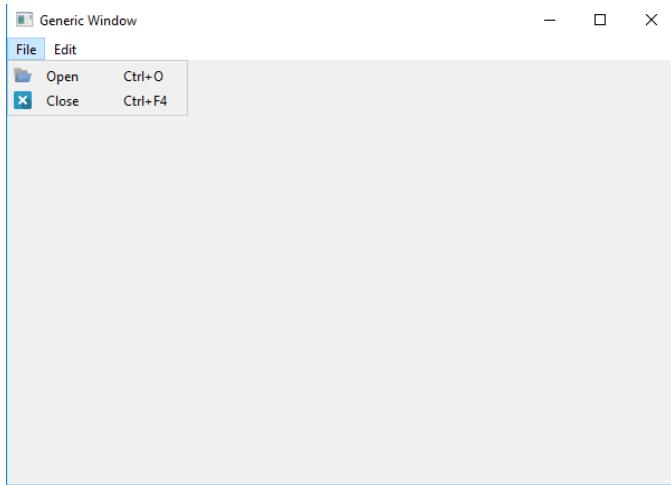
openAction = filePopup.addAction(QIcon('Icons/open.png'), '&Open')
openAction.setShortcut(QKeySequence.Open)
openAction.triggered.connect(self.onOpenTriggered)

editPopupAction = self.menuBar().addMenu('&Edit')

copyAction = editPopup.addAction(QIcon('Icons/copy.png'), 'Copy')
copyAction.setShortcut(QKeySequence.Copy)

pasteAction = editPopup.addAction(QIcon('Icons/paste.png'), 'Paste')
pasteAction.setShortcut(QKeySequence.Paste)

closeAction = filePopup.addAction(QIcon('Icons/close.png'), '&Close')
closeAction.setShortcut(QKeySequence.Close)
closeAction.triggered.connect(self.onCloseTriggered)
```



Biz kısa yol tuşu olarak istediğimiz bir tuş kombinasyonunu da atayabiliriz. Bunun için QKeySequence nesnesini tuş kombinasyonunu belirten yazıyla yaratmalıyız. Örneğin:

```
copyAction.setShortcut('Ctrl+k')
```

Kombinasyonların arasında '+' karakterinin getirildiğine dikkat ediniz.

Bazen bir eylemde menü elemanlarının konumlarının değiştirilmesi gerekebilir. Bu durumda menü elemanlarının sınıfın veri elemanı yapılması uygun olur. Örneğin:

```
self.fruitPopup = self.menuBar().addMenu('&Fruit')

self.bananaAction = self.fruitPopup.addAction('&Banana')
self.bananaAction.setCheckable(True)

self.appleAction = self.fruitPopup.addAction('&Apple')
self.appleAction.setCheckable(True)
```

QAction sınıfının setCheckable metodu menü elemanı her seçildiğinde onu "checked" ya da "unchecked" duruma getirmek kullanılır. Bu metot True parametresiyle çağrılırsa (default False durumdadır) bu otomatik

checked/unchecked sağlanmış olmaktadır. Eğer QAction nesnesi "checkable" durumdaysa biz onu yine setChecked metodu ile "checked" ya da "unchecked" duruma getirebiliriz.

Bir QAction nesnesi "enabled" ya da "disabled" durumda olabilir. Default durum "enabled" biçimdedir. "Disabled" durumda nesne seçilemez. Bazen programlarda o anda kullanılması anlamlı olmayan menü ya da araç çubuğu elemanları "disabled" yapılır. Örneğin File opoup menüsünde "open" ve "close" isimli iki menü elemanı olsun. Tek dökümanlı bir uygulamada doküman henüz "open" yapılmadan "close" yapılmayacağına göre baştan "open" menüsünün "enabled", "close" menüsünün "disabled" olması uygundur. Ancak kullanıcı "open" yaptıktan sonra artık tam ters durum olmalıdır. Bu işlemin kodu şöyle oluşturulabilir:

```
filePopup = self.menuBar().addMenu('&File')

self.openAction = filePopup.addAction(QIcon('Icons/open.png'), '&Open')
self.openAction.setShortcut(QKeySequence.Open)
self.openAction.triggered.connect(self.onOpenTriggered)

self.closeAction = filePopup.addAction(QIcon('Icons/close.png'), '&Close')
self.closeAction.setShortcut(QKeySequence.Close)
self.closeAction.setEnabled(False)
self.closeAction.triggered.connect(self.onCloseTriggered)

...

def onOpenTriggered(self):
    self.openAction.setEnabled(False)
    self.closeAction.setEnabled(True)

def onCloseTriggered(self):
    self.openAction.setEnabled(True)
    self.closeAction.setEnabled(False)
```

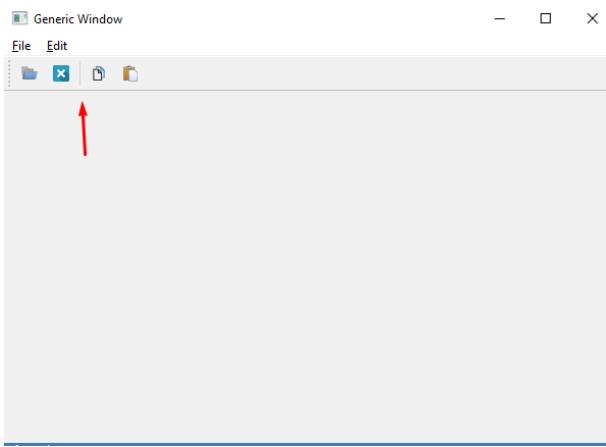
Araç Çubuklarının Kullanılması

Araç çubuklarını kullanabilmek için yine ana pencerenin QMainWindow sınıfından türetilmesi gerekmektedir. QMainWindow sınıfının addToolBar isimli metodu ile ana pencereye bir araç çubuğu eklenebilir. Bu metot bize QToolBar sınıfı türünden eklenen araç çubuğu nesnesini verir. addToolBar metodu bizden araç çubuğunu simgeleyen bir isim istemektedir.

Araç çubukları aslında genellikle (ama her zaman değil) menü elemanları için kısa yol amaçlı kullanılmaktadır. Araç çubuklarına da PyQt'de QAction nesneleri eklenmektedir. Yani başka bir deyişle QToolBar sınıfının da bir addAction metodu vardır. Genellikle programcı hem menülere hem de araç çubuklarına aynı action nesnesini verir.

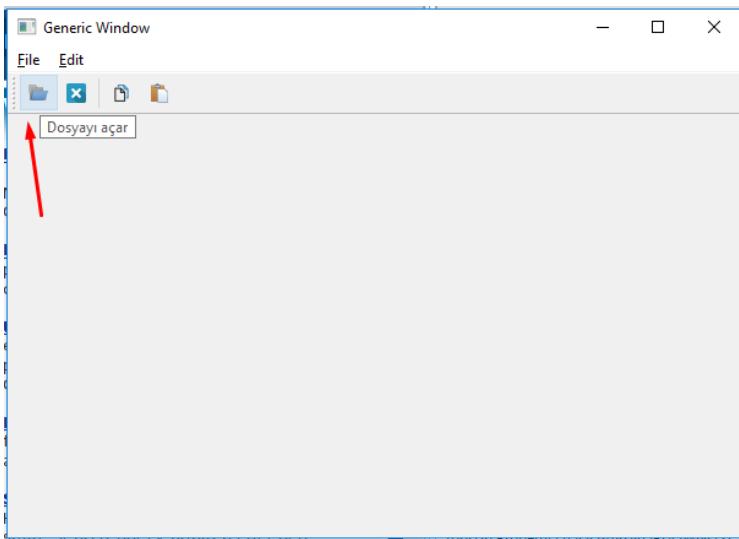
Menü ve araç çubuklarında (Yani QMenu ve QToolBar sınıflarında) addSeparator isimli metodlar bir ayırmaya çizgisi oluşturmaktadır. Böylece mantıksal gruplar birbirlerinden ayrılabilmektedir. Örneğin:

```
toolBar = self.addToolBar('MyToolBar')
toolBar.addAction(openAction)
toolBar.addAction(closeAction)
toolBar.addSeparator()
toolBar.addAction(copyAction)
toolBar.addAction(pasteAction)
```



QAction sınıfının setToolTip metodu fare menü ya da araç çubuğu elemanın üzerinde bekletildiğinde çıkartılacak yazıyı belirlemekte kullanılmaktadır. Örneğin:

```
openAction.setToolTip('Dosyayı açar')
```



QSlider Kullanımı

Slider yaygın biçimde kullanılan bir UI elemanıdır. Bir slider yürütece sahiptir. Kullanıcı yürüteci belli bir pozisyonu çekerek bazı ayarlamaları yapar. Özellikle volüm kontrolü, renk kontrolü gibi, ekran parlaklığı gibi derecesi olan büyülükler slider ile temsil edilme eğilimindedir. PyQt'de slider QSlider isimli bir sınıfta temsil edilmiştir. Bir slider oriyanşyon belirtilerek (default Vertical biçimdedir) aşağıdaki gibi yaratılır:

```
self.slider = QSlider(Qt.Horizontal, self)
self.slider.move(20, 20)
self.slider.resize(300, 40)
```

Default durumda minimum = 0, maksimum = 99 biçimdedir. Ancak biz sınıfın setMinimum ve setMaximum metodlarıyla bu değeri değiştrebiliriz. Slider'in konumu value metodu ile elde edilebilir ve programlama yoluyla setValue metodu ile set edilebilir. Sınıfın setTickInterval metodu ile tick çizgilerinin aralığı belirlenebilir. Ancak tick'lerin çıkışması için bizim sınıfın setTickPosition isimli property'si ile bunların pozisyonlarını belirlememiz gereklidir.

QSlider sınıfının valueChanged isimli sinyali pozisyon parametresi almaktadır. Yürüteç konum değiştiğinde arka planda bu sinyal tetiklenmektedir.

Örneğin:

```

import sys
from PyQt5.QtCore import *
from PyQt5.QtWidgets import *
from PyQt5.QtGui import *

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle('Generic Window')
        self.resize(600, 200)

        self.labelPos = QLabel('0', self)
        self.labelPos.setFont(QFont('Arial', 20))
        self.labelPos.move(340, 20)

        self.slider = QSlider(Qt.Horizontal, self)
        self.slider.move(20, 20)
        self.slider.resize(300, 40)
        self.slider.setTickPosition(QSlider.TicksBothSides)
        self.slider.setTickInterval(5)
        self.slider.valueChanged.connect(self.onValueChanged)

        self.buttonOk = QPushButton('Ok', self)
        self.buttonOk.move(20, 100)
        self.buttonOk.clicked.connect(lambda: QMessageBox.information(self, 'Message',
str(self.slider.value())))

    def onValueChanged(self, pos):
        self.labelPos.setText(str(pos))

def main():
    app = QApplication(sys.argv)
    mainWindow = MainWindow()
    mainWindow.show()
    app.exec()

main()

```



Pencere Renklerinin Değiştirilmesi

Qt5 ile birlikte pencerelerdeki renk değiştirme gibi görsel öğeler için CSS benzeri bir modele geçilmiştir. Ancak eski biçimdeki renk değiştirmeler desteklenmeye devam etmektedir.

QWidget sınıfındaki setPalette metodu renk değiştirmek için genel bir metottur. Bu metot bizden bir QPalette nesnesi alır. Bir QPalette nesnesi içerisinde pencerenin değişik öğelerinin renkleri belirtilebilmektedir. Bu öğelere role denilmektedir. QPalette sınıfının setColor isimli metodu ile biz bir rol için bir renk set edebiliriz. İstenirse yine QWidget sınıfının palette metodu ile aktif palet alınabilmektedir. Renkler QColor isimli sınıfta temsil edilmiştir. Bir rengi oluştururken onun RGB bileşenlerini [0,255] aralığında belirtmemiz gereklidir. Bu durumda örneğin pencerenin zemin rengini şöyle değiştirebiliriz:

```

palette = QPalette()
palette.setColor(QPalette.Window, QColor(255, 255, 0))
self.setPalette(palette)

```

Bu örnekteki QPalette.Window zemin rengi rolünü belirtmektedir.

QColor sınıfında bazı renkler önceden tanımlanmış olarak static eleman biçiminde bulundurulmaktadır. Bu renkler için RGB belirtmek yerine Qt.GlobalColor sınıfındaki elemanları kullanabiliriz. Örneğin:

```

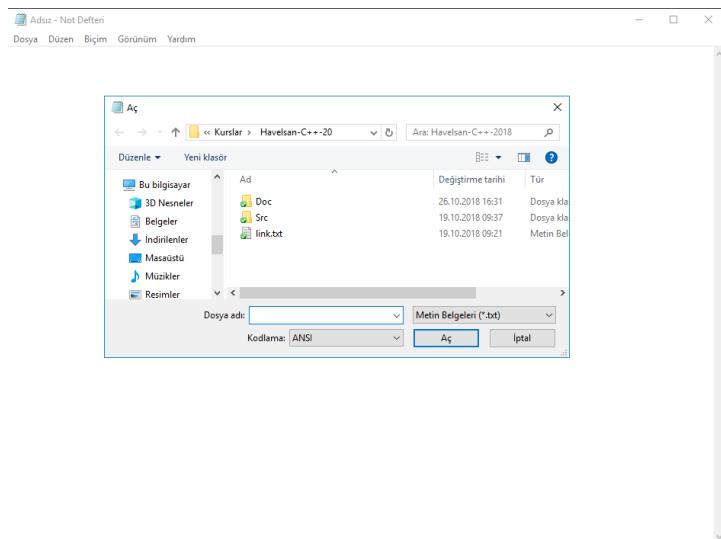
palette = QPalette()
palette.setColor(QPalette.Window, Qt.GlobalColor.yellow)
self.setPalette(palette)

```

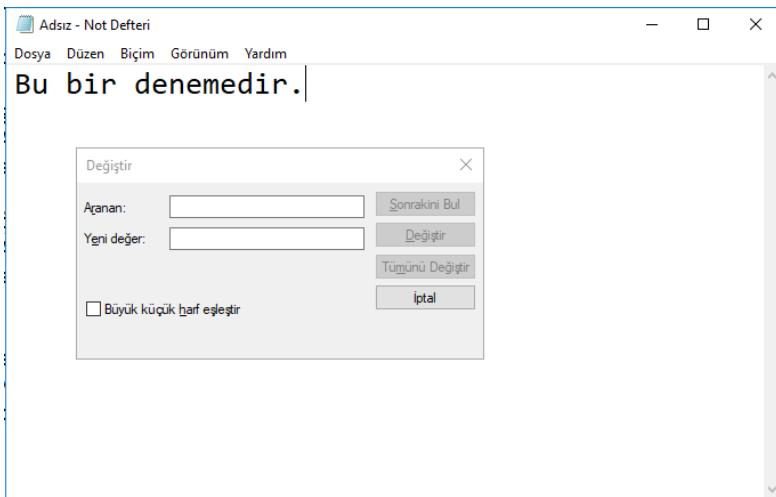
Bazı ana renkler doğrudan Qt modülün içerisinde de bulundurulmuştur. Yani biz örneğin Qt.red, Qt.blue, Qt.yellow biçiminde de bazı renkleri QColor olarak belirtebilmekteyiz.

Diyalog Pencereleri

Diyalog pencereleri "sahiplenilmiş (owned)" tarzda pencerelerdir. Bunlar hem ana pencere (top level windows) hem de alt pencere (child window) gibi davranışlarırlar. Kendi ücret pencerelerinin sınırları dışına çıkabilirler ancak her zaman üst pencerelerinin görsel bakımdan üzerinde görüntülenirler. Diyalog pencereleri Modal ve Modeless olmak üzere ikiye ayrılmaktadır. Modal bir pencere açıldığında arka plandaki pencereyle etkileşim ortadan kalkar. Ta ki modal pencere kapatılana kadar. MessageBox gibi OpenFileDialog gibi pek çok diyalog penceresi modal pencerelerdir. Örneğin:



Modeless diyalog pencerelerinde ise arka plan etkişelmiş devam etmektedir. Örneğin "Find and Replace" tarzı pencereler modeless pencerelerdir. Örneğin:



QtPy'da Modal Diyalog Pencerelerinin Oluşturulması

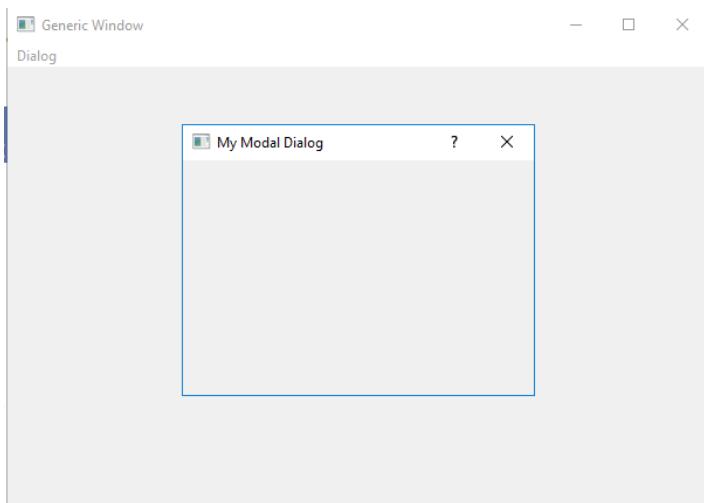
Modal diyalog pencereleri sırasıyla şu adımlardan geçilerek oluşturulmaktadır:

- 1) Modal diyalog penceresinin kendisi QDialog sınıfından türetilerek oluşturulmalıdır. Örneğin:

```
class MyModalDialog(QDialog):
    def __init__(self, parent):
        super().__init__(parent)
        self.setWindowTitle('My Modal Dialog')
        self.resize(300, 200)
```

- 2) Modal diyalog penceresi açılacağı zaman bu sınıf türünden bir nesne yaratılıp QDialog sınıfından gelen exec fonksiyonu çağrılır. Örneğin:

```
def onModalTriggered(self):
    myModalDialog = MyModalDialog(self)
    myModalDialog.exec()
```



Modal pencere yaratılırken onun üst penceresinin verildiğine dikkat ediniz. Yaratımın sırasında self ana pencereyi temsil etmektedir. Ayrıca QDialog sınıfından sınıf türettiğimizde taban sınıfın __init__ metodunu çağrırmayı unutmamalıyız.

3) Dialog pencerelerinin içerisinde genellikle etkileşim için GUI elemanları bulundurulur. Örneğin en azıncan modal dialog pencerelerinde bir Ok ve Cancel tuşları bulundurulmaktadır.

4) Diyalog pencerelerini kapatmak için close metodu değil QDialog sınıfından gelen done isimli metod çağrılmalıdır. Bu metod hangi nedenle çıktılığını belirten bir parametre almaktadır.

Modal diyalog penceresine ilişkin bir örnek aşağıda verilmiştir:

```
import sys
from PyQt5.QtCore import *
from PyQt5.QtWidgets import *
from PyQt5.QtGui import *

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle('Generic Window')
        self.resize(600, 400)

        dialogPopup = self.menuBar().addMenu('Dialog')

        modalAction = dialogPopup.addAction('Modal')
        modalAction.triggered.connect(self.onModalTriggered)
        modalAction.setShortcut('Ctrl+d');

    def onModalTriggered(self):
        myModalDialog = MyModalDialog(self)

        result = myModalDialog.exec()
        if result == QDialog.Accepted:
            QMessageBox.information(self, 'Message', 'Adı Soyadı: {}, No: {}'.format(myModalDialog.editName.text(), myModalDialog.editNo.text()))

class MyModalDialog(QDialog):
    def __init__(self, parent):
        super().__init__(parent)
        self.setWindowTitle('My Modal Dialog')
        self.resize(370, 150)

        self.labelName = QLabel('Adı Soyadı', self)
        self.labelName.move(10, 10)

        self.editName = QLineEdit(self)
        self.editName.move(10, 25)
        self.editName.resize(250, 20)

        self.labelNo = QLabel('No', self)
        self.labelNo.move(10, 60)

        self.editNo = QLineEdit(self)
        self.editNo.move(10, 75)
        self.editNo.resize(250, 20)

        self.buttonOk = QPushButton('Ok', self)
        self.buttonOk.move(190, 120)
        self.buttonOk.clicked.connect(self.onButtonOk)

        self.buttonCancel = QPushButton('Cancel', self)
        self.buttonCancel.move(275, 120)
        self.buttonCancel.clicked.connect(self.onButtonCancel)
```

```

self.label = QLabel(self)
self.label.setGeometry(290, 20, 64, 64)
self.label.setPixmap(QPixmap('Icons/person.png'))

def onButtonOk(self):
    self.done(QDialog.Accepted)

def onButtonCancel(self):
    self.done(QDialog.Rejected)

def main():
    app = QApplication(sys.argv)
    mainWindow = MainWindow()
    mainWindow.show()
    app.exec()

main()

```

QtPy'da Modeless Diyalog Pencerelerinin Oluşturulması

Modeless diyalog pencereleri şu aşamalardan geçirilerek oluşturulmaktadır:

- 1) Modal diyalog pencerelerinde olduğu gibi modeless diyalog pencerelerinde de öncelikle QDialog sınıfından bir sınıf türetilir.
- 2) Modeless diyalog pencerelerinin açılması QDialog sınıfının exec metodu ile değil show metodu ile yapılmalıdır.
- 3) Modeless diyalog pencerelerinden çıkış done metoduyla değişim doğrudan close metoduyla yapılmalıdır.

Modal diyalog penceresi örneğimize modeless diyalog penceresi örneğini söyle ekleyebiliriz:

```

import sys
from PyQt5.QtCore import *
from PyQt5.QtWidgets import *
from PyQt5.QtGui import *

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle('Generic Window')
        self.resize(600, 400)

        dialogPopup = self.menuBar().addMenu('Dialog')

        modalAction = dialogPopup.addAction('Modal')
        modalAction.triggered.connect(self.onModalTriggered)
        modalAction.setShortcut('Ctrl+d');

        modelessAction = dialogPopup.addAction('Modeless')
        modelessAction.triggered.connect(self.onModelessTriggered)

    def onModalTriggered(self):
        myModalDialog = MyModalDialog(self)

        result = myModalDialog.exec()
        if result == QDialog.Accepted:
            QMessageBox.information(self, 'Message', 'Adı Soyadı: {}, No: {}'.format(myModalDialog.editName.text(), myModalDialog.editNo.text()))

    def onModelessTriggered(self):
        myModelessDaialog = MyModelessDialog(self)

```

```

myModelessDialog.show()

class MyModalDialog(QDialog):
    def __init__(self, parent):
        super().__init__(parent)
        self.setWindowTitle('My Modal Dialog')
        self.resize(370, 150)

        self.labelName = QLabel('Adı Soyadı', self)
        self.labelName.move(10, 10)

        self.editName = QLineEdit(self)
        self.editName.move(10, 25)
        self.editName.resize(250, 20)

        self.labelNo = QLabel('No', self)
        self.labelNo.move(10, 60)

        self.editNo = QLineEdit(self)
        self.editNo.move(10, 75)
        self.editNo.resize(250, 20)

        self.buttonOk = QPushButton('Ok', self)
        self.buttonOk.move(190, 120)
        self.buttonOk.clicked.connect(self.onButtonOk)

        self.buttonCancel = QPushButton('Cancel', self)
        self.buttonCancel.move(275, 120)
        self.buttonCancel.clicked.connect(self.onButtonCancel)

        self.label = QLabel(self)
        self.label.setGeometry(290, 20, 64, 64)
        self.label.setPixmap(QPixmap('Icons/person.png'))

    def onButtonOk(self):
        self.done(QDialog.Accepted)

    def onButtonCancel(self):
        self.done(QDialog.Rejected)

class MyModelessDialog(QDialog):
    def __init__(self, parent):
        super().__init__(parent)
        self.setWindowTitle('My Modal Dialog')
        self.resize(360, 240)

        self.labelRed = QLabel('Red', self)
        self.labelRed.move(20, 10)

        self.redSlider = QSlider(Qt.Horizontal, self)
        self.redSlider.setMaximum(255)
        self.redSlider.move(20, 30)
        self.redSlider.resize(300, 20)
        self.redSlider.setTickPosition(QSlider.TicksBothSides)
        self.redSlider.setTickInterval(5)
        self.redSlider.valueChanged.connect(self.onValueChanged)

        self.labelGreen = QLabel("Green", self)
        self.labelGreen.move(20, 60)

        self.greenSlider = QSlider(Qt.Horizontal, self)

```

```

self.greenSlider.setMaximum(255)
self.greenSlider.move(20, 80)
self.greenSlider.resize(300, 20)
self.greenSlider.setTickPosition(QSlider.TicksBothSides)
self.greenSlider.setTickInterval(5)
self.greenSlider.valueChanged.connect(self.onValueChanged)

self.labelBlue = QLabel("Blue", self)
self.labelBlue.move(20, 110)

self.blueSlider = QSlider(Qt.Horizontal, self)
self.blueSlider.setMaximum(255)
self.blueSlider.move(20, 130)
self.blueSlider.resize(300, 20)
self.blueSlider.setTickPosition(QSlider.TicksBothSides)
self.blueSlider.setTickInterval(5)
self.blueSlider.valueChanged.connect(self.onValueChanged)

self.buttonQuit = QPushButton('Quit', self)
self.buttonQuit.move(260, 200)
self.buttonQuit.clicked.connect(self.onButtonQuit)

palette = self.parent().palette()
color = palette.color(QPalette.Background)

self.redSlider.setValue(color.red())
self.greenSlider.setValue(color.green())
self.blueSlider.setValue(color.blue())

def onButtonQuit(self):
    self.close()

def onValueChanged(self):
    redValue = self.redSlider.value()
    blueValue = self.blueSlider.value()
    greenValue = self.greenSlider.value()

    palette = QPalette(QColor(redValue, greenValue, blueValue))
    self.parent().setPalette(palette)

def main():
    app = QApplication(sys.argv)
    mainWindow = MainWindow()
    mainWindow.show()
    app.exec()

main()

```

Burada modeless pencere içerisinde üç slider bulunmaktadır. Bu slider'lar hareket ettirildiğinde ana pencerenin zemin rengi değiştirilmektedir.

PyQt'de Bazı Standart Diyalog Pencerelerinin Kullanımı

PyQt'de standart bazı diyalog pencereleri için QDialog sınıfından türetilmiş sınıflar tasarılmıştır.

QFileDialog Penceresinin Kullanımı

QFileDialog sınıfı "dosya açma" ve "dosya saklama" amacıyla dosya seçmek için kullanılan standart bir diyalog penceresidir. Bu diyalog penceresi şöyle kullanılır:

1) QDialog sınıfı türünden bir nesne yaratılır. Bu pencere yaratılırken QDialog sınıfının `__init__` metodunda istenirse pencere başlığında çıkartılacak yazı da belirlenebilir. Örneğin:

```
fileDialog = QDialog(self, 'Dosya Seçiniz')
```

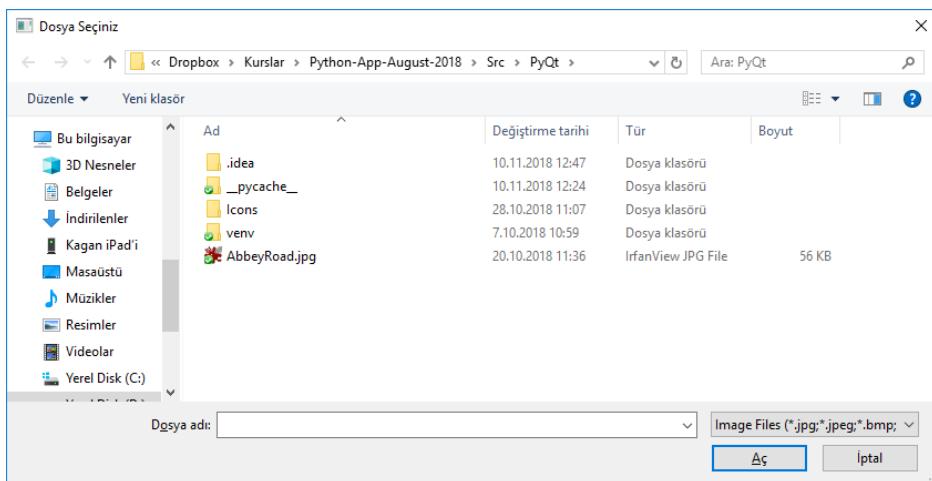
2) Sonra exec fonksiyonu ile diyalop penceresi açılır. exec fonksiyonun geri dönüş değeri yine QDialog.Accepted ya da QDialogRejected biçimindedir.

3) Seçilen dosyanın yol ifadesi QDialog sınıfının selectedFiles isimli metoduyla alınabilir. Bu metot bize dosyaların yol ifadelerinden oluşan str türünden bir liste vermektedir. Bu durumda seçilen dosyanın yol ifadesi şöyle yazdırılabilir:

```
def onOpenTriggered(self, sender):
    fileDialog = QDialog(self, 'Dosya Seçiniz')
    if fileDialog.exec() == QDialog.Accepted:
        QMessageBox.information(self, 'Message', fileDialog.selectedFiles()[0])
```

4) QDialog sınıfının setNameFilters isimli metodu bizden bir string listesi ister. Bu listenin her elemanı "Gösterilecek yazı (filtreleme ifadesi)" biçiminde olmalıdır. Örneğin:

```
def onOpenTriggered(self, sender):
    fileDialog = QDialog(self, 'Dosya Seçiniz')
    fileDialog.setNameFilters(['Text Files (*.txt)', 'Image Files (*.jpg;*.jpeg;*.bmp;*.png)', 'All Files (*.*)'])
    if fileDialog.exec() == QDialog.Accepted:
        QMessageBox.information(self, 'Message', fileDialog.selectedFiles()[0])
```



QFileDialog sınıfının set FileMode isimli metodu ile biz diyalog penceresinde nelerin bulundurulacağını belirleyebiliriz. Örneğin bu metodu QDialog.DirectoryOnly argümanıyla çağrırsak bu durumda bu dialog penceresinden yalnızca biz dizin seçebiliriz.

QFileDialog sınıfının daha pek çok özelliği vardır. Örneğin filtreleme yazısının hangisinin ilk açılısta aktif olacağı, dosya seçeरken dosya zaten varsa (save amaçlı olduğunda) uyarı yazısının çıkip çıkmayacağı, çoklu seçim yapıp yapılmayacağı gibi. Sınıfın ayrıntılı kullanımı için ilgili dokümanlara başvurabilirsiniz.

Aslında QDialog sınıfında getOpenFileName ve getOpenFileNames isimli statik iki metot bulunmaktadır. Bu metot zaten kendi içerisinde QDialog nesnesini yaratıp exec yapmaktadır:

```
QFileDialog.getOpenFileName (QWidget parent = None, QString caption = '',
                           QString directory = '', QString filter = '',
                           QString selectedFilter = '', Options options = 0)
```

```
QFileDialog.getOpenFileNames (QWidget parent = None, QString caption = '',
    QString directory = '', QString filter = '', Options options = 0)
```

getOpenFileName metodu tek dosya seçmek için getOpenFileNames metodu birden fazla dosya seçmek için kullanılmaktadır. Fonksiyonların birinci parametreleri üst pencere nesnesini, ikinci parametreleri pencere başlığını, üçüncü parametreleri başlangıçta diyalog penceresinin hangi dizinde açılacağını, dördüncü parametreleri filtreleme yazısını belirtmektedir. Filtre yazısı tek bir string biçiminde oluşturulmaktadır. Birden fazla seçenek varsa aralarına ";" karakterleri yerleştirilir. getOpenFileName metodu bir demete geri dönmektedir. Demetin ilk elemanı seçilmiş olan dosyanın yol ifadesini, ikinci elemanı ise onun hangi filtreleme yazısı ile seçildiğini belirtir. Örneğin:

```
def onOpenTriggered(self, sender):
    path = QFileDialog.getOpenFileName(self, 'Bir dosya seçiniz', r'c:\windows', 'Text
Files (*.txt);;All Files (*.*)')
    QMessageBox.information(self, 'Message', path[0])
```

Eğer diyalop penceresi cancel tuşıyla kapatılmışsa metodun geri dönüş değerine ilişkin demetin elemanları boş string içermektedir. Bu kontrolün yapılması uygun olur. Örneğin:

```
def onOpenTriggered(self, sender):
    path = QFileDialog.getOpenFileName(self, 'Bir dosya seçiniz', 'c:\windows', 'Text
Files (*.txt);;All Files (*.*)')
    if path[0] != '':
        QMessageBox.information(self, 'Message', path[0])
```

getOpenFileNames isimli metot ise birden fazla dosya seçimine izin vermektedir. Bu metodun geri dönüş değeri yine bir demettir. Demetin birinci elemanı seçilen dosyalara ilişkin bir string listesi ikinci elemanı ise seçimin hangi filtre elemanı ile yapıldığını belirten string nesnesidir.

Ayrıca QDialog sınıfının bir de getSaveFileName isimli static metodu vardır. Bu metot da "save etme" için dosya seçimine izin vermektedir:

```
tuple QFileDialog.getSaveFileNameAndFilter (QWidget parent = None, QString caption = '',
QString directory = '', QString filter = '', QString initialFilter = '', Options options = 0)
```

Metodun kullanımı getOpenFileName metodu ile aynı biçimdedir. Örneğin:

```
def onOpenTriggered(self, sender):
    path = QFileDialog.getSaveFileName(self, 'Bir dosya seçiniz', '.', 'Text Files (*.txt);;All
Files (*.*)')
    if path[0] != '':
        QMessageBox.information(self, 'Message', path[0])
```

QTableWidget Kullanımı

QTableWidget çok sütunlu bir liste kontrolüdür. Belli türden olguların çeşitli özelliklerini görüntülemek için kullanılmaktadır. (Örneğin bir insanın Adı Soyadı, Doğum Yeri, Doğum Tarihi gibi.) QTableWidget şöyle kullanılır:

1) Öncelikle bir QTableWidget nesnesi yaratılır. Örneğin:

```
self.tableWidget = QTableWidget(self)
self.tableWidget.setGeometry(10, 10, 580, 230)
```

2) Daha sonra tablodaki sütun sayısı ve satır sayısı QTableWidgetItem sınıfının setColumnCount ve setRowCount metodlarıyla belirlenir. Örneğin:

```
self.tableWidget = QTableWidget(self)
self.tableWidget.setGeometry(10, 10, 580, 230)
```

```
self.tableWidget.setColumnCount(3)
self.tableWidget.setRowCount(5)
```

3) Artık sıra tepedeki başlık satırının hücrelerini uygun yazılarla set etmeye gelmiştir. Bunun için sınıfın setHorizontalHeaderLabels isimli metodu kullanılır. Bu metot parametre olarak stringler'den oluşan bir liste almaktadır. Örneğin:

```
self.tableWidget = QTableWidget(self)
self.tableWidget.setGeometry(10, 10, 580, 230)
self.tableWidget.setColumnCount(3)
self.tableWidget.setRowCount(5)
self.tableWidget.setHorizontalHeaderLabels(['Adı Soyadı', 'Mesleği', 'Numarası'])
```

	Adı Soyadı	Mesleği	Numarası
1			
2			
3			
4			
5			

4) QTableWidgetItem nesnesindeki her hücre QTableWidgetItem isimli bir sınıfta temsil edilmiştir. Hücreleri set etmek için her bir hücre için QTableWidgetItem nesnesi yaratıp bu nesneyi QTableWidgetItem sınıfının setItem metoduyla set etmemiz gereklidir. setItem bizden set edilecek hücrenin satır ve sütun numarasını ve bir de QTableWidgetItem nesnesini ister. QTableWidgetItem sınıfının setText metodu hücre içerisindeki yazıyı set etmek için, setForeground metodu bu yazının rengini set etmek için, setBackground metodu ise zemin rengini set etmek için kullanılmaktadır. Örneğin:

```
self.tableWidget = QTableWidget(self)
self.tableWidget.setGeometry(10, 10, 580, 230)
self.tableWidget.setColumnCount(3)
self.tableWidget.setRowCount(5)
self.tableWidget.setHorizontalHeaderLabels(['Adı Soyadı', 'Mesleği', 'Numarası'])

item1 = QTableWidgetItem()
item1.setText('Ali Serçe')
self.tableWidget.setItem(0, 0, item1)

item2 = QTableWidgetItem()
item2.setText('Bilgisayar Mühendisi')
self.tableWidget.setItem(0, 1, item2)

item3 = QTableWidgetItem()
item3.setText('1234')
self.tableWidget.setItem(0, 2, item3)
```

Aslında QTableWidgetItem sınıfının str parametrelü başlangıç metodunda hücre görüntülenecek yazı verilebilir. Örneğin:

```
item1 = QTableWidgetItem('Ali Serçe')
self.tableWidget.setItem(0, 0, item1)
```

Tabii uygulamada hücreleri tek tek set etmek yerine bilgileri bir yerden alıp bir döngü içerisinde set etmek daha uygundur. Örneğin:

```
personInfo = [('Ali Serçe', 'Bilgisayar Mühendisi', '1234'), ('Ahmet İnce', 'İşçi', '3567'),
('Sacit Bulut', 'Muhasebeci', '4786')]
for row in range(len(personInfo)):
```

```

item = QTableWidgetItem()
item.setText(personInfo[row][0])
self.tableWidget.setItem(row, 0, item)
item = QTableWidgetItem()
item.setText(personInfo[row][1])
self.tableWidget.setItem(row, 1, item)
item = QTableWidgetItem()
item.setText(personInfo[row][2])
self.tableWidget.setItem(row, 2, item)

```

Tabii bu işlemi iç içe döngüyle de yapabiliyoruz:

```

personInfo = [('Ali Serçe', 'Bilgisayar Mühendisi', '1234'), ('Ahmet İnce', 'İşçi', '3567'),
('Sacit Bulut', 'Muhasebeci', '4786')]
for row in range(len(personInfo)):
    for col in range(len(personInfo[row])):
        item = QTableWidgetItem()
        item.setText(personInfo[row][col])
        self.tableWidget.setItem(row, col, item)

```

Aslında programcı işin başında tüm satırları yaratmak zorunda değildir. QTableWidgetItem sınıfının insertRow isimli metodu yeni bir satır insert etmektedir. Yani yukarıdaki örnek şöyle de olabilir:

```

personInfo = [('Ali Serçe', 'Bilgisayar Mühendisi', '1234'), ('Ahmet İnce', 'İşçi', '3567'),
('Sacit Bulut', 'Muhasebeci', '4786')]
for row in range(len(personInfo)):
    self.tableWidget.insertRow(row)
    for col in range(len(personInfo[row])):
        item = QTableWidgetItem()
        item.setText(personInfo[row][col])
        self.tableWidget.setItem(row, col, item)

```

Aslında QTableWidgetItem nesnesinin başlık kısımları da QTableWidgetItem türünden nesneler almaktadır. Biz yukarıdaki örneklerde başlık kısımlarındaki yazıları tek hamlede setHorizontalHeaderLabels metodu ile aşağıdaki gibi set ettik:

```
self.tableWidget.setHorizontalHeaderLabels(['Adı Soyadı', 'Meslegi', 'Numarası'])
```

Aslında bu işlem arka planda QTableWidgetItem nesneleri yaratılarak yapılmaktadır. Biz de başlıkları böyle set etmek yerine daha ayrıntılı biçimde QTableWidgetItem nesnelerini oluşturarak da set edebiliriz. QTableWidgetItem nesneleri QTableWidgetItem sınıfının setHorizontalHeaderItem metoduyla set edilebilmektedir.

```

header0 = QTableWidgetItem('Adı Soyadı')
header0.setBackground(Qt.red)
header0.setFont(QFont('Arial', 12))
header0.setForeground(Qt.red)
self.tableWidget.setHorizontalHeaderItem(0, header0)

header1 = QTableWidgetItem('Meslegi')
header1.setBackground(Qt.red)
header1.setFont(QFont('Arial', 12))
header1.setForeground(Qt.red)
self.tableWidget.setHorizontalHeaderItem(1, header1)

header2 = QTableWidgetItem('No')
header2.setBackground(Qt.red)
header2.setFont(QFont('Arial', 12))
header2.setForeground(Qt.red)
self.tableWidget.setHorizontalHeaderItem(2, header2)

```

QTableWidget sınıfının pek çok sinyali vardır. Örneğin aktif hücre değiştiğinde currentCellChanged sinyali oluşmaktadır. Bir hücrede kullanıcı değişiklik yaptığında currentItemChanged sinyali tetiklenir. Diğer sinyaller için PyQt dökümanlarından bilgi alabilirsiniz.

QTableWidget sınıfının currentItem isimli metodu bize o anda aktif olan hücreye ilişkin QTableWidgetItem nesnesini vermektedir.

PyQt'de QtDesigner Aracının Kullanılması

GUI elemanlarının kod yoluyla oluşturulması biraz zaman alıcı olmaktadır. Özellikle elemanların uygun konumlara yerleştirilmesi biraz zahmetlidir. İşte bunun için QtDesigner denilen bir araç kullanılmaktadır. QtDesigner QtCreator IDE'sinin bir parçası durumuna getirilmiştir. Ancak bağımsız olarak da yüklenebilmektedir.

QtDesigner isimli araç bağımsız bir program olarak da yüklenebilmektedir. Ancak normal olarak artık bu araç QtCreator denilen IDE'nin bir parçası olarak bulunmaktadır. QtDesigner aşağıda siteden bağımsız bir program olarak indirilebilir:

<https://build-system.fman.io/qt-designer-download>

Ancak eğer aynı zamand abir C/C++ programcısı iseniz QtDesigner aracı için QtCreator IDE'sini yüklemizi tavsiye deriz. Yukarıda da belirtildiği gibi zaten QtCreator IDE'sinin içerisinde QtDesigner hazır olarak bulunmaktadır. Ayrıca QtDesigner Python dünyasından da pip programıyla indirilip kurulabilmektedir.

Python'da PQtDesigner'ın kullanılması oldukça kolaydır.:

1) Önce görsel olarak designer'da tasarım yapılır ve bu tasarım .ui uzantılı bir XML dosyası olarak diskte saklanır.

2) Elde edilen .ui dosyası pyuic5 (python user interface compiler) denilen program tarafından bir python modülüne dönüştürülür. (Bu programın PyQt4'teki ismi pyuic4 biçimindedir.) pyuic5 programı şöyle kullanılmaktadır:

pyuic5 -o <hedef python dosyasının yol ifadesi> <ui dosyasının yol ifadesi>

Örneğin:

pyuic5 -o mainwindow.py mainwindow.ui

Bu işlemin sonucunda biz designer'da oluşturduğumuz görsel öğeleri oluşturan bir python müdü elde etmiş oluruz. Yani başka bir deyişle buradaki mainwindow.py dosyası içerisinde aslında designer'da oluşturduğumuz görsel öğeleri oluşturan Python kodları vardır. Bu modülün içerisinde Ui_XXX biçiminde bir sınıf vardır. Buradaki XXX designer'daki ana formın "object name" property'sinden gelmektedir.

3) Şimdi programcı ui dosyasından elde ettiği Python modülünü kendi programında import etmelidir:

```
import sys
from PyQt5.QtWidgets import *
import mainwindow
```

Bundan sonra programcı bir pencere sınıfı oluşturup (QWidget'tan ya da QMainWindow'dan türetilmeli) __init__ metodunda bu modüldeki görseli kendi ana penceresine gömmelidir. Bu işlem iki satırda yapılır. Önce modüldeki Ui_XXX sınıfı türünden bir nesne yaratılır. Sonra da bu nesne yoluyla bu Ui_XXX sınıfının setupUi isimli метод çağrılır. Bu metot görsel öğeleri parametresiyle aldığı Widget nesnesinin içerisine yerlestirecektir. Bu nedenle parametrede için self argümanının geçilmesi uygun olur. Örneğin:

```
class MainWindow(QWidget):
    def __init__(self):
        super().__init__()
```

```
uiForm = mainwindow.Ui_Form()
uiForm.setupUi(self)
```

Bu durumda designer kullanan tüm program şöyle oluşturulabilir:

```
import sys
from PyQt5.QtWidgets import *
import mainwindow

class MainWindow(QWidget):
    def __init__(self):
        super().__init__()
        uiForm = mainwindow.Ui_Form()
        uiForm.setupUi(self)

def main():
    app = QApplication(sys.argv)
    mainWindow = MainWindow()
    mainWindow.show()
    app.exec()

main()
```

Tabii eğer Ui_XXX nesnesi birtakım metotlardan da kullanılsa bunu da sınıfın veri elemanı yapmak uygun olur. Örneğin:

```
import sys
from PyQt5.QtWidgets import *
import mainwindow

class MainWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.uiForm = mainwindow.Ui_MyWindow()
        self.uiForm.setupUi(self)

def main():
    app = QApplication(sys.argv)
    mainWindow = MainWindow()
    mainWindow.show()
    app.exec()

main()
```

4) Program içerisinde designer'da oluşturduğumuz görsel öğeleri oarada verdigimiz "object name" isimleriyle kullanabiliriz. Başka bir deyişle pyuic5 programı Ui_XXX sınıfının örnek veri elemanı olarak bu görsel öğelere ilişkin nesneleri oluşturmuş durumdadır. Biz de bunu programımızda kullanabiliriz. Örneğin designer'da listWidgetNames ismi verilmiş QListWidget nesnesine şöyle elemanlar ekleyebiliriz. Örneğin:

```
class MainWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.uiForm = mainwindow.Ui_MyWindow()
        self.uiForm.setupUi(self)

        self.uiForm.listWidgetNames.addItem('Ali')
        self.uiForm.listWidgetNames.addItem('Veli')
        self.uiForm.listWidgetNames.addItem('Selami')
        self.uiForm.listWidgetNames.addItem('Ayşe')
        self.uiForm.listWidgetNames.addItem('Fatma')

        self.uiForm.buttonOk.clicked.connect(self.onButtonOk)
```

5) GUI mesajları yine ilgili nesneler üzerinde connect işlemi yapılarak işlenebilir. Örneğin:

```

import sys
from PyQt5.QtWidgets import *
import mainwindow

class MainWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.uiForm = mainwindow.Ui_MyWindow()
        self.uiForm.setupUi(self)

        self.uiForm.listWidgetNames.addItems(['Ali', 'Veli', 'Selami', 'Ayşe', 'Fatma'])
        self.uiForm.buttonOk.clicked.connect(self.onButtonOk)

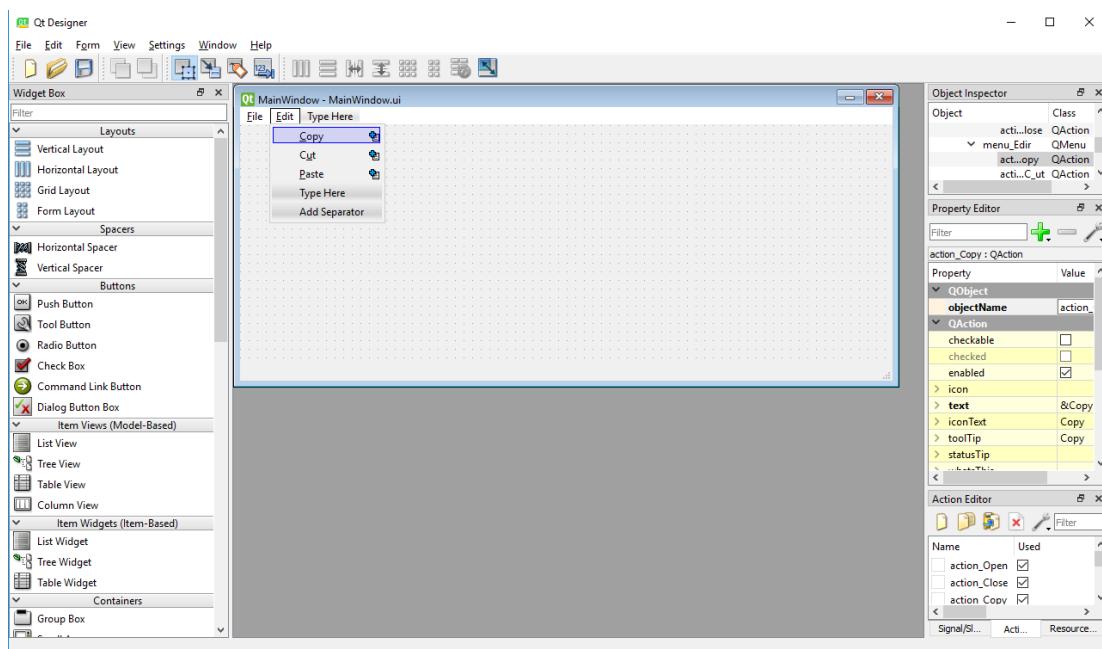
    def onButtonOk(self):
        QMessageBox.information(self, 'Message', 'Checked' if
        self.uiForm.checkBoxEmail.isChecked() else 'Unchecked')

def main():
    app = QApplication(sys.argv)
    mainWindow = MainWindow()
    mainWindow.show()
    app.exec()

main()

```

Örneğin şimdi de menülü bir programı designer'da oluşturalım:



Artık designer'da formu yaratırken pencere cinsi olarak `MainWindow` seçilmelidir. Daha sonra yine save edilen `.ui` dosyası `pyuic5` programı ile işleme sokulur. Sonra da programdan import edilerek kullanılır.

Tabii programımızda başka ana pencereler ya da dialog pencereleri varsa bunlar için de ayrı `.ui` dosyaları oluşturulur. Bunlar da aynı biçimde kullanıma sokulmaktadır. Örneğin menüden bir eleman seçildiğinde bir diyalog penceresinin açılmasını isteyelim. Burada diyalog penceresi yine designer'da oluşturulabilir ve aşağıdaki kodda kullanılabilir:

```

import sys
from PyQt5.QtWidgets import *
import mainwindow
import modaldialog

```

```

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.uiForm = mainwindow.Ui_MainWindow()
        self.uiForm.setupUi(self)
        self.uiForm.modalDialogAction.triggered.connect(self.onModalDialog)

    def onModalDialog(self):
        myDialog = ModalDialog()
        myDialog.exec()

class ModalDialog(QDialog):
    def __init__(self):
        super().__init__()
        self.uiDialog = modaldialog.Ui_Dialog()
        self.uiDialog.setupUi(self)

def main():
    app = QApplication(sys.argv)
    mainWindow = MainWindow()
    mainWindow.show()
    app.exec()

main()

```

Renk Seçme Dialog Penceresinin Kullanımı

Renk seçmek için kullanılan standart diyalog penceresi PyQt'de QColorDialog sınıfıyla temsil edilmiştir. Bu sınıfın kullanımı şöyledir:

1) QColorDialog sınıfı türünden bir nesne yaratılır ve sınıfın exec metodu çağrılır. Yine exec metodunun geri dönüş değeri QDialog.Accepted ya da QDialog.Rejected biçimindedir. Örneğin:

```

def onColorTriggered(self):
    cd = QColorDialog()
    if cd.exec() == QDialog.Accepted:
        pass

```

2) Dialog penceresinden seçilen renk QColor olarak sınıfın selectedColor metoduyla alınabilir. Örneğin seçilen renk ile pencerenin zemini boyamak isteyelim:

```

import sys
from PyQt5.QtWidgets import *
from PyQt5.QtGui import *

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.resize(600, 350)
        dialogPopup = self.menuBar().addMenu('Dialog')

        modalAction = dialogPopup.addAction('Color...')
        modalAction.triggered.connect(self.onColorTriggered)
        modalAction.setShortcut('Ctrl+c');

    def onColorTriggered(self):
        cd = QColorDialog()
        if cd.exec() == QDialog.Accepted:
            palette = QPalette()

```

```

palette.setColor(QPalette.Window, cd.selectedColor())
self.setPalette(palette)

def main():
    app = QApplication(sys.argv)
    mainWindow = MainWindow()
    mainWindow.show()

    app.exec()

main()

```

3) Sınıfın setCurrentColor isimli metodu şe biz diyalog penceresi açıldığında görüntülenecek default rengi de belirleyebiliriz.

PyQt'de Çizim İşlemleri

İşletim sistemlerindeki pek çok GUI alt sisteminde pencere içerisindeki görüntüler bu alt sistem tarafından otomatik olarak tutulup geri basılmamaktadır. Bu sistemlerde pencere içerisindeki görüntünün oluşturulması programcıya bırakılmıştır. Programcı pencere içeresine bir çizim yaptıgında o pencerenin üzerine bir pencere getirilip tekrar açıldığında o çizim kaybolur. İşte işletim sisteminin GUI alt sistemi bunu tutarak geri basmamaktadır. Bunun yerine pencere içerisindeki görüntünün bozulduğunu anlatan bir mesajı uygulamanın (thread'in) mesaj kuyruğuna bırakır. Bu mesaj pencere içerisindeki görüntünün bozulduğu ve yeniden çizilmesi gerektiği anlamına gelmektedir. Dolayısıyla bu sistemlerde çizimler bu mesaj geldiğinde yapılmalıdır. (Bazı işletim sistemlerinin GUI alt sistemleri pencere içerisindeki çizimleri kendisi tutup basabilmektedir. Ancak Qt'nin "cross platform" özelliği ortak bir bölene göre tasarılmıştır. Qt'de pencere içerisindeki çizim bozulduğunda framework tarafından QWidget sınıfının paintEvent isimli fonksiyonu çağrılır. O halde çizimler bu fonksiyon içerisinde yapılmalıdır. Örneğin:

```

import sys
from PyQt5.QtWidgets import *
from PyQt5.QtGui import *

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.resize(600, 400)

    def paintEvent(self, event):
        pass

def main():
    app = QApplication(sys.argv)
    mainWindow = MainWindow()
    mainWindow.show()
    app.exec()

main()

```

Çizimin paintEvent isimli metotta yapılması gerektiğini belirttiğimizde Pekiyi çizim nasıl yapılacaktır? İşte çizim yapmak için QPainter isimli bir sınıf kullanılır. QPainter sınıfı türünden bir nesne hangi pencere içeresine çizim yapılacağını anlatan bir QWidget parametresiyle yaratılır. Örneğin:

```

def paintEvent(self, event):
    painter = QPainter(self)

```

İste çizim metodları aslında QPainter sınıfının örnek metodları biçiminde bulunmaktadır. Örneğin:

```

def paintEvent(self, event):
    painter = QPainter(self)
    painter.drawEllipse(10, 10, 100, 100)

```

Kalem (Pen) ve Fırça (Brush) Nesneleri

Çizimlerin şekilleri kalem (pen) ile, iç boyamaları ise fırça (brush) ile yapılmaktadır. QPanter nesnesi yaratıldığında defauşlt bir kalem ve fırça da nesneye ilişirilmiş durumdadır. Ancak daha sonra biz farklı kalemler ve fırçalar yaratarak painter nesnesinin onu kullanmasını sağlayabiliriz.

Kalemler QPen isimli sınıfı temsil edilmiştir. Bir QPen nesnesi sınıfın aşağıdaki `__init__` metoduyla yaratılabilir:

```
__init__(self, Qt.PenStyle)
__init__(self, QBrush brush, float width, Qt.PenStyle style = Qt.SolidLine, Qt.PenCapStyle cap
= Qt.SquareCap, Qt.PenJoinStyle join = Qt.BevelJoin)
```

Kelman tipik üç özelliği stili, rengi ve kalınlığıdır. Eğer `__init__` tek argümanla çağrılırsa buradaki değer kalemin stilini belirtir. Kalem stilleri şunlardan oluşmaktadır:

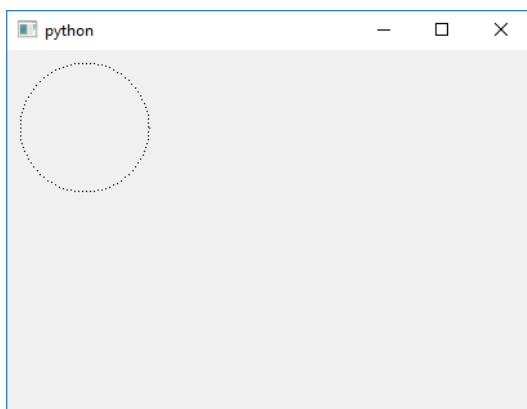


Bir kalem yaratıldıktan sonra QPainter nesnesinin o kalemi kullanabilmesi için set işleminin yapılması gereklidir. Örneğin:

```
def paintEvent(self, event):
    painter = QPainter(self)

    pen = QPen(Qt.DotLine)
    painter.setPen(pen)

    painter.drawEllipse(10, 10, 100, 100)
```

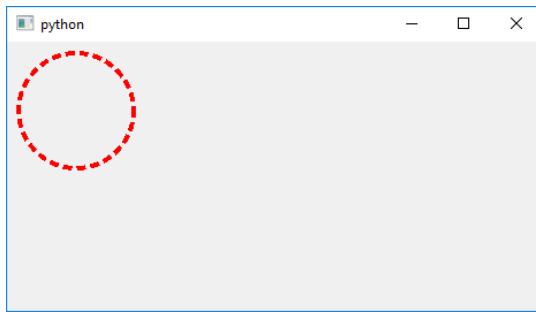


Bir kalemi yaratırken renk, kalınlık ve stil birlikte verilebilir. Renk QBrush nesnesi biçiminde yanı bir fırça gibi verilmelidir. (Kalem yeteri kadar kalın olduğunda fırça etkisi yaratmaktadır.) Örneğin:

```
def paintEvent(self, event):
    painter = QPainter(self)

    pen = QPen(QBrush(Qt.red), 4, Qt.DotLine)
    painter.setPen(pen)
```

```
painter.drawEllipse(10, 10, 100, 100)
```

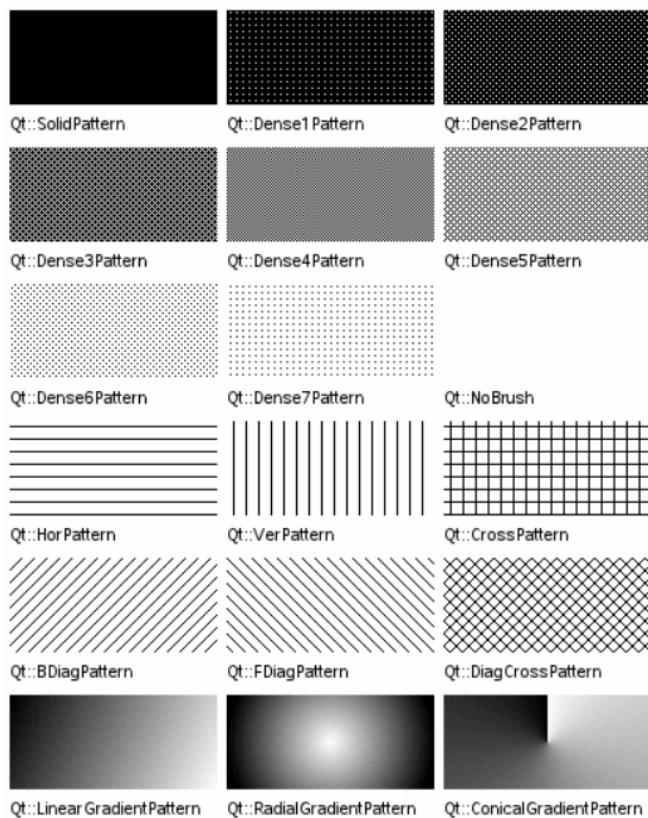


Aslında setPen metodu yalnızca renk verilerek de çağrılabılır. Bu durumda verilen renkte, kalınlıkta, düz çizgili bir kalem kullanılır.

Fırça nesneleri QBrush sınıfıyla temsil edilmiştir. Bir QBrush nesnesi sınıfın aşağıdaki `__init__` metotlarıyla yaratılabilir:

```
__init__(self, Qt.BrushStyle bs)
__init__(self, QColor color, Qt.BrushStyle style = Qt.SolidPattern)
__init__(self, QColor color, QPixmap pixmap)
__init__(self, QPixmap pixmap)
__init__(self, QImage image)
__init__(self, QGradient gradient)
__init__(self, QBrush brush)
__init__(self, QVariant variant)
```

Bu metodlar fırçanın biçimini, rengini ve fırçayı oluşturan resmi bizden argüna olarak almaktadır. Fırça biçimleri aşağıda verilmiştir:



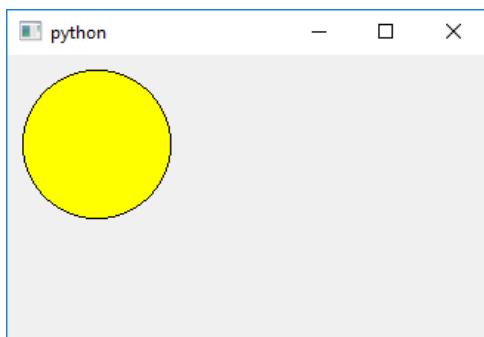
QPainter Sınıfının Çizim Yapan Önemli Metotları

Temel olarak QPainter sınıfında drawXXX ve fillXXX biçiminde (yani ismi draw ve fill ile başlayan) bir grup çizim metodu vardır. drawXX metotları hem çizimin dışını kalem nesneyle yaparlar hem de içini fırça nesneyle boyarlar, halbuki fillXXX metotları ise yalnızca kapılı şekillerin içinin boyanması için kullanılmaktadır. QPainter nesnesi yaratıldığında default kalem 1 kalınlıklı, siyah, düz çizgili kalemdir. Default fırça ise transparan (yani boş) bir fırçadır. Burada bu metodların önemlileri ele alınacaktır.

- drawEllipse metodu elips çizmek için kullanılır. drawEllipse metodu temel olarak bizden bir dikdörtgen ister ve o dikdörtgenin iç teğet elipsini çizer. Dikdörtgen kareye yaklaşıkca elips de çembere yaklaşır. drawEllipse elipsin içini de set edilen fırçayla boyamaktadır. Örneğin:

```
def paintEvent(self, event):
    painter = QPainter(self)

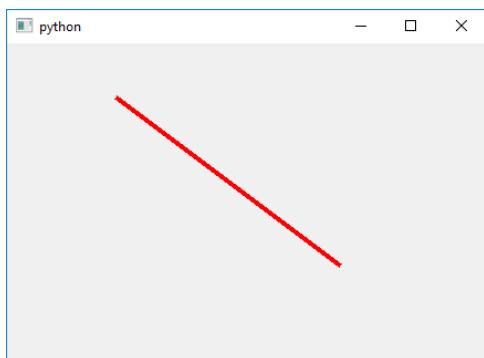
    painter.setBrush(QBrush(Qt.yellow))
    painter.drawEllipse(10, 10, 100, 100)
```



- drawLine metodu iki nokta belirtilerek bir çizgi çizmek için kullanılır. Örneğin:

```
def paintEvent(self, event):
    painter = QPainter(self)

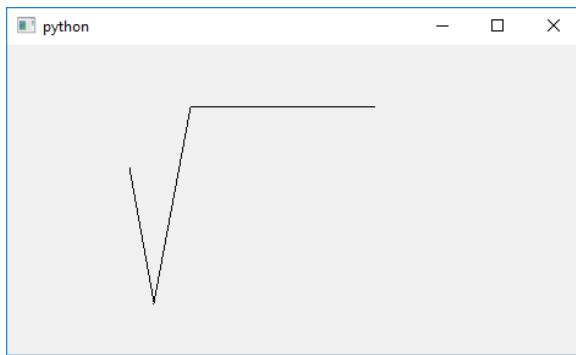
    painter.setPen(QPen(QBrush(Qt.red), 4, Qt.SolidLine))
    painter.drawLine(100, 50, 300, 200);
```



- drawPolyLine metodu bir grup noktası alıp bu noktaları birleştirerek çizgiler çizer. Metot bizden birden fazla QPoint nesnesi istemektedir. Örneğin:

```
def paintEvent(self, event):
    painter = QPainter(self)

    painter.drawPolyline(QPoint(100, 100), QPoint(120, 210), QPoint(150, 50), QPoint(300, 50))
```



Aslında metodun parametresi '*'lidir. Yani bu durumda bizim girdiğimiz argümanlar bir demet biçimine dönüştürülerek aktarılır. Tabii biz istersek argümanı '*'layarak aynı etkiyi oluşturabiliriz. Örneğin:

```
def paintEvent(self, event):
    painter = QPainter(self)

    points = [QPoint(100, 100), QPoint(120, 210), QPoint(150, 50), QPoint(300, 50)]
    painter.drawPolyline(*points)
```

Örneğin bir sinüs eğrisini şöyle çizebiliriz:

```
def paintEvent(self, event):
    painter = QPainter(self)

    painter.setPen(QPen(QBrush(Qt.red), 4, Qt.SolidLine))
    xorg, yorg = 350, 150

    points = []

    for x in [i * 0.1 for i in range(-60, 60)]:
        points.append(QPoint(xorg + round(x * 50), yorg - round(math.sin(x) * 50)))

    painter.drawPolyline(*points)
```



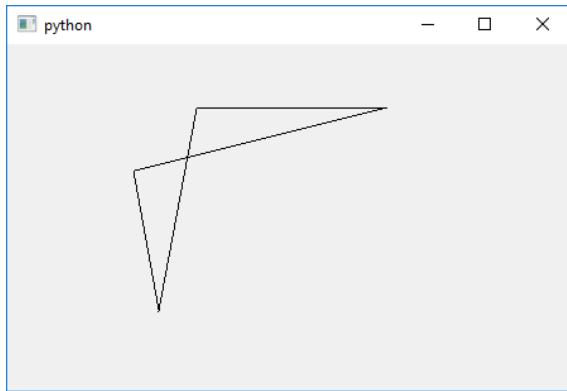
Çizimde birkaç noktaya dikkat ediniz:

- 1) range fonksiyonu gerçek sayısal artırım yapamamaktadır. Bu nedenle bunun yerine "list comprehension" kullanılmıştır.
- 2) Kartzyen sistemde y ekseni yukarıya doğru artar. Ancak ekran koordinat sisteminde aşağıya doğru artmaktadır.
- 3) sinüs fonksiyonundan elde edilen değerin pixel'e dönüştürülmesi için değer 50 ile çarpılmıştır. Başka bir deyişle gerçek kartzyen sistmedeki 1 ekran koordinat sisteminde 50 pixel'e karşılık gelmektedir.
- 4) Çizimin orijin noktasının 350, 150 olduğunu dikkat ediniz.

- drawPolygone metodu tıpkı drawPolyLine gibidir. Ancak son noktaya ilk noktayı birleştirir. Örneğin:

```
def paintEvent(self, event):
    painter = QPainter(self)

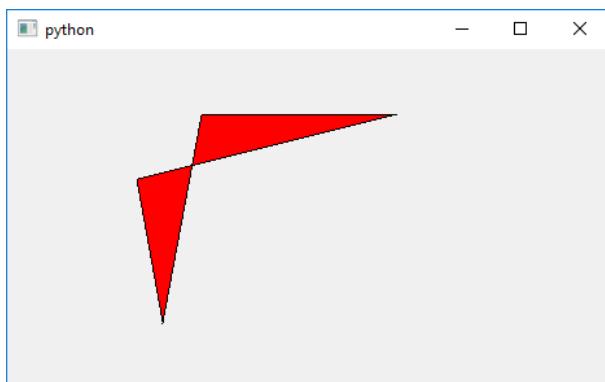
    points = [QPoint(100, 100), QPoint(120, 210), QPoint(150, 50), QPoint(300, 50)]
    painter.drawPolygon(*points)
```



Tabii biz istersek QPainter nesnesine bir fırça da set ederek çizimin içini boyanmasını sağlayabiliriz. Örneğin:

```
def paintEvent(self, event):
    painter = QPainter(self)

    painter.setBrush(QBrush(Qt.red))
    points = [QPoint(100, 100), QPoint(120, 210), QPoint(150, 50), QPoint(300, 50)]
    painter.drawPolygon(*points)
```



- drawRect dikdörtgen çizmek için kullanılmaktadır. Örneğin:

```
def paintEvent(self, event):
    painter = QPainter(self)

    painter.setPen(QPen(QBrush(Qt.red), 4, Qt.SolidLine))
    painter.setBrush(QBrush(Qt.yellow))

    painter.drawRect(100, 100, 200, 200)
```

- drawPixmap metodu bir resmi çizmek için kullanılmaktadır. QPixmap sınıfı resmi temsil eder. drawPixmap metodu da resmi çizer. Örneğin:

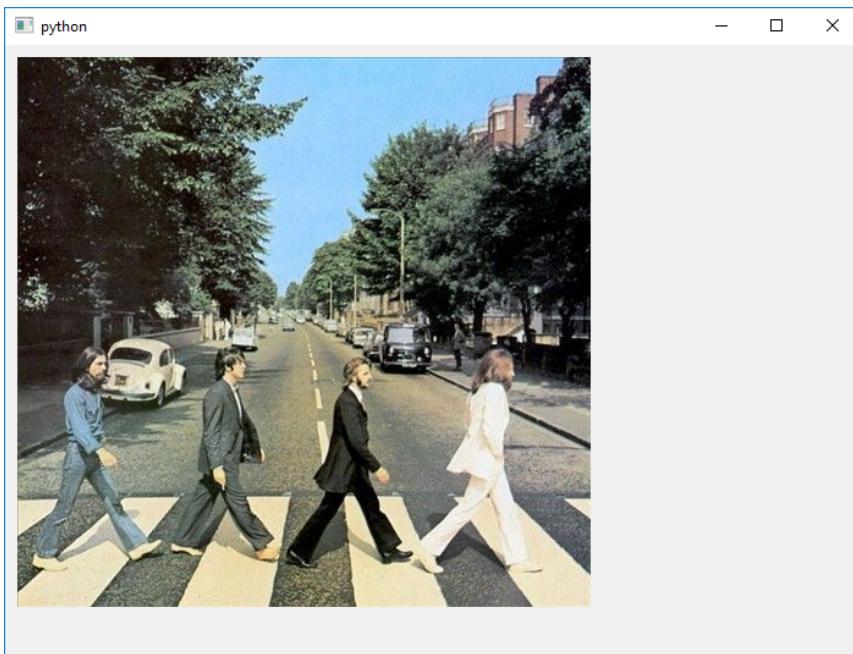
```

def paintEvent(self, event):
    painter = QPainter(self)

    pixmap = QPixmap("AbbeyRoad.jpg")
    painter.drawPixmap(10, 10, pixmap.width(), pixmap.height(), pixmap)

```

`drawPixmap` metodu bizden çizilecek yerin sol üst köşesini ve yapılacak çizimin genişlik ve yükseliğini tabii bir de resmin kendisini istemektedir. Bu metot orijinal resmi bizim belirttiğimiz genişlik ve yüksekliğe ölçeklendirerek çizer. Yukarıdaki örnekte resim orijinal büyülüğünde çizilmiştir.



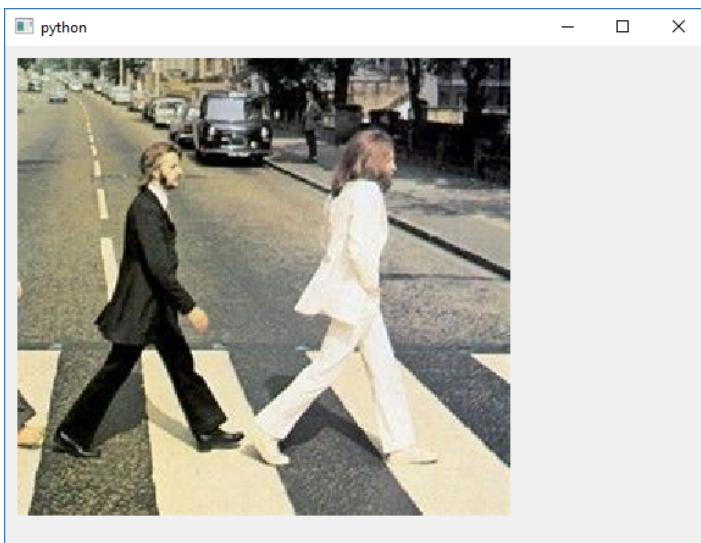
B`drawPixmap` metodu ile istenirse resmin belli bir kısmı ölçeklendirilerek de çizilebilir. Örneğin:

```

def paintEvent(self, event):
    painter = QPainter(self)

    pixmap = QPixmap("AbbeyRoad.jpg")
    painter.drawPixmap(10, 10, 600, 600, pixmap, 200, 200, 400, 400)

```



Fare Hareketlerinin ve Eylemlerinin İzlenmesi

Pencere üzerinde fare tıklandığında ya da pencere üzerinde fare hareket ettirildiğinde PyQt Ortamı bizim çeşitli metotlarımızı çağırarak bize bildirimde bulunmaktadır. Böylece programcı fare ile ilgili işlemler yapabilmektedir. Bu metotlar şunlardır:

```
mousePressEvent  
mouseReleaseEvent  
mouseMoveEvent  
mouseDoubleClickEvent
```

Metotların QMouseEvent isimli sınıfından bir parametreleri vardır. Bu parametre bize ilgili fare olayı hakkında detayları vermektedir. (Örneğin farenin hangi tuşuna basık durumdadır, O anda fare hangi konumdaır gibi). Biz PyQt'de yukarıdaki metotları pencere sınıfının içerisinde yazarak bu olayları yakalayabiliriz. Çünkü PyQt fare ilgili hareketler yapıldığında ilgili sınıfın yukarıdaki metotlarını çağırmaktadır. İlgili pencere içerisinde farenin herhangi bir tuşuna tıklandığında mousePressEvent metodu, el fareden çekildiğinde ise mouseRelease metodu çağrılmaktadır. Farenin herhangi bir tuşuna basılıp fare hareket ettirildiğinde bir dizi mouseMoveEvent çağrımları yapılır. Ancak bu çağrımların her pixel için yapılacak garanti değildir. Farenin sürüklentimes sırasında hangi yoğunlukta mouseMoveEvent metodunun çağrılabileceği sistemin genel durumuna bağlı olmaktadır. Nihayet pencere içerisinde fare ile çift tıklandığında mouseDoubleClickEvent metodu çağrılmaktadır. Biz bu metotların parametrelerinden farenin o andaki konum bilgisini ve farenin hangi tuşa basılmış olduğu bilgisini elde edebiliriz. Örneğin:

```
import sys  
from PyQt5.QtWidgets import *  
from PyQt5.QtGui import *  
  
class MainWindow(QMainWindow):  
    def __init__(self):  
        super().__init__()  
        self.resize(700, 500)  
  
    def mousePressEvent(self, mouseEvent):  
        print('mousePressEvent', mouseEvent.x(), mouseEvent.y())  
  
    def mouseReleaseEvent(self, mouseEvent):  
        print('mouseReleaseEvent', mouseEvent.x(), mouseEvent.y())  
  
    def mouseMoveEvent(self, mouseEvent):  
        print('mouseMoveEvent', mouseEvent.x(), mouseEvent.y())  
  
def main():  
    app = QApplication(sys.argv)  
    mainWindow = MainWindow()  
    mainWindow.show()  
    app.exec()  
  
main()
```

Yazboz Tahtası (Scratchpad) Uygulaması

Bazen fare mesajlarında çizimler yapmak isteyebiliriz. Çizimleri fare mesajlarında yaptığımda bunların paintEvent metodunda saklanıp yeniden yapılması gerekir. İşte genellikle bunun yerine tüm çizimlerin paintEvent metodunda yapılması ancak fare mesajlarında yalnızca çizilecek bilgilerin güncellenmesi yoluna gidilmektedir. paintEvent metodunu biz manuel olarak kendimiz çağrırmamalıyız. (Neden bizim bu paintEvent metodunu çağrırmamamız gereğinin bazı ayrıntıları vardır. Ancak burada ele alınmayacağından)

Yazboz tahtası uygulamasında biz elimizi fareye tıklayıp kaldırana kadarki çizimleri QPolygon isimli PyQt'nin liste benzeri bir sınıfında saklayacağız. Sonra bu QPolygon nesnelerini de ayrı bir listede biriktireceğiz. Farenin her bir hareketinde paintEvent çizimlerinin yeniden yapılması için update metodunu uygulayacağız. Aşağıda böyle bir yazboz tahtası uygulaması verilmiştir. Uygulamada painter nesnesine "antialiasing" özelliği verildiğine dikkat ediniz. "Antialiasing" çizim sırasında komşu piksellerin uygun renklerle boyanmasını sağlayan algoritmik teknique verilen bir

isimdir. Böylece çözümünürlüğü gerçek kartezyen sisteme göre çok daha düşük olan ekran koordinat sisteminde çizimlerin "kırıklı" görülmesinin bir derece üstesinden gelinebilmektedir.

```
import sys
from PyQt5.QtCore import *
from PyQt5.QtWidgets import *
from PyQt5.QtGui import *

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.resize(700, 500)
        self.line = None
        self.lines = []
        self.drawFlag = False

    def mousePressEvent(self, mouseEvent):
        if mouseEvent.button() == Qt.LeftButton:
            self.line = QPolygon()
            self.drawFlag = True

    def mouseReleaseEvent(self, mouseEvent):
        if self.drawFlag:
            self.lines.append(self.line)
            self.drawFlag = False

    def mouseMoveEvent(self, mouseEvent):
        if self.drawFlag:
            self.line.append(mouseEvent.pos())
            self.update()

    def paintEvent(self, *args, **kwargs):
        painter = QPainter(self)
        painter.setRenderHints(QPainter.Antialiasing | QPainter.SmoothPixmapTransform)
        painter.setPen(QPen(QBrush(Qt.red), 4, Qt.SolidLine))
        for polygon in self.lines:
            painter.drawPolyline(polygon)
        if self.line:
            painter.drawPolyline(self.line)

def main():
    app = QApplication(sys.argv)
    mainWindow = MainWindow()
    mainWindow.show()
    app.exec()

main()
```

Proseslerarası Haberleşme (Interprocess Communication - IPC)

Bir prosesten diğerine belli miktarda byte gönderip alma sürecine "Proseslerarası Haberleşme" denilmektedir. Örneğin biz bir Python programından diğerine bilgi gönderip diğerinin bu bilgiyi kullanmasını sağlayabiliriz. Proseslerarası haberleşme kabaca ikiye ayrılır:



Proseslerarası haberleşme yöntemlerine pek çok durumda gereksinim duyulmaktadır. Örneğin bir proses dış dünyadan elde ettiği verileri başka proseslere iletmek isteyebilir. Ya da bir proses başka bir proses tarafından yönetilmek istenebilir. Aynı makinenin prosesleri arasındaki haberleşmelerde kullanılan tipik teknikler şunlardır:

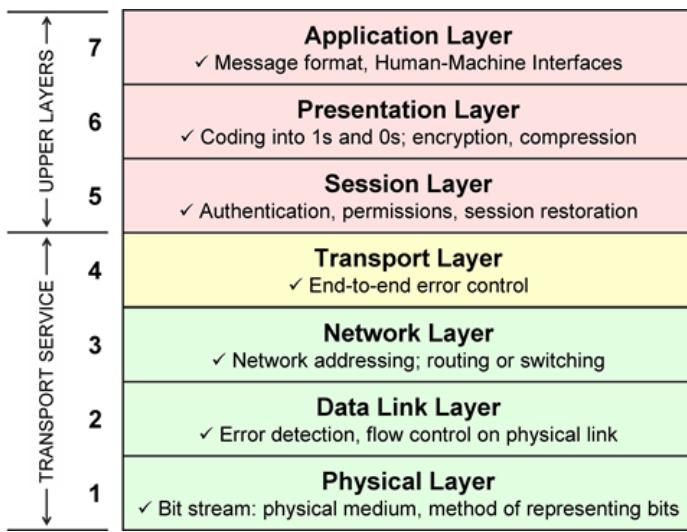
- Borular (Pipes)
- Paylaşılan Bellek Alanları (Shared Memory)
- Mesaj Kuyrukları (Message Queue)

Ancak Python kursumuzda bu yöntemleri ele almayacağız.

Farklı makineler birbirlerine ağ içerisinde bağlanmış olabilir. Biz bir makinede çalışan bir programın ağa bağlı başka bir makinedeki prosese bilgi göndermesini ve almasını isteyebiliriz. Böyle bir haberleşmede artık işletim sisteminin dışında başka birtakım aktörler de devreye girecektir. Örneğin kablolama sisteminden kullanılan hub'a kadar bazı donanım birimleri işin içine karışmaktadır. Üstelik bu tür haberleşmelerde işletim sistemleri bile birbirlerinden farklı olabilmektedir. İşte hetorejen böyle ortamlarda haberleşmenin sağılıklı yürütülmesi için önceden belirlenmiş birtakım kuralların bulunması gereklidir. (Örneğin kablo standartları ve konnektörler nelerdir? Network kartının özellikleri nasıl olacaktır? Bilgiler nasıl paketlere ayrılp gönderilecektir? Makinalar nasıl birbirlerinden ayrılacaktır vs. gibi...) İşte tüm bu belirlemelere protokol denilmektedir. Bugün en yaygın kullanılan protokol Internet'in de kullandığı IP (Internet Protocol) protokol ailesidir. Kursumuzda IP protokol ailesi ile haberleşme ele alınacaktır.

Tıpkı fonksiyonların birbirlerini çağırarak daha yüksek seviyeli işlemleri yapar hale gelmesi gibi protokoller de üst üste yiğilarak ayrı ayrı oluşturulmaktadır. Her üst protokol aşağıdaki zaten hazır olduğu fikriyle yalnızca kendi gereksinimlerini tanımlamaktadır. Böyle katmanlı tasarımın pek çok faydası vardır. Örneğin bu sayede üst seviye protokoller detay barındırmazlar ve aşağı düzeydeki protokollerin değişmesinden fazlaca etkilenmezler. İşte farklı maknelerin haberleşmesi için bu biçimde oluşturulmuş pek çok protokol ailesi vardır. Örneğin AppleTalk, NETBIOS vs. gibi...

Network altında bilgisayar haberleşmesi için protokol katmanlarının nasıl oluşturulması gerektiğine yönelik IEEE ismine OSI (Open System Interconnection) denilen bir doküman yayımlanmıştır. Buna OSI model denilmektedir. OSI model bir protokol ailesi değildir. Protokol ailesi oluşturacaklar için bir kılavuz niteliğindedir. OSI'nin toplam 7 katmanı vardır:



OSI'nin en aşağı katmanına "Fiziksel Katman (Physical Layer)" deilmektedir. Fiziksel katmanda iletişimini yapılacak ortam tanımlanmaktadır. Örneğin kullanılacak kablolar, konnektörler, gerilim seviyeleri gibi. Bunun üzerinde "Veri Bağlantı Katmanı (Data Link Layer)" bulunmaktadır. Bu katmanda network kartlarına ilişkin belirlemeler, fiziksel adresleme belirlemeleri vs. bulunmaktadır. Örneğin Ethernet kartlarının protokolü olan Ethernet Protokolü bir Veri Bağlantı Katmanı Protokolüdür. Network katmanı (Network Layer) mantıksal adreslemenin tanımlandığı, bilginin nasıl paketlere ayrılp gönderileceğinin tanımlandığı en önemlî katmanlardan biridir. Örneğin IP protokol ailesinin IP Protokolü (Internet Protocol) OSI'ye göre Network katmanına ilişkindir. Network katmanında ayrıca "internetworking" için rotalama belirlemeleri de bulunmaktadır. Network üzerinde "İleti Katmanı (Transport Layer)" bulunmaktadır. Burada paketlerin numaralandırılması, mantıksal port adreslerinin tanımlanması, hata durumunda bunun telafi edilmesi gibi belirlemeler bulundurulmaktadır. Örneğin IP protokol ailesindeki TCP ve UDP protokoller ileti Katmanına ilişkin protokollerdir. "Oteturum Katmanı (Session Layer)" pek çok ailede bulunmamaktadır. Burada haberleşme için gereken oturum açmaya yönelik belirlemeler bulunur. Örneğin izinler, kimlik doğulama gibi. Bunun yukarısında da "Sunum Katmanı (Presentation Layer)" bulunur. Sunum katmanında gönderilip alnına bilgilerin sıkıştırılmasına, açılmasına, şifrelenmesine vs. yönelik belirlemeler bulunmaktadır. IP protokol ailesi Sunum Katmanına da sahip değildir. Nihayet en tepede "Uygulama Katmanı (Application Layer)" bulunmaktadır. Bu katman artık belli bir amacı gerçekleştirmek için oluşturulan yazılımların kullanacağı belirlemeleri içerir. Örneğin eposta için kullanılan POP3, dosya transferi için kullanılan FTP birer Uygulama Katmanı Protokolüdür.

Internetin Kısa Tarihi

Bilgisayarları birbirlerine bağlamak ilk kez 60'lı yıllarda insanların aklına gelmiştir. Soğuk savaş yıllarında Amerika Savunma Bakanlığına bağlı olan DARPA (Defense Advanced Research Project Agency) kurumu birkaç üniversite ile birlikte 1969 yılında ARPANET isimli bir proje başlattı. ARPANET ilk kez 1969 yılında uzak mesafeden dört üniversitenin birbirlerine bağlanmasıyla hayatı geçirilmiş oldu. ARPANET'te daha sonra bazı devlet kurumları ve üniversiteler katılmaya başlamıştır. 70'li yılların sonlarına doğru ARPANET Amerika'da gelişmeye başlamıştır. 1983 yılında ARPANET NCP (Network Control Protocol) protokolünü bırakarak IP ailesine ailesine geçmiştir. Ve arık ağ Internet ismiyle yayılmaya devam etmiştir. Internet 80 yıllarda Avrupa'ya ve Türkiye'ye de geldi. Ancak tabi kişisel bilgisayarlar dahaenyidi ve Internet'e ancak Üniversitelerden ve bazı devlet kurumlarından, özel sektörden bağlanılıbiliyordu. 1990-91 yıllarında HTTP protokü tasarlandı ve ilk Web sayfaları oluşturulmaya başlandı. 90'lı yılların ortalarına doğru tüm dünyada kişisel bilgisayarlarla servis sağlayıcılar sayesinde Internet'e girmek mümkün hale gelmiştir. Daha sonraları modern modem/router'lara yüksek hızlı evden erişimler sağlanmıştır.

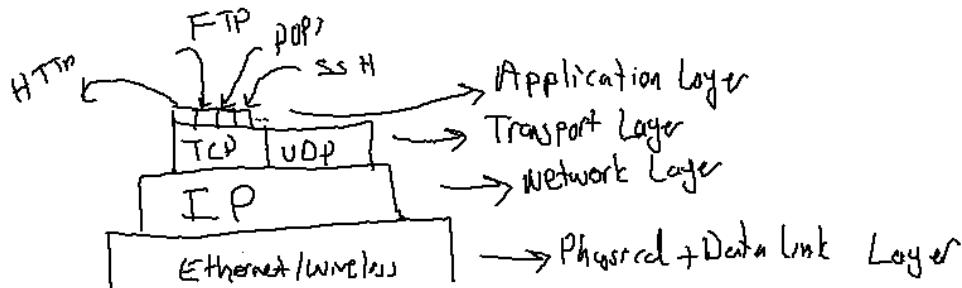
Internet ismi "internetworking" sözcüğünden gelmektedir. Internetworking "yerel ağların birbirlerine router isimli cihazlarla bağlanmasıyla oluşturulmaktadır. Internetworking temel bir terimdir ve IP protokol ailesinin ismi buradan gelmektedir. Bugün Internet denildiğinde herkesin bağlandığı ARPANET'ten evrimleşen dev ağ aklıma gelir. (Internet yazarken I'yı büyük yazarsak bu ağ anlaşılır.) Şüphesiz mevcut protokoller sayesinde herkes kendi internetini kurabilir. Örneğin biz de birkaç arkadaşımızla ayrı bir Internet dünyası oluşturabiliz. Hatta bazı ülkelerin bu biçimde kendilerine özgü Internet'leri vardır.

IP Protokol Ailesi

IP açık bir protokol ailesidir. Burada açık demekle hiçbir şirketin malının olmadığı bağımsız konsorsiyumlar tarafından yönetildiği anlamına gelmektedir. Ayrıca dokümanlar herkes tarafından paylaşılınmakta ve isteyen kişiler önerilerde bulunabilmektedir.

IP protokolü Vint Cerf ve Bob Kahn tarafından 1974 yılında önce TCP sonra IP biçiminde tasarlanmıştır. Sonra aileye diğer üyeleri katılmıştır. İlk ciddi gerçekleştirmi BSD sistemlerinde yapılmıştır. 1983 yılında ARPANET'in IP ailesine geçmesiyle popüleritesi çok artmıştır.

IP protokol ailesinin temel protokollerini dört katmandan oluşmaktadır.

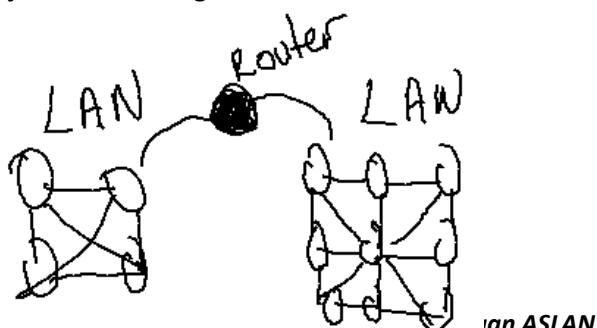


IP protokol ailesi aslında geniş bir ailedir. Ailede pek çok yardımcı protokol vardır. Yukarıdaki şekil yalnızca kursumuzda söz konusu edilen konuları kapsayacak biçimde oluşturulmuştur.

Ailenin en önemli taban protokolü IP (Internetworking Protocol) protokolüdür. Zaten aileye ismini bu protokol vermiştir. IP protokolü paket anahtarlamalı (packet switching) bir protokoldür. Yani bilgiler paket denilen öbeklere ayrılarak gönderilip alınır. IP protokolünde adresleme artık fiziksel değil mantıksaldır. IP protokol ailesinde ağa bağlı her birime "host" denilmektedir. IP protokolünde her host'un ismine IP adresi denilen mantıksal bir adresi vardır. Mantıksal adres bunun donanımsal olarak belirlenmediği yazılımsal olarak atandığı anlamına gelmektedir. Fakat örneğin Ethernet protokolünün kullandığı MAC adresi fiziksel bir adresdir. Fiziksel adres bunun donanımsal olarak kartın üzerine çakılı olduğu ya da donanımın kendisinin bunu tespit edip işlem yaptığı adres demektir. Dolayısıyla mantıksal adresler dinamiktir, fiziksel adresler statiktir. Mantıksal adresler biz ağa dahil olduğumuzda bize atanmaktadır. Tabi biz de istediğimiz adresin atanması konusunda ısrarcı olabiliriz.

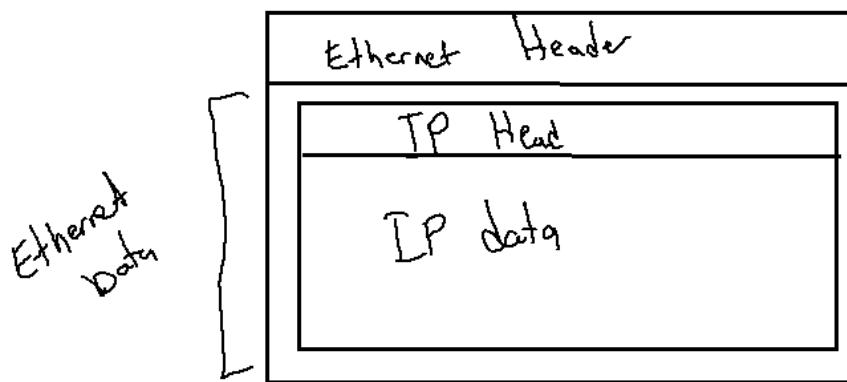
IP protokolünün de versiyonları vardır. Şu anda hala ağırlıklı kullanılan versiyon IPV4'tür. Ancak IPV6 yavaş yavaş daha yaygın kullanılır hale gelmiştir. IPV4'te IP adresleri 4 byte uzunluktadır. Ancak IPV6'da IP adresleri 16 byte'tır. 4 byte'lık IP adresleri şu an için artık çok yetersiz kalmaktadır.

Bugün bilgisayarlarımıza fiziksel ve data link katmanı olarak Ethernet ve Wireless Protokollerini kullanılmaktadır. Ethernet protokolü ethernet kartına gereksinim duyar. Bu kart fizikselli olarak bilgileri bilgisayarımızdan dışarı gönderip almakta kullanılır. Ethernet protokolü de paket anahtarlamalı bir protokoldür. Yani bilgiler paket paket gönderilip alınır. Paket anahtarlama hattın etkin kullanımını sağlar. Biz Ethernet kartlarını bir hub'la biribirine bağlayarak yerel bir ağ (local area network) oluşturabiliriz. Bugün evlerimizdeki ağ da yerel bir ağdır. Yerel ağları birbirlerine bağlamak için "router" denilen aygıtlar kullanılır. Ethernet kartı (yani network kartı) aynı ağdaki bir bilgisayardan diğerine paket haberleşmesi için kullanılmaktadır. Ancak router farklı ağlar arasında paket haberleşmesi için kullanılır. Bugün evlerimizdeki ADSL modemler aynı zamanda birer router görevindedir.



Bizim evimizdeki yerel ağ Internet isimli dev ağa router aracılığıyla tek bir host gibi bağlanmaktadır. Dolayısıyla bizim Internet için dışarıdan kullanılacak tek bir IP adresimiz vardır (Tabi tek bir router ve hattımızın bulunduğu varsayıyoruz). Bizim evimizdeki yerel ağ ayrı bir IP ağıdır. Yani ayrı bir dünyadır. Biz istersek hiç Internet'te çıkmadan kendi yerel ağımızda tüm Internet uygulamalarını (Yani IP protokol uygulamalarını) çalıştırabiliriz. Buna genellikle "Intranet" denilmektedir. O halde bizim evimizdeki bir bilgisayarın bir yerel IP adresi vardır. Router dış dünyadan gelen paketleri yerel ağda uygun bilgisayara dağıtmaktadır. Yerel ağdaki paketleri de dış dünyaya ilişkinse dış dünyaya yollamaktadır. Biz yerel ağımızdaki bir host'tan diğerine bilgi gönderirken router devreye girmez.

Ip protokolünde gönerilen bir paketin başında "IP header" isimli bir başlık kısmı vardır. Burada pakete ilişkin metadata bilgileri bulunur. Örneğin paket hangi IP adresine gönderilmektedir? Checksum bilgisi nedir? Hangi IP versiyonu kullanılmaktadır? vs. Aslında tabi (böylek olmak zorunda değil ama) bilgiler neticede ethernet kartı ile gönderilip alındığı için IP paketi aslında Ethernet protokolünün ethernet paketinin data bölümünde kodlanır. Ethernet protokolünün de ayrı bir header bölümü vardır. Örneğin:



Ethernet protokolü IEEE 802.3 numaralı standarıyla belirlenmiştir. Wireless protokolü de aynı ailedendir. O da IEEE 802.11 numaralı standarttır.

Ip protokü ile birden fazla paketten oluşan bilgi gönderilebilir mi? Evet fakat bunun için paketlere numara vererek bizim de adeta ayrı bir protokol oluşturuyoruz gereklidir. Zaten TCP protokolü buna benzer bir protokoldür.

TCP protokolü güvenli (reliable) bir protokoldür. Burada güvenlik demek verisin yolda bozulmasının teleafi edilmesi ve paketlerin düzgün aktarılması anlamına gelir. Çünkü TCP'de bir akış kontrolü (flow control) vardır. Gönderen tarafla alan taraf karşılıklı konuşarak hatalı giden paketlerin tefafisini sağlayabilmektedir. TCP stream tabanlı bir protokoldür. Stream tabanlı demekle byte byte okumaya kaldığı yerden devam edebilmek anlaşıılır. TCP ile biz daha büyük bilgileri gönderip alabiliriz. TCP bu durumda bu bilgiyi IP paketlerine böler. Onlara numara verir ve onların karşı tarafa güvenli ulaşmasını denetler. Karşı taraf gelen bilgiyi sanki borudan okuma yapıyormuş gibi byte byte elde edebilir.

UDP (User Datagram Protocol) güvenli olmayan paket tabanlı (datagram) bir haberleşme sunar. Yani UDP'de bilgiler IP'deki gibi bağımsız paketler halinde gönderilip alınır. UDP'de bir paket ya alınır ya alınmaz. Byte byte okuma mümkün değildir. Paketin alındığına dair bir geri bildirim yapılmaz. Tabi bu özelliğinden dolayı UDP daha hızlıdır. UDP özellikle periyodik data gönderimlerinde, televizyon yayını gibi işlemlerde tercih edilmektedir.

TCP bağlantılı (connection oriented) bir protokoldür, UDP bağlantısızdır (connectionless). Bağlantılı protokol demek iki taraf haberleşmeden önce birbirlerine bağlanıp karşılıklı konuşma için birbirlerini tanımları demektir. TCP tipik olarak client-server tarzda bir çalışmayı akla getirmektedir. Client-Server haberleşmede bir taraf client bir taraf server olur. Client taraf server tarafa bağlanır, haberleşme bundan sonra yapılır.

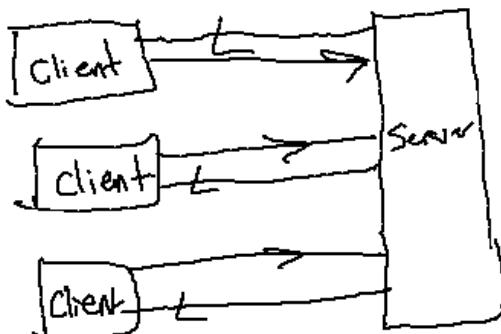
TCP	UDP
Bağlantılı	Bağlantısız
Stream Tabanlı	Datagram Tabanlı
Güvenilir	Güvenilir Değil
Yavaş	Hızlı

TCP ve UDP'de işin içine port numarası kavramı da girmektedir. Port numarası aynı host'taki uygulamaları birbirlerinden ayırmak için düşünülmüştür. Adeta şirketlerdeki içsel (internal) telefon numaralarına benzetilebilir. TCP ve UDP protokollerinde bilgi göndermek için yalnızca gönderilecek host'un IP'sinin bilinmesi yeterli değildir. Aynı zamanda oradaki uygulamanın hangi port ile ilgiliğinin de bilinmesi gereklidir. Genellikle gösterimde ip adresi ve port numarası aralarına ':' karakteri getirilerek "ip:port" biçiminde belirtilmektedir. IPV4'te toplam 65536 port numarası vardır (yani port numarası için iki byte yer ayrılr). IPV6'da ise port numaraları 4 byte uzunluğundadır. IPV4'te ilk 1024 port numarası Internet'in kendi uygulama protokelleri için ayrılmıştır. Bunlara "well known" portlar da denilmektedir. Örneğin FTP 21, SSH 22, Telnet 23, HTTP 80 numaralı portları kullanmaktadır. Biz kendi uygulamalarımız için port numarası belirleyeceksek ilk 1024 portu kullanmamalıyız.

IP protokol ailesinde her host'un bir IP adresinin olduğunu belirtmiştir. IP adreslerinin akılda tutulması zordur. Bu nedenle IP ailesinde IP adresleri host isimleriyle eşleştirilmiştir. Protokol host isimleriyle değil IP adresleriyle çalışır. Ancak IP ailesinde host isimlerinin hangi IP adreslerine karşılık geldiği ismine "DNS" denilen özel server'larda tutulmaktadır. Bu serverlardan sorgulama "DNS Protokolü" ile yapılmaktadır. Host isimleri IP adresleri bire bir eşlenmiş değildir. Bir host ismine birden fazla IP adresi karşılık gelebileceği gibi, bir IP adresine de birden fazla host ismi karşılık gelebilmektedir.

Client-Server Çalışma Modeli

Yukarıda da belirtildiği gibi TCP tipik olarak client-server bir çalışmayı akla getirmektedir. Client-Server modelde ismine client ve server denilen iki ayrı program vardır. Asıl işi server program yapar. Client yalnızca istekte bulunur. Server işi yapar sonuçları client'a gönderir. Bir server birden fazla client'a hizmet verebilmektedir. >



Client-Server modelde önce client server'a bağlanır. Haberleşme ondan sonra başlar. Client-Server uygulamalar her ne kadar TCP'yi çağrıstırıysa da aslında bu bir haberleşme mimarisidir. Yani aslında client-server çalışma için IP ailesinin kullanılması gerekmey. Bu çalışma örneğin aynı makinadaki prosesler arasında borularla mesaj kuyruklarıyla da sağlanabilir.

Client-Server çalışmanın şu avantajları vardır:

- 1) Server programın çalıştığı makine güclü olabilir. Biz de onun gücünden yararlanmak istiyor olabiliriz. Örneğin uzun zaman alan bir işlemi el terminalinden yapmak yerine el terminalini client olarak kullanıp asıl işi server'a yaptırmak uygun olabilir.

2) Server program kaynak paylaşımı sağlayabilir. Örneğin yazıcı telk bir bilgisayara bağlıdır. Başka bilgisayardaki print programları client gibi çalışarak yazıcının bağlı makinadaki server programa isteği iletir. Server da print işlemini client için yapar. Ya da örneğin server'a bir veritabanı bağlıdır. Client ondan istekte bulunur. Örneğin banka ATM'lerinde veritabanı ATM makinasının içerisinde değildir. ATM'deki program client program gibi davranışmaktadır.

3) Server program client'lar arasında işbirliği sağlayabilir. Onlar arasındaki iletişime aracılık edebilir. Örneğin bir char programında client'lar birbirini tanıtmamaktadır. Herkes yalnızca server'i tanır. Her client server'a bağlanır. Server client arasında haberleşmeye aracılık eder. Ağ üzerinde çalışan oyun programları bu biçimde bir server'in işbirliği ile gerçekleştirilmektedir.

4) Client-Server çalışma dağıtık uygulamalarda da karşımıza çıkabilmektedir. Yani bir işin belirli parçalarını başka bilgisayarlarda yapıp sonra onu birleştirmek isteyebiliriz.

Yerel IP Adresleri ve Internet IP Adresleri

Internet sözcüğü "internetworking" sözcüğünden kısaltılarak oluşturulmuştur. "Internetworking" ise "ağların birbirlerine bağlanması" anlamına gelmektedir. Bu sistemde yerel ağlar ismine "router" denilen aygıtlarla birbirlerine bağlanmaktadır. Böylece iki IP adres alanı oluşturmaktadır. Yerel ağın içerisindeki host'ların IP adreslerine "yerel IP adresleri" denilmektedir. IPv4'te IP adresleri sınıflara ayrılmıştır. (Burada bu sınıflardan bahsetmeyeceğiz) Yerel ağdaki adresler genellikle "192.168" biçiminde başlarlar. Biz yerel ağda iki host arasında IP haberleşmesi yaparken bu yerel IP adreslerini kullanırız. Ancak yerel ağımız dış dünyaya (yani tipik olarak The Internet'e) router'ın IP adresiyle bağlıdır. Başka bir deyişle dış dünya bizim yerel ağımızı tek bir host gibi görmektedir. İşte dış dünyadan gelen bilgiler aslında router'a gelmektedir. Router'da bunu yerel ağda uygun host'a iletmektedir. Şimdi biz yerel ağdaki bir host'tan dış dünyadaki bir host'a bağlanmak isteyelim. Bu durumda router bu bağlantı isteğinin hangi yerel host tarafından yapılmak istediğini not alır. Gelen IP paketlerini yerel ağdaki o host'a yönlendirir. Yan yerel ağdaki client programın dış dünyaya bağlanmasında bir sorun yoktur. Pekiyi biz yerel ağdaki bir host'ta bir server program çalıştırduğumda dış dünyadaki bir client bu server'a nasıl bağlanacaktır? İşte bu durumda bizim kendi router'ımızda port yönlendirmesi yapmamız gerekmektedir. Çünkü dış dünyada bize bağlanmak isteyen host bizim yerel IP'mizle değil Internet IP'mizle (yani router'ın IP'si ile) bağlantı yapacaktır. Port yönlendirmesi ile biz router'ımıza "falanca port'tan gelen bağlantı isteklerini yerel ağdaki şu host'a yönlendir" demiş oluruz.

Socket Kavramı

Farklı bilgisayarlar arasında proseslerarası haberleşmede kullanılcaka protokollerin işletim sistemleri tarafından destekleniyor olması gerekmektedir. Bugün işletim sistemleri bazı yaygın protokollerini destekler durumdadır. Windows gibi, MAC OS X gibi, Linux gibi işletim sistemleri IP protokol ailesini uzun süredir desteklemektedir. İşletim sistemlerinde bir protokol ailesi kullanılarak uygulama programlarının yazılabilmesi için bir kütüphanenin de bulunuyor olması gereklidir. İşte bu kütüphaneye "socket kütüphanesi" denilmektedir. Windows, MAC OS X ve Linux gibi sistemler soket kütüphanesini biribirine çok benzer biçimde desteklemektedir. Bu soket kütüphanesi aslında C Programlama Dilinden kullanım için oluşturulmuştur. Ancak pek çok bu C kütüphanelerini kendine özgü biçimde kullanarak benzer kütüphaneleri oluşturmuştur. Python da aslında arka planda işletim sistemlerinin sunduğu C'de yazılmış olan soket kütüphanesini kullanmaktadır.

Soket kütüphanesi yalnızca IP ailesi için oluşturulmuş bir kütüphane değildir. Soket fonksiyonları pek çok protokol ailesinin ortak fonksiyonlarıdır. Yani biz IP ailesinde de çalışıksak, Apple Talk ailesinde de çalışıksak yine aynı soket fonksiyonlarını kullanırız.

Python'da soket kütüphanesi standart kütüphanedeki "socket" modülüyle gerçekleştirilmiştir. Dolayısıyla soket arayüzüni kullanabilmek için bizim "socket" modülünü import etmemiz gereklidir.

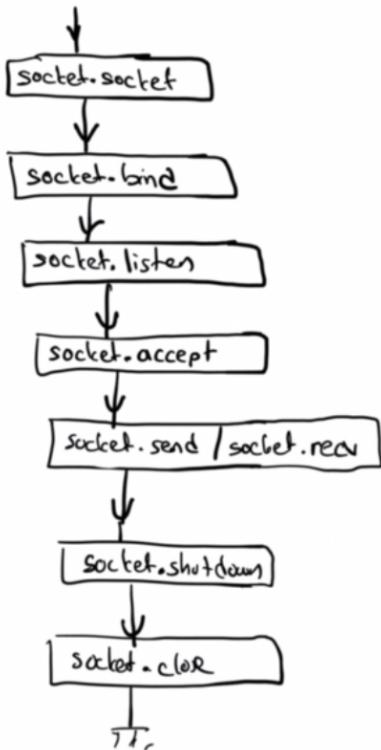
Python'da TCP/IP Uygulamaları

TCP/IP uygulamları için bir tane server programın bir tane de client programın yazılması gerekmektedir. Bu modelde client program server'ın IP adresini ve port numarasını belirterek server programa bağlanır. Sonra karşılıklı veri alış verisi yapılır. Biz de burada önce server sonra da client programın yazımını göreceğiz.

Python'da soket işlemlerinde fonksiyonlar ve metodlar birtakım hatalar karşısında socket.error isimli bir türle raise işlemi yapmaktadır. Programcının try-except bloklarıyla bu tür hataları ele alması uygun olur.

TCP/IP Server Programlarının Yazımı

Bir server program sırasıyla şu adımlardan geçirilerek oluşturulmaktadır:



1) Server programın öncelikle bir soket yaratması gerekmektedir. Bu işlem soket isimli fonksiyonla yapılır. Fonksiyonun parametrik yapısı şöyledir:

```
socket.socket(family=AF_INET, type=SOCK_STREAM, proto=0, fileno=None)
```

Fonsiyonun family parametresi kullanılacak protokol ailesini belirtmektedir. Bu parametre default biçimde socket.AF_INET değerini almıştır. AF_INET "IPV4" protokolü anlamına gelmektedir. IPV6 için AF_INET6 kullanılmalıdır. type parametresi soketin stream tabanlı mı, yoksa datagram tabanlı mı olduğunu belirtir. TCP uygulamaları için bu parametresinin sock.SOCK_STREAM biçiminde girilmesi gerekmektedir. proto parametresi spesifik protokolü belirtir. Ancak IP protokol ailesinde zaten ikinci parametre socket.SOCK_STREAM geçildiğinde bu spesifik transport protokolü TCP, sock.SOCK_DGRAM geçildiğinde UDP anlaşılmaktadır. fileno parametresi UNIX türevi sistemlerde anlamlıdır. Bu durumda bir TCP soket yaratma şöyle yapılabilir:

```
import socket  
  
serverSock = socket.socket(family=socket.AF_INET, type=socket.SOCK_STREAM)
```

Tabii aslında verilen arımanlar parametre değişkenlerinin default değerleridir. Bu durumda aynı işlem şöyle de yapılabilir:

```
import socket  
  
serverSock = socket.socket()
```

socket.socket fonksiyonundan elde edilen ürün socket.socket isimli bir sınıf türündendir. Artık diğer işlemler bu sınıfın metodlarıyla yapılacaktır.

2) Soket yaratıldıktan sonra onun bağlanması (bind edilmesi) gerekmektedir. Bağlama işlemi socket.socket sınıfının bind isimli metoduyle yapılır. bind işlemi sırasında "hangi network kartından gelen bağlantı taleplerinin" ve "hangi port için" gelen bağlantı taleplerinin dikkate alınacağı belirtilir. Yani bizim bind işleminde "şu network kartından ve port için gelen bağlantı isteklerini kabul et" belirmesini yapmış oluruz. socket.bind fonksiyonunun parametrik yapısı şöyledir:

```
bind(address)
```

bind fonksiyonu (host, port numarası) biçiminde bir demeti parametre olarak almaktadır. Buradaki host IP adresini temsil etmektedir. IP adresleri bir yazı biçiminde noktalı formda verilebilir. Örneğin:

```
'192.168.1.100'  
'78.180.123.119'
```

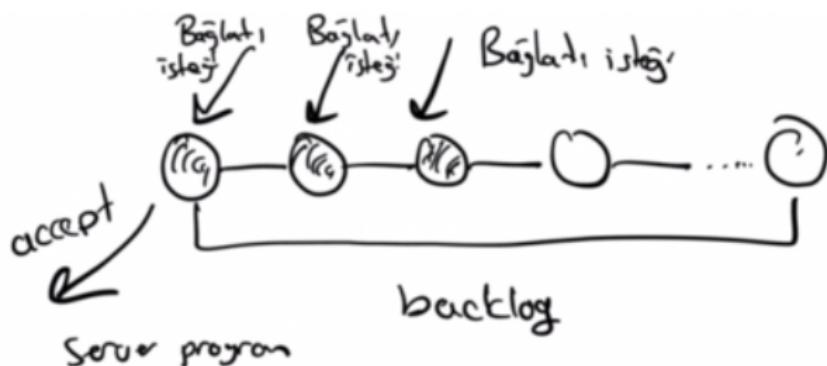
gibi. Biz host olarak doğrudan host'un adını da girebiliriz. Bu durumda DNS işlemi zaten ilgili fonksiyon tarafında yapılmaktadır. Server bind işlemini yaparken belli bir network kartının IP adresini verebilir. Bu durumda yalnızca o karttan gelen bağlantı istekleri kabul edilecektir. Ya da server tüm kartlardan gelen bağlantı isteklerini kabul edebilir. Bunun için ip adresi boş string geçilebilir ya da '127.0.0.1' biçiminde loopback adres geçilebilir. Örneğin:

```
import socket  
  
PORT = 5050  
  
try:  
    serverSock = socket.socket(family=socket.AF_INET, type=socket.SOCK_STREAM)  
    serverSock.bind(('', PORT))  
except socket.error as msg:  
    print('Socket error:{}' .format(msg))  
else:  
    print('ok')
```

3) Artık sıra listen işlemine gelmiştir. socket.socket sınıfının listen metodu soketi aktif dinleme konumuna sokar. listen metodu blokeye yol açmamaktadır. Ancak bu metottan sonra işletim sistemi artık gelen bağlantı isteklerini bunu bekleyen programımıza iletecektir. listen metodunun parametrik yapısı şöyledir:

```
listen([backlog])
```

Metodun parametresi dinleme kuruğunu uzunluğunu almaktadır. İşletim sistemi gelen bağlantı isteklerini sokete ilişkin bir dinleme kuyruğuna yerleştirir. Eğer bu kuyruk tamamen dolarsa yeni bağlantı istekleri reddedilecektir. Tabii server her bağlantıyı kabul ettiğinde kuyrukta yer açılır.



Örneğin:

```
import socket  
  
PORT = 5050
```

```

try:
    serverSock = socket.socket(family=socket.AF_INET, type=socket.SOCK_STREAM)
    serverSock.bind(('', PORT))
    serverSock.listen(8)
except socket.error as msg:
    print('Socket error:{}' .format(msg))
else:
    print('ok')

```

4) Artık server program gelen bağlantı isteklerini socket.socket sınıfının accept metoduyla kabul edecktir. Tabii bunun için client'ın bağlantı talebinde bulunuyor olması gereklidir. Default blokeli modda server accept metodunda bir bağlantı olulsana kadar blokede bekler. Tabii server eğer accept uygulamadan önce kurulmuş bir bağlantı isteği varsa accept hiç bloke olmadan doğrudan bağlantıyı sağlar. accept metodu parametresizdir. accept metodunun geri dönüş değeri iki elemanlı bir demet nesnesidir. Demetin ilk elemanı client ile konuşmakta kullanılacak soketi, ikinci elemanı ise bağlanılan client'a ilişkin adresi belirtir. Server listen ve accept işlemi için bir soket kullanmaktadır. Bu örneklerimizdeki serverSocket isimli sokettir. Ancak accept başarılı olduğunda bu metot bize yeni bir soket geri döndürür. Artık server o client'la o soketi kullanarak konuşur. server accept metodunu birden fazla kez çağrırsa birden fazla client ile bağlantı kurar. Her bağlantı kurduğu client ile o accept'in geri dönüş değeri ile verilen soketle konuşur. Örneğin:

```

import socket

PORT = 5050

try:
    serverSock = socket.socket(family=socket.AF_INET, type=socket.SOCK_STREAM)
    serverSock.bind(('', PORT))
    serverSock.listen(8)

    print('Waiting for connection...')
    clientSock, clientAddr = serverSock.accept()
except socket.error as msg:
    print('Socket error:{}' .format(msg))
else:
    print('ok')

```

5) Artık client ile bağlantı sağlandıktan sonra sıra okuma ve yazmaya gelmiştir. Bunun için socket.socket sınıfının send ve recv metodları kullanılır. Ancak bu metodlar client program anlatıldıkten sonra ayrı bir başlıkta ele alınacaktır.

6) İşimiz bittikten sonra soketi shutdown edip kapatırız. Soketi kapatmak için close metodu kullanılmaktadır. Örneğin:

```

import socket

PORT = 5050

try:
    serverSock = socket.socket(family=socket.AF_INET, type=socket.SOCK_STREAM)
    serverSock.bind(('', PORT))
    serverSock.listen(8)

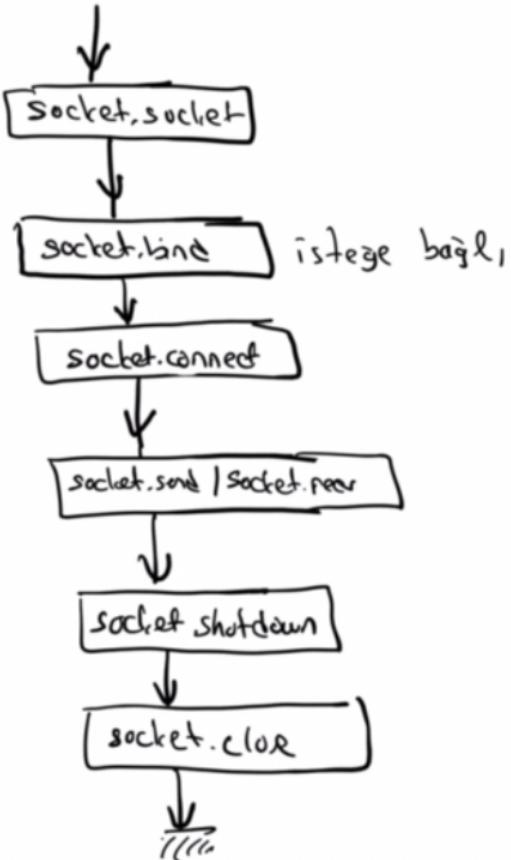
    print('Waiting for connection...')
    clientSock, clientAddr = serverSock.accept()
    .....
    clientSock.close()
    serverSock.close()
except socket.error as msg:
    print('Socket error:{}' .format(msg))
else:

```

```
print('Ok')
```

TCP IP Client Programın Yazımı

TCP client program tipik olarak şu aşamadan geçilerek yazılmaktadır:



1) Client program da önce socket modülünün soket fonksiyonuyla aynı biçimde soket nesnesini yaratır. Örneğin:

```
import socket
PORT = 5050

try:
    clientSock = socket.socket(family=socket.AF_INET, type=socket.SOCK_STREAM)
except socket.error as msg:
    print('Socket error:{}' .format(msg))
else:
    print('Ok')
```

2) Client program belli bir kaynak porttan bağlanmak isteyebilir. Bunun client programın da bind işlemi yapması gereklidir. Ancak client program bind yapmazsa işletim sistemi tarafından ona boş bir kaynak port atanmaktadır.



Bağlantı oluştduğunda atarafların bir kaynak port numarası ve kaynak IP adresi, bir de hedef port numarası ve hedef IP adresi vardır. Bağlantı client tarafın hedef IP adresini ve hedef port numarasını belirterek kurulur. Ancak bu işlemi client yaparken kendi IP adresi ve kendi kaynak port numarası vardır. İşte biz bazen server'a belli bir kaynak port numarası ile bağlanmak isteyebiliriz. (Bazı server'lar bizim belli bir kaynak portu kullanmamızı isteyebilmektedir.) Bu işlem client'in bind işlemi yapmasıyla sağlanır. Ancak client bind işlemi yapmazsa otomatik olarak ona bir kaynak port numarası atanmaktadır.

3) Artık client.socket sınıfının connect metoduyla server'a bağlanır. client.connect metodunu yine bir demet içerisinde (host, port no) biçiminde argüman oluşturarak çağırır. Tabii hedef makinenin host ismi ya da IP adresi verilebilmektedir. Ancak port numarasının server'in dinlediği port numarasıyla aynı olması gereklidir. Yukarıda da belirtildiği gibi genel olarak TCP protokolünde server'a farklı bir kaynak porttan bağlanılabilir. connect metodu çağrıldığında belli bir zaman aşımı (timeout) miktarı kadar client bekler. Eğer server bu süre zarfında accept işlemi yapmazsa bağlantı başarısız olur. Örneğin:

```
import socket

PORT = 5050

try:
    clientSock = socket.socket(family=socket.AF_INET, type=socket.SOCK_STREAM)
    clientSock.connect(('127.0.0.1', PORT))
    print('connected')
    clientSock.close()
except socket.error as msg:
    print('Socket error:{}' .format(msg))
```

Bu durumda bağlantı uygulayan en yalın server ve client programlar şöyle yazılabılırler:

```
#server.py

import socket

PORT = 5050

try:
    serverSock = socket.socket()
    serverSock.bind(('', PORT))
    serverSock.listen(8)

    print('wating for connection...')
    clientSock, clientAddr = serverSock.accept()
    print('connected:{}' .format(clientAddr))
    clientSock.close()
    serverSock.close()
except socket.error as msg:
    print('Socket error:{}' .format(msg))
```

```
#client.py

import socket

PORT = 5050

try:
    clientSock = socket.socket()
    clientSock.connect(('127.0.0.1', PORT))
    print('connected')
```

```

    clientSock.close()
except socket.error as msg:
    print('Socket error:{}' .format(msg))

```

Bağlantı kurulduktan sonra artık iki taraf karşılık biçimde (full duplex) birbirlerine bilgi gönderip alabilirler. Bunun için socket.socket sınıfının send ve recv metotları kullanılmaktadır. send metodunun parametrik yapısı şöyledir:

```
socket.send(bytes[, flags])
```

Metodun birinci parametresi gönderilecek bytes türünden nesneyi ikinci parametresi gönderim bayraklarını belirtmektedir. İkinci parametre girilmeyebilir. send metodу tüm bilgi network tamponuna bırakılana kadar blokede kalmaktadır. Bilgi network tamponuna bırakıldıktan sonra işletim sistemi tarafından TCP paketi haline dönüştürülp gönderilmektedir. Ancak send metodу geri döndüğün de bilgi henüz yerel bilgisayardan dışarı çıkmamış olabilir. send metodу gönderilmek istenen byte sayısı ile geri dönmektedir. recv metodunun parametrik yapısı da şöyledir:

```
socket.recv(bufsize[, flags])
```

Metodun birinci parametresi kaç byte okunacağı bilgisidir. İkinci parametre okuma bayrağını belirtir. Bu bayrak girilmeyebilir. Metot geri dönüş değeri olarak bytes nesnesi verir. recv metodу blokeli modda (yani default durumda) henüz sokete hiçbir bilgi gelmemişse blokede bekler. Ancak sokette en az 1 byte bilgi varsa parametresiyle belirtilen miktarda byte'in okunması için beklemez. Okuyabildiği kadar byte'i okur, okuyabildiği byte sayısı ile geri döner. Eğer recv sırasında karşı taraf soketi close ile kapatmışsa bu durum recv boş bir byte dizisi verir. Boş bir byte dizisinin mantıksal olarak False biçimde değerlendirildiğini anımsayınız. recv sırasında karşı taraf soketi kapatmadan bağlantı koparsa bu durum exception oluşturmaktadır.

Bağlantıdan sonra server'in client'a yazı gönderip client'in bu yazıyı ekrana bastığı bir soket uygulaması şöyledir olabilir:

```

#server.py

import socket

PORT = 5050

try:
    with socket.socket() as serverSock:
        serverSock.bind(('', PORT))
        serverSock.listen(8)

        print('Waiting for connection...')
        clientSock, clientAddr = serverSock.accept()
        print('Connected:{}' .format(clientAddr))
        while True:
            text = input('Bir yazı giriniz:')
            b = text.encode('UTF-8')
            clientSock.send(b)
            if text == 'quit':
                break
    clientSock.close()

except socket.error as msg:
    print('Socket error:{}' .format(msg))

```

```
#client.py
```

```

import socket

PORT = 5050

try:

```

```

with socket.socket() as clientSock:
    clientSock.connect(('127.0.0.1', PORT))
    print('connected')
    while True:
        b = clientSock.recv(1000)
        if not b:
            break
        s = b.decode('UTF-8')
        if s == 'quit':
            break
        print(s)
except socket.error as msg:
    print('Socket error:{}' .format(msg))

```

Yukarıda da belirtildiği gibi client program bind işlemi yaparak kendi kaynak port numarasını belirleyebilir. Eğer böyle bir belirleme yapılmamışsa kaynak port olarak herhangi boş bir port seçilir. Örneğin:

```

#client.py

import socket

DEST_PORT = 5050
SOURCE_PORT = 6890

try:
    with socket.socket() as clientSock:
        clientSock.bind(('', SOURCE_PORT))
        clientSock.connect(('127.0.0.1', DEST_PORT))

        print('connected')
        while True:
            b = clientSock.recv(1000)
            s = b.decode('UTF-8')
            if s == 'quit':
                break
            print(s)
except socket.error as msg:
    print('Socket error:{}' .format(msg))

```

Çok Client'lı Server Uygulamaları

Çok client'lı server uygulamalarında her client'la bağlantı kurabilmek için server programın yeniden accept uygulaması gereklidir. Yani her accept işlemi yeni bir client ile bağlantı sağlamaktadır. Bu durumda server'in bir döngü içerisinde accept uygulaması uygun olur. Ancak aynı anda birden fazla client ile server nasıl konuşacaktır? Bu tür durumlarda server bir client ile konuşurken diğer client'larla aynı anda konuşamayacağından sorunlar oluşur. İşte bunun çeşitli modeller kullanılabilmektedir.

Thread modelinde server her client bağlantısı gerçekleştiğinde yeni bir thread açar. O thread'le o client konuşur. Böylece server bir client ile konuşurken bloke olursa bu işlemden diğer client konuşmaları etkilenmez. Aşağıda örnek bir thread modelli multi client uygulama verilmiştir. Bu uygulamada client server'a bağlanıp server'a mesajlar göndermektedir.

```

#server.py

import socket
import threading

PORT = 5050

def main():

```

```

try:
    with socket.socket() as serverSock:
        serverSock.bind(('', PORT))
        serverSock.listen(8)

    print('Waiting for connection...')

    while True:
        clientSock, clientAddr = serverSock.accept()
        print('Connected:{}'.format(clientAddr))
        thread = threading.Thread(target=threadProc, args=(clientSock, clientAddr))
        thread.start()

except socket.error as msg:
    print('Socket error:{}'.format(msg))

def threadProc(clientSock, clientAddr):
    try:
        while True:
            b = clientSock.recv(1000)
            if not b:
                break
            s = b.decode('UTF-8')
            if s == 'quit':
                break
            print('{}: {}'.format(clientAddr, s))

    except socket.error as msg:
        print('Socket error:{}'.format(msg))
    finally:
        clientSock.shutdown(socket.SHUT_RDWR)
        clientSock.close()

main()

#client.py

import socket

PORT = 5050

def main():
    try:
        with socket.socket() as clientSock:
            clientSock.connect(('127.0.0.1', PORT))

            print('Connected...')
            while True:
                s = input('Yazi giriniz:')
                b = s.encode('UTF-8')
                clientSock.send(b)
                if s == 'quit':
                    break
            clientSock.shutdown(socket.SHUT_RDWR)
    except socket.error as msg:
        print('Socket error:{}'.format(msg))

main()

```

Şimdi buradaki server'a gelen yazının tersini client'a gönderecek biçimde değiştirelim:

```
#server.py
```

```

import socket
import threading

PORT = 5050

def main():
    try:
        with socket.socket() as serverSock:
            serverSock.bind(('', PORT))
            serverSock.listen(8)

            print('Waiting for connection...')

            while True:
                clientSock, clientAddr = serverSock.accept()
                print('Connected:{}'.format(clientAddr))
                thread = threading.Thread(target=threadProc, args=(clientSock, clientAddr))
                thread.start()

    except socket.error as msg:
        print('Socket error:{}'.format(msg))

def threadProc(clientSock, clientAddr):
    try:
        while True:
            b = clientSock.recv(1000)
            if not b:
                break
            s = b.decode('UTF-8')
            if s == 'quit':
                break
            print('{}: {}'.format(clientAddr, s))

            sr = s[::-1]
            print(sr)
            br = sr.encode('UTF-8')
            clientSock.send(br)

    except socket.error as msg:
        print('Socket error:{}'.format(msg))
    finally:
        clientSock.shutdown(socket.SHUT_RDWR)
        clientSock.close()

main()

```

```

#client.py

import socket

PORT = 5050

def main():
    try:
        with socket.socket() as clientSock:
            clientSock.connect(('127.0.0.1', PORT))

            print('Connected...')
            while True:
                s = input('Yazi giriniz:')
                b = s.encode('UTF-8')
                clientSock.send(b)
                if s == 'quit':

```

```

        break
    br = clientSock.recv(1000)
    if not br:
        break
    sr = br.decode('UTF-8')
    print('message from server: {}'.format(sr))
    clientSock.shutdown(socket.SHUT_RDWR)
except socket.error as msg:
    print('Socket error:{}'.format(msg))

main()

```

Bir soket yaratıldıktan sonra artık onunla bir dosyaymış gibi işlem yapabiliriz. Bunun için makefile metodu kullanılmaktadır. makefile metodunun parametrik yapısı şöyledi:

```
socket.makefile(mode='r', buffering=None, *, encoding=None, errors=None, newline=None)
```

Metodun birinci parametresi open fonksiyonundaki gibidir. Ancak yalnızca 'r', 'w' ve 'b' modları kullanılabilir. Bu metot bize bir file nesnesi verir. Böylece biz sanki sokette bir dosyaymış gibi işlem yapabiliriz. Örneğin soketten bir satır okumak için doğrudan dosya nesnesi ile readline методu çağrılabılır. Oluşturulan bu dosya nesnesi tamponlu bir biçimde çalışmaktadır. Yani aslında bu nesne soketten daha fazla bilgiyi kendi tamponuna okur ve bize daha önce okuduğu bilgiyi verir. Benzer durum yazma için de geçerlidir.

Client-Server Arasında Mesajlaşmalar

Client server programa pek çok şey yaptırabilir. Pekiyi neyi yaptıracagini client server'a nasıl iletecektir? İşte client ilse server arasında bu istekler ve sonuçlar çeşitli mesajlaşmalarla gerçekleştirilir. Mesajlaşmalar text ya da binary düzeyde yapılabilir. Ancak text tabanlı mesajlaşmalar daha kolay gerçekleştirilirler. Bu nedenle ilk tercih edilecek mesajlaşma formatı text formattır. Aslında Internet'in uygulama katmanındaki FTP, TELNET, SSH, HTTP, POP3, SMTP gibi protokoller hep text tabanlı mesajlaşma yapmaktadır. Örneğin dört işlem yapan bir server düşünelim. Client server'a yaptıracagi işlemleri birer komut olarak yazışal biçimde oluşturur:

ADD	Toplama
SUB	Çıkarma
MULTIPLY	Çarpma
DIVIDE	Bölme

Bu durumda client'in server'a göndereceği mesaj formatı şöyle olabilir:

```
<islem türü> <operand1><operand2>\n
```

Örneğin:

```
ADD 100 200
```

Server'in da client'a gönderdiği mesajın formatı şöyle olabilir:

```
RESULT: <sonuç>
```

Örneğin bir char programı nasıl tasarlanabilir. Chat programında client önce server'a TCP protokolü ile bağlanır. Buna fiziksel bağlantı diyebiliriz. Bundan sonra client server'a username ve password bilgileri gönderir. Server bunu kendi elindeki username ve password ile karşılaştırır. Eğer bir uyuşma varsa server client'i sisteme kabul eder. Buna mantıksal bağlantı diyebiliriz. Şimdi client artık diğer login olmuş kullanıcıları dağıtılması için server'a bir mesaj gönderir. Server da tüm client'lara bir mesaj olarak bunu iletir. Böyle bir programda kabaca mesajlar şöyle düzenlenlenebilir:

Client'tan Server'a Mesajlar

```
LOGIN <user name> <password>\n
PUTMSG <message text>\n\n
LOGOUT
```

Server'dan Client'a Mesajlar

```
OK\n
MESSAGE <message text>\n\n
ERROR <error message>\n
LOGIN_CLIENT <user name>\n
LOGOUT_CLIENT <client user name>\n
```

Genel olarak client'in server'a gönderdiği her mesaj için server'in bir yanıt vermesi iyi tekniktir. Yukarıdaki örnekte server işlem olumluysa OK mesajı olumsuzsa ERROR mesajı döndürmektedir.

Şimdi dört işlem yapan bir client-server TCP uygulaması yazalım. Client'tan server'a mesajlar şunlardır:

```
ADD <operand1> <operand2>
SUB <operand1> <operand2>
MULTIPLY <operand1> <operand2>
DIVIDE <operand1> <operand2>
```

Server'dan client'a gönderilen mesajlar da şunlardır:

```
RESULT <sonuç>
ERROR <metin>
```

Server client programları aşağıda verilmiştir:

```
#server.py

import socket
import threading

PORT = 5050

def main():
    try:
        with socket.socket() as serverSock:
            serverSock.bind(('', PORT))
            serverSock.listen(8)

            print('Arithmetic server running...')

            while True:
                clientSock, clientAddr = serverSock.accept()
                print('Connected Client:{}' .format(clientAddr))
                thread = threading.Thread(target=threadProc, args=(clientSock, clientAddr))
                thread.start()

    except socket.error as msg:
        print('Socket error:{}' .format(msg))

def add_proc(fileWrite, param1, param2):
    fileWrite.write('RESULT {}\n'.format(param1 + param2))
    fileWrite.flush()

def sub_proc(fileWrite, param1, param2):
    fileWrite.write('RESULT {}\n'.format(param1 - param2))
    fileWrite.flush()
```

```

def multiply_proc(fileWrite, param1, param2):
    fileWrite.write('RESULT {}\n'.format(param1 * param2))
    fileWrite.flush()

def divide_proc(fileWrite, param1, param2):
    fileWrite.write('RESULT {}\n'.format(param1 / param2))
    fileWrite.flush()

cmds = {'ADD': add_proc, 'SUB': sub_proc, 'MULTIPLY': multiply_proc, 'DIVIDE': divide_proc}

def threadProc(clientSock, clientAddr):

    fileRead = clientSock.makefile('r')
    fileWrite = clientSock.makefile('w')

    try:
        while True:
            line = fileRead.readline()

            args = line.split()
            if args[0] == 'quit':
                break
            proc = cmds.get(args[0], None)
            if not proc:
                fileWrite.write('ERROR "invalid operation"\n')
                fileWrite.flush()
                continue
            if len(args) != 3:
                fileWrite.write('ERROR "two operands must be specified"\n')
                fileWrite.flush()
                continue
            try:
                param1 = float(args[1])
                param2 = float(args[2])
            except:
                fileWrite.write('ERROR "invalid operand"\n')
                continue
            proc(fileWrite, param1, param2)
            print('Message From Client {}, Command: {}'.format(clientAddr, line))

    except socket.error as msg:
        print('Socket error:{}'.format(msg))
    finally:
        clientSock.shutdown(socket.SHUT_RDWR)
        clientSock.close()
        fileRead.close()
        fileWrite.close()
        print('Client Logout {}, Command: {}'.format(clientAddr, line))

main()

```

```

#client.py

import socket

PORT = 5050

def main():
    try:
        with socket.socket() as clientSock:
            clientSock.connect(('127.0.0.1', PORT))

            print('Arithmetic client connected...')


```

```

fileRead = clientSock.makefile('r')
fileWrite = clientSock.makefile('w')

while True:
    s = input('CMD>').strip()

    if s == '':
        continue
    fileWrite.write(s + '\n')
    fileWrite.flush()
    if s == 'quit':
        break
    response = fileRead.readline()
    print(response, end='')

clientSock.shutdown(socket.SHUT_RDWR)
except socket.error as msg:
    print('Socket error:{}' .format(msg))

main()

```

TCP Server Programının Daha Zahmetsiz Oluşturulması

TCP server programının daha zahmetsiz oluşturulması için hazır bazı sınıflar da bulundurulmuştur. Bu sınıflar socketserver isimli modülün içerisindeindedir. Aslında bu modülde UDP server için de sınıflar vardır. Ancak biz bu noktada TCP server sınıfları üzerinde duracağız. Zahmetsiz bir TCP server programı socketserver modülü kullanılarak şöyle yazılır:

- 1) Bu modelde aslında server işlevsellüğünün önemli bölümünü BaseRequestHandler isimli sınıf yapmaktadır. Yani listen ve accept dahil olmak üzere tüm işlemler aslında bu sınıf tarafından yapılmaktadır. Bu nedenle ilk iş olarak BaseRequestedHandler isimli sınıfın bir sınıf türetilir.
- 2) BaseRequestHandler sınıfından türetilmiş olan sınıfın handle isimli metot override edilir. Bu metot server'a bir bağlantı isteği geldiğinde taban sınıf tarafından çağrılmaktadır.

```

class MyTCPServer(socketserver.BaseRequestHandler):
    def handle(self):
        pass

```

- 3) Server'ı kontrol etmek için bir kontrol nesnesi yaratmamız gereklidir. Kontrol nesnesi olarak TCPServer sınıfı ya da ThreadingTCPServer sınıfı kullanılabilir. Bu sınıfların dunder init metotları bizden iki parametre isterler. Birinci parametre server'in dinleme yapacağı network adresi ve port numarasını içeren demettir. İkinci parametre ise BaseRequestHandler sınıfından türetilen sınıfın ismidir. (Anımsanacağı gibi Python'da sınıf isimleri o sınıf bilgilerinin bulunduğu type türünden bir sınıf nesnesinin adresini belirtmektedir.)

- 4) Son olarak bizim artık server programı çalışır duruma getirmemiz gerekmektedir. Bunun için kontrol sınıfının serve_forever isimli metodu çağrılır. Örneğin:

```

def main():
    try:
        tcpServer = socketserver.TCPServer(('', PORT), MyTCPServer)
        tcpServer.serve_forever()
        print('ok')
    except socket.error as msg:
        print('Socket error:{}' .format(msg))

main()

```

Bu durumda server yeni bağlantı olduğunda handler metodunu çağırır. Biz de bu metotta server programın yapacağını yazarız. Tabii aslında bu haliyle çok fazla bir kolay bize sunulmamaktadır. Zaten Python'da TCP server programının kendisini yazmak oldukça kolaydır.

Kontrol sınıfı TCPServer sınıfıyla oluşturulursa tek bir client bağlantısı söz konusu olur. Örneğin:

```
#tcpserver.py

import socket
import socketserver

PORT = 5050

class MyTCPServer(socketserver.BaseRequestHandler):
    def handle(self):
        print('new client connected...')
        while True:
            b = self.request.recv(1000)
            if not b:
                break
            s = b.decode('UTF-8')
            if s == 'quit':
                break
            print(s)

def main():
    try:
        tcpServer = socketserver.TCPServer(('', PORT), MyTCPServer)
        print('waiting for connection...')
        tcpServer.serve_forever()
        print('ok')
    except socket.error as msg:
        print('Socket error:{}' .format(msg))

main()

#tcpclient.py

import socket

PORT = 5050

def main():
    try:
        with socket.socket() as clientSock:
            clientSock.connect(('127.0.0.1', PORT))

            print('connected...')
            while True:
                s = input('Yazi giriniz:')
                b = s.encode('UTF-8')
                clientSock.send(b)
                if s == "quit":
                    break

            clientSock.shutdown(socket.SHUT_RDWR)
    except socket.error as msg:
        print('Socket error:{}' .format(msg))

main()
```

Kontrol sınıfı olarak TCPServer kullanılmışsa şu noktalar göz önünde bulundurulmalıdır:

- Bu durumda server aynı anda tek bir client ile konuşacak durumdadır.
- Bir client bağlantısı sonlandığında server yeni bir client ile yeniden bağlanabilir.

Kontrol sınıfları (yani TCPServer ve ThreadingTCPServer) BaseServer isimli bir sınıfından türetilmiştir. Kontrol sınıfının bazı faydalı metotları da vardır. Örneğin shutdown ve close metotları server'ı durdurmak için kullanılabilirlerdir. Diğer metotları Python dokümanlarından inceleyebilirsiniz.

socketserver modülüyle biz çok client'lı server uygulamaları da yazabiliriz. Bunun için kontrol sınıfı olarak ThreadingTCPServer sınıfı kullanılır. Aslında diğer tüm işlemler aynıdır. Bu durumda her handle metodu ayrı bir thread yaratılarak çağrılmaktadır. Böylece multiclient bağlantı yapılmaktadır. Örneğin:

```
#tcpserver.py

import socket
import socketserver

PORT = 5050

class MyTCPServer(socketserver.BaseRequestHandler):
    def handle(self):
        print('new client connected...')
        while True:
            b = self.request.recv(1000)
            if not b:
                break
            s = b.decode('UTF-8')
            if s == 'quit':
                break
            print(s)

def main():
    try:
        tcpServer = socketserver.ThreadingTCPServer(('', PORT), MyTCPServer)
        print('waiting for connection...')
        tcpServer.serve_forever()
        print('ok')
    except socket.error as msg:
        print('Socket error:{}\n'.format(msg))

main()
```

```
#tcpclient.py
```

```
import socket

PORT = 5050

def main():
    try:
        with socket.socket() as clientSock:
            clientSock.connect(('127.0.0.1', PORT))

            print('connected...')
            while True:
                s = input('Yazi giriniz:')

                b = s.encode('UTF-8')
                clientSock.send(b)
```

```

    if s == "quit":
        break

    clientSock.shutdown(socket.SHUT_RDWR)
except socket.error as msg:
    print('Socket error:{}' .format(msg))

main()

```

Biz yukarıdaki örneklerde taban sınıf olarak BaseRequestHandler sınıfını kullandık. Ayrıca taban sınıf olarak StreamRequestHandler sınıfı da kullanılabilir. Bu durumda sınıfın rfile ve wfile isimli elemları birer dosya nesnesi durumundadır. Yani başka bir deyişle StreamRequestHandler sınıfı BaseRequestHandler sınıfının makefile yapılmış hali gibidir. Örneğin:

```

#tcpserver.py

import socket
import socketserver

PORT = 5050

class MyTCPServer(socketserver.StreamRequestHandler):
    def handle(self):
        print('new client connected...')
        while True:
            b = self.rfile.readline()
            if not b:
                break
            s = b.decode('UTF-8')
            if s == 'quit':
                break
            print(s)

def main():
    try:
        tcpServer = socketserver.ThreadingTCPServer(('', PORT), MyTCPServer)
        print('waiting for connection...')
        tcpServer.serve_forever()
        print('ok')
    except socket.error as msg:
        print('Socket error:{}' .format(msg))

main()

```

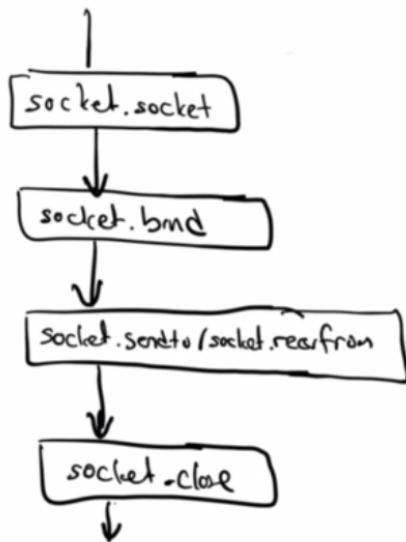
Burada tek farklılık artık request elemanı terine rfile elemanın kullanılmasıdır.

UDP Haberleşmesi

Anımsanacağı gibi UDP bağlantısız datagram tabanlı bir haberleşme sunmaktadır. Client-Server terimleri UDP için çok anlamlı olmasa da bu protokolde yine istekte bulunulan tarafa server, istek yapana tarafa da client denilmektedir.

UDP Server Programın Yazımı

UDP Server program tipik olarak şu aşamalardan geçilerek yazılır.



Bu durumda örnek bir UDP server programı şöyle yazılabılır:

```

import socket

PORT = 5051

def main():
    try:
        with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as sock:
            sock.bind(('', PORT))
            print('Waiting for data...')
            while True:
                b, addr = sock.recvfrom(1024)
                s = b.decode('UTF-8')
                if s == 'quit':
                    break
                print('{}: {}'.format(addr, s))

    except socket.error as msg:
        print('Socket error:{}'.format(msg))

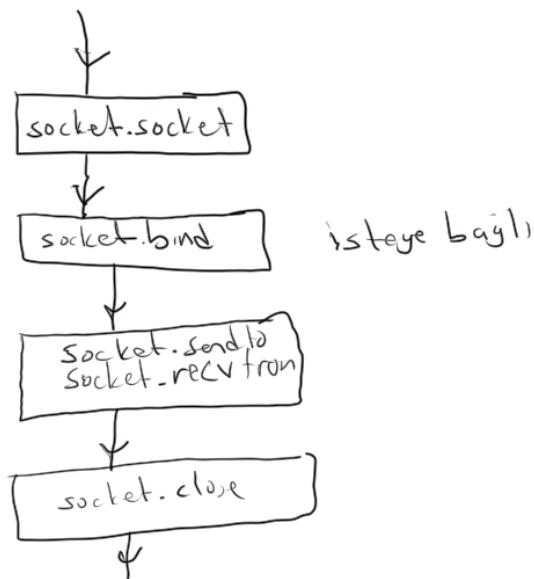
main()

```

Burada recvfrom metodu iki elemanlı bir demete geri dönmektedir. Demetin birinci elemanı UDP paketini oluşturan bytes türünden bilgiyi ikinci eleman ise bilgiyi gönderen client'in IP adresini ve port numarasını belirtmektedir.

UDP Client Programının Yazımı

UDP Client program yukarıda da belirtildiği gibi bir connect işlemi yapmaz. Doğrudan server'in IP adresini ve port numarasını belirterek sendto metodıyla gönderme yapar:



Örnek bir client program da şöyle yazılır:

```

#udpclient.py

import socket

PORT = 5051

def main():
    try:
        with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as clientSock:
            while True:
                s = input('Yazi giriniz:')
                b = s.encode('UTF-8')
                clientSock.sendto(b, ('95.5.148.84', PORT))
                if s == 'quit':
                    break
    except socket.error as msg:
        print('Socket error:{}' .format(msg))

main()

```

Python'da Matemiksel ve İstatistiksel Veri Analizinde Kullanılan Yaygın Kütüphaneler

Python'ın standart kütüphanesinde bazı ve matemiksel istatistiksel işlemleri yapan modüller vardır. Ancak bunların yeterliliği şüphelidir. Bu nedenle her ne kadar standart kütüphanenin bir parçası olmasa da birtakım kütüphaneler çok yaygın olarak kullanılmaktadır. Bunların en önemlileri Numpy, SciPy ("saypay" biçiminde okunuyor) ve Pandas isimli kütüphanelerdir. NumPy en taban kütüphanedir. SciPy Numpy kullanılarak gerçekleştirilmiştir. Benzer biçimde Pandas da NumPy kütüphanesinin üzerine kurulmuştur. Biz kursumuzun bu bölümünde bu kütüphanelerin kullanımları üzerinde duracağız. Bu kütüphanelerin hepsi C Programlama Dilinde yazılmıştır. Dolayısıyla çok hızlı olma iddiasındadır.

NumPy Kütüphanesinin Kurulumu

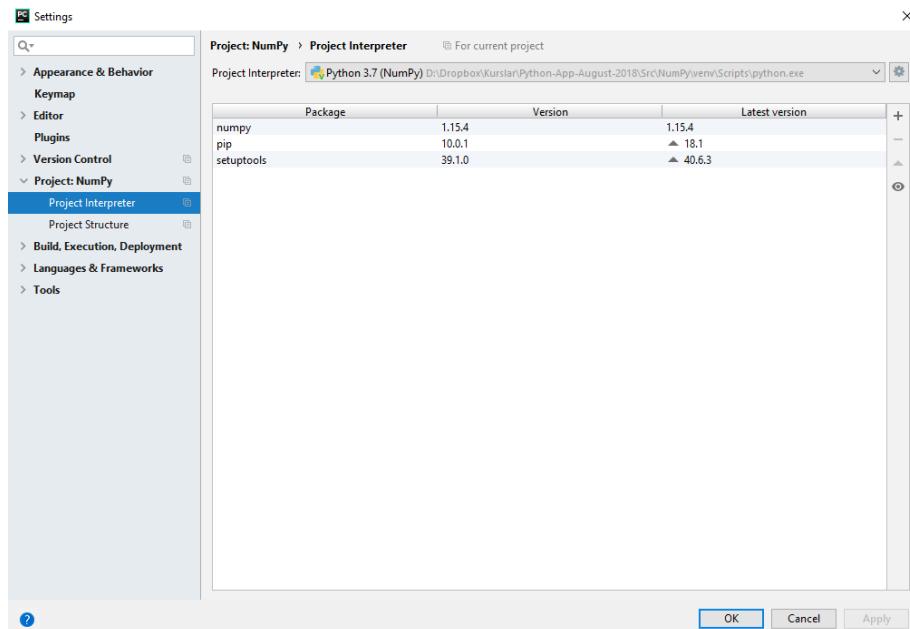
NumPy standart kütüphaneye dahil olmadığı için ayrıca kurulması gerekmektedir. Kurulum yine pip kullanılarak basit biçimde aşağıdaki gibi yapılabilir:

```
python -m pip install numpy
```

Ya da doğrudan pip programının kendisiyle de kurulum şöyle yapılabilir:

```
pip install numpy
```

PyCharm IDE'sinde projede seçilen "Interpter"a dikkat ediniz. PyCharm "Virtual Environment" adı altında her proje için default olarak bir Python yorumlayıcısını da bulundurmaktadır. Yine paketler proje özgü olarak bu ortamda saklanmaktadır. Bu nedenle PyCharm'da ya bu "Project Interpeeter"ı değiştirilmeli ya da mevcut "Project Interpreter"ınapaket yeniden yüklenmelidir. Örneğin:



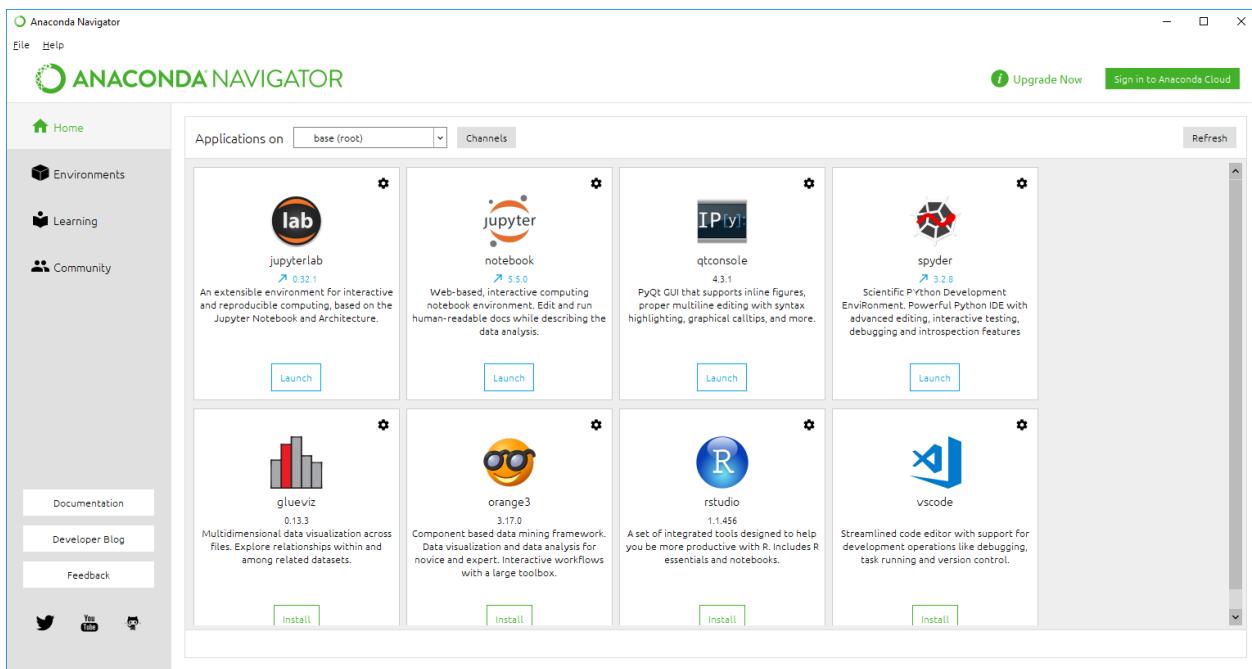
Biz Python programlarında numpy kütüphanesini import ederken çoğu kez kolay yazım sağlamak için numpy ismini aşağıdaki biçimde np olarak kullanacağız:

```
import numpy as np
```

Anaconda Dağıtımında NumPy Kullanımı

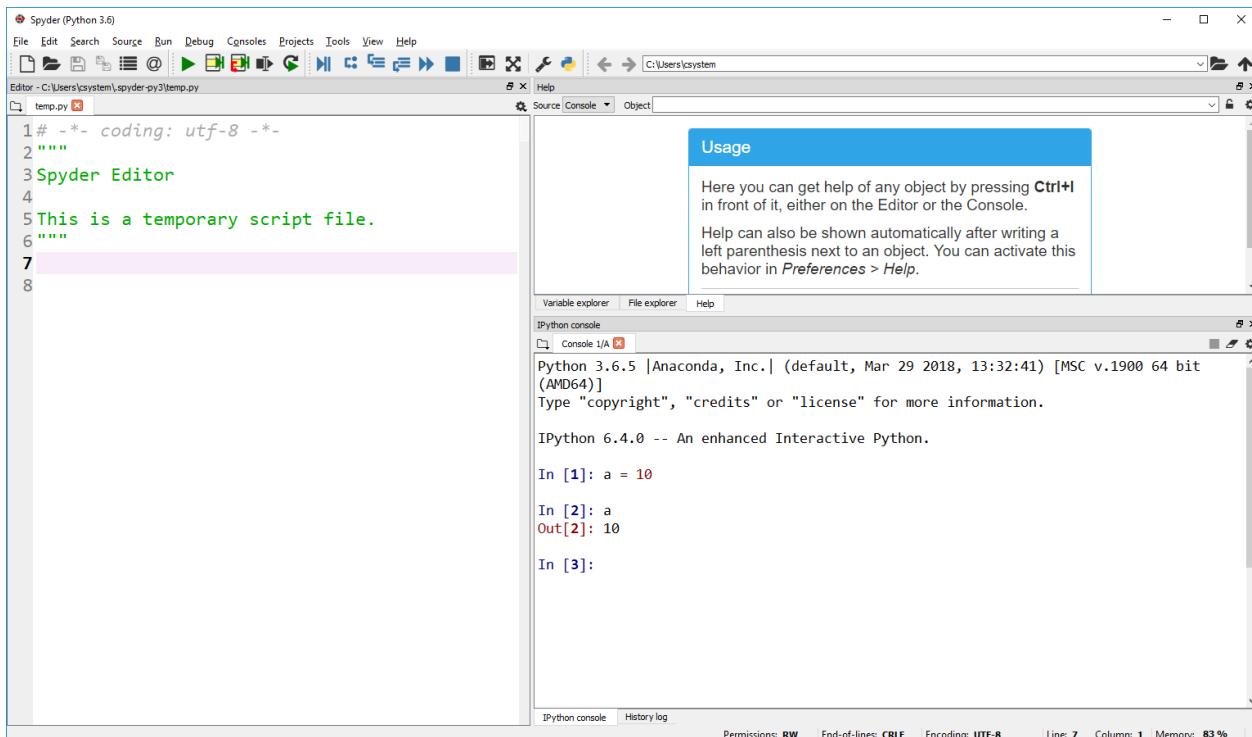
Python'ın Anaconda dağıtımını kendi içerisinde yüzlerce paketi zaten barındırmaktadır. Bunun içerisinde NumPy, SciPy ve Pandas paketleri de vardır. Dolayısıyla Anaconda dağıtımını kullanılıyorsa bu kütüphaneler için kurulunum yapılmasına gerek yoktur. Biz kursumuzda bu noktada sizleri alıştırmak için Anconda dağıtımını da kullanacağız.

Anaconda dağıtımının ana programı "Anaconda Navigator" isimli programdır.



Anaconda dağıtımının IDE'si Spyder isimli IDE'dir. Spyder oldukça basit bir kullanıma sahiptir. Spyder IDE'sini PyCharm'a eşdeğer kabul edebilirsiniz.

Anaconda dağıtımında kullanılan komut satırına IPython denilmektedir. Aslıda IPython ayrıca da yüklenebilmektedir. Yani Anaconda dağıtımına özgü değildir.



Yine Anconda dağıtımında da tek bir dosya üzerinde çalışma ya da proje yaratarak çalışma modelleri vardır. Proje çalışma biçimi tamamen PyCharm benzemektedir.

NumPy Dokümantasyonu

NumPy için resmi dokümantasyon aşağıdaki istende bulundurulmuştur:

<https://docs.scipy.org/doc/>

Programcı NumPy'İN tüm sınıflarına ve fonksiyonlarına ilişkin bilgileri buradan edinebilir.

Python'da NumPy Kütüphanesinin Kullanımı

Python'IN temel veri yapıları (listeler, demetler, kümeler, sözlükler) matematiksel ve istatistiksel veri analizi için uygun olsa da çok uygun değildir. Genellikle R, Matlab gibi dillerde olduğu gibi matematiksel ve istatistiksel veri analizinde dizisel işlemler daha uygundur. Oysa Python'IN veri yapıları buna uygun değildir. İşte NumPy'da ismine ndarray denilen yeni bir veri yapısı oluşturulmuş ve işlemlerde hep bu veri yapısı kullanılmıştır. Python'IN built-in list veri yapısıyla NumPy'İN ndarray veri yapısı arasındaki farkı basit bir örnekle söyle açıklayabiliriz:

```
>>> import numpy  
>>> a = [1, 2, 3, 4, 5]  
>>> b = [10, 20, 30, 40, 50]  
>>> c = a + b  
>>> c  
[1, 2, 3, 4, 5, 10, 20, 30, 40, 50]  
>>> x = numpy.array([1, 2, 3, 4, 5])  
>>> y = numpy.array([10, 20, 30, 40, 50])  
>>> z = x + y  
>>> z  
array([11, 22, 33, 44, 55])
```

Burada önce Python'IN built-in list nesesi ile iki list toplanmıştır. Bildiğiniz gibi biz iki list nesnesini topladığımızda yeni bir list nesnesi yaratılır. Bu list nesnesi iki nesnenin elemanlarının birleşiminde olmaktadır. Oysa NumPy kütüphanesinin ndarray denilen sınıfı söz konusu olduğunda bu sınıf türünden iki nesneyi topladığımızda karşılıklı elemanlar toplanmaktadır. İşte bu nedenlerden dolayı NumPy'İN en önemli temel veri yapısı ndarray denilen sınıfıdır. Biz önce bu ndarray sınıfını inceleyeceğiz.

NumPy'da ndarray Veri Yapısı

ndarray vektörel bir liste oluşturmak için kullanılan NumPy paketine özgü temel bir veri yapısıdır. Dolayısıyla SciPy ve Pandas kütüphaneleri de hep bu ndarray veri yapısını kullanmaktadır. Bu veri yapısı yukarıda da belirtildiği gibi ndarray sınıfı ile temsil edilmektedir.

Bir ndarray nesnesini yaratmanın çeşitli yolları vardır. En temel yol array isimli fonksiyonu kullanmaktadır. array fonksiyonu bizden dolaşılabilir bir nesne alıp ondan ndarray nesnesi yaratmaktadır. Örneğin:

```
>>> import numpy as np  
>>> a = np.array([1, 2, 3, 4, 5])  
>>> a  
array([1, 2, 3, 4, 5])  
>>> print(a)  
[1 2 3 4 5]  
>>> type(a)  
<class 'numpy.ndarray'>
```

Örneğin:

```
>>> a = np.array((1, 2, 3, 4, 5))  
>>> a  
array([1, 2, 3, 4, 5])  
>>> print(a)  
[1 2 3 4 5]  
>>> type(a)  
<class 'numpy.ndarray'>
```

Bilindiği gibi Python'ın built-in türü olan `int` sınırsız uzunlukta bir tamsayı türündür. Halbuki nümerik uygulamalarda işlemcinin işleyebildiği uzunlukta veriler çok daha etkin işlemelere sokulabilmektedir. Bu nedenle `ndarray` nesnesinde kullanılabilecek özel türler tanımlanmıştır. Bu türler de şüphesiz Python'da aslında bir sınıf görünümündedir. Bu türlerde "dtype" denilmektedir. Her bir `ndarray` nesnesinin bir `dtype` türü vardır. Temel `dtype` türleri şunlardan oluşmaktadır:

```
int (default mimariye bağlı int32 ya da int64)
int8
int16
int32
int64
uint (mimariye bağlı default uint32 ya da uint64)
uint8
uint16
uint32
uint64
float (default float64)
float16
float32
float64
float128
complex (default complex64)
complex64
complex128
complex256
```

Tür isimlerindeki sayılar o türün bir uzunluğuyla ilgilidir. Bir `ndarray` nesnesinin `dtype`'ının ne olacağı nesne içerisindeki en geniş türe bağlıdır. Ancak bir `ndarray` nesnesinin bütün elemanları aynı türden olmak zorundadır. Bunu Python'ın built-in list türüyle karıştırmayınız. `ndarray` nesnesinin `dtype`'ını sınıfın `dtype` isimli property elemanı ile elde edebiliriz. Örneğin:

```
array([1. , 2.2, 3. , 4. , 5. ])
>>> a.dtype
dtype('float64')
```

Örneğin:

```
>>> a = np.array([1, 2, '3', 4, 5])
>>> a
array(['1', '2', '3', '4', '5'], dtype='<U11')
>>> a.dtype
dtype('<U11')
```

Eğer istersek biz `array` fonksiyonunda `dtype` isimli argümanı kullanarak `dtype` türünü istediğimiz gibi belirleyebiliriz. Örneğin:

```
>>> a = np.array([1, 2, 3, 4, 5], dtype=np.int16)
>>> a
array([1, 2, 3, 4, 5], dtype=int16)
```

Eğer `dtype` belirtilmemişse `array` fonksiyonu `dtype` olarak tamsayılar için verilen değerleri karşılayacak biçimde `int32`, `int64` türlerinden birini almakatdır. Benzer biçimde eğer `dtype` belirtilmemişse ve dolaşılabilir nesnede noktalı bir sayı varsa `array` fonksiyonu default `dtype` türünü `float64` olarak almaktadır. Örneğin:

```
>>> a = np.array([1, 2, 3, 4, 5])
>>> a.dtype
dtype('int32')
```

Örneğin:

```
>>> a = np.array([1, 2, 3.23])
>>> a.dtype
dtype('float64')
```

ndarray yalnızca tek boyutlu değil çok boyutlu diziyi de temsil etmektedir. Biz bu fonksiyona parametre olarak liste listesi gibi dolaşılabilir bir nesne verirsek o da bize çok boyutlu bir ndarray nesnesi verir. Örneğin:

```
>>> a = np.array([[1, 2], [3, 4], [5, 6]], dtype=np.int64)
>>> a
array([[1, 2],
       [3, 4],
       [5, 6]], dtype=int64)
>>> print(a)
[[1 2]
 [3 4]
 [5 6]]
```

Bir ndarray nesnesinin boyutu onun ndim property'si ile elde edilebilir. Örneğin:

```
>>> a = np.array([[1, 2], [3, 4], [5, 6]], dtype=np.int64)
>>> a.ndim
2
>>> b = np.array([1, 2, 3, 4, 5])
>>> b.ndim
1
```

Aslında array fonksiyonun ndmin isimli bir argümanı da vardır. Biz listeyi tek boyutlu girsek bile bu ndim argümanı ile boyutu ayarlayabiliriz. Örneğin:

```
>>> a = np.array([1, 2, 3, 4, 5], ndmin=2)
>>> a
array([[1, 2, 3, 4, 5]])
>>> print(a)
[[1 2 3 4 5]]
```

Aslında array fonksiyonun birinci parametresine biz dolaşılabilir bir nesne vermeyebiliriz. Bu durumda skaler bir ndarray nesnesi oluşturulur. Bu pek kullanılan bir durum değildir. Örneğin:

```
>>> a = np.array(10)
>>> a
array(10)
>>> print(a)
10
>>> type(a)
<class 'numpy.ndarray'>
>>> a.ndim
0
```

ndarray Nesnesi Yaratan Diğer Fonksiyonlar

Biz yukarıda ndarray nesnesi yaratmak için array fonksiyonunu kullandık. Aslında ndarray nesnesi yaratmak için başka fonksiyonlar da bulunmaktadır.

zeros isimli fonksiyon içi sıfır dolu bir ndarray nesnesi yaratmak için kullanılmaktadır. Fonksiyonun parametrik yapısı şöyledir:

```
numpy.zeros(shape, dtype=float, order='C')
```

Fonksiyonun birinci parametresi sıfırlarla dolu olacak ndarray nesnesinin boyutunu belirtir. Boyut tipik olarak bir demetle ifade edilmektedir. Demetin her elemanı boyut uzunluklarını belirtmektedir. Bu parametre demet yerine doğrudan da geçilebilir. Bu durumda ndarray nesnesi 1 boyutlu dizi biçiminde olacaktır. Örneğin:

```
>>> a = np.zeros(10)
>>> a
array([0., 0., 0., 0., 0., 0., 0., 0., 0.])
>>> b = np.zeros((3, 2))
>>> b
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
>>> c = np.zeros((10,))
>>> c
array([0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

zeros fonksiyonun dtype parametresi dtype türünü belirtir. Bunun default olarak float olduğunu görüyorsunuz. order parametresi pek çok fonksiyonda bulunmaktadır. Bu nesnenin içel diziliminin Sırun temelli mi ('C') yoksa satır temelli mi ('R') olduğunu belirtir. Bu konu ileride ele alınacaktır. Örneğin:

```
>>> c = np.zeros(10, dtype=np.int16)
>>> c
array([0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=int16)
```

ones isimli fonksiyon zeros fonksiyonu gibidir. Fakat diziyi 1'lerle doldurur. Örneğin:

```
numpy.ones(shape, dtype=None, order='C')
```

Örneğin:

```
>>> a = np.ones(10)
>>> a
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

diag isimli fonksiyon diagonaldeki vektörü oluşturmak için kullanılmaktadır. Parametrik yapısı şöyledir:

```
numpy.diag(v, k=0)
```

Fonksiyonun birinci parametresi matrisi ikinci parametresi diyagonal numarası belirtir. Örneğin:

```
>>> a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> b = diag(a)
>>> b = np.diag(a)
>>> b
array([1, 5, 9])
>>> b = np.diag(a, 1)
>>> b
array([2, 6])
>>> c = np.diag(a, -1)
>>> c
array([4, 8])
```

Asıl diyagonalın numarası 0'dır. Yukarıya doğru pozitif, aşağıya doğru negatif numralandırma yapılmaktadır.



arange isimli fonksiyon Python'ın standart range fonksiyonu gibidir. Parametrik yapısı şöyledir:

```
numpy.arange([start, ]stop, [step, ]dtype=None)
```

Örneğin:

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = np.arange(5, 10)
>>> b
array([5, 6, 7, 8, 9])
>>> c = np.arange(10, 20, 2)
>>> c
array([10, 12, 14, 16, 18])
>>> c = np.arange(20, 10, -1)
>>> c
array([20, 19, 18, 17, 16, 15, 14, 13, 12, 11])
```

linspace isimli fonksiyon doğrusal biçimde iki aralıkta değer üretmektedir. Fonksiyonun parametrik yapısı şöyledir:

```
numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)
```

Fonksiyonun birinci parametresi başlangıç, ikinci parametresi bitiş, üçüncü parametresi ise kaç tane değer üretileceğini belirtmektedir. Bitiş değeri dahildir. Örneğin:

```
>>> a = np.linspace(10, 20, 11)
>>> a
array([10., 11., 12., 13., 14., 15., 16., 17., 18., 19., 20.])
```

Örneğin:

```
>>> b = np.linspace(10, 20, 22)
>>> b
array([10.        , 10.47619048, 10.95238095, 11.42857143, 11.9047619 ,
       12.38095238, 12.85714286, 13.33333333, 13.80952381, 14.28571429,
       14.76190476, 15.23809524, 15.71428571, 16.19047619, 16.66666667,
       17.14285714, 17.61904762, 18.0952381 , 18.57142857, 19.04761905,
       19.52380952, 20.        ])
```

Bir ndarray nesnesi belirli bir değerin belirli bir sayıda doldurulmasıyla da oluşturulabilir. Örneğin biz 100 tane 10 değerinden oluşan bir ndarray nesnesi yaratmak isteyebiliriz. Bunun için full fonksiyonu kullanılmaktadır. full fonksiyonun parametrik yapısı şöyledir:

```
numpy.full(shape, fill_value, dtype=None, order='C')
```

Fonksiyonun birinci parametresi oluşturulacak ndarray nesnesinin boyutunu belirtir. Bu boyut demet biçiminde girilebilir. Eğer tekil değer girilirse tek boyutlu dizi anlaşıılır. İkinci parametre doldurum değeridir. Üçüncü parametre dtype türüdür (default sisteme bağlı olarak int32 ya da int64). Örneğin:

```

>>> a = np.full(10, 100)
>>> a
array([100, 100, 100, 100, 100, 100, 100, 100, 100, 100])
>>> b = np.full((5, 2), 100)
>>> b
array([[100, 100],
       [100, 100],
       [100, 100],
       [100, 100],
       [100, 100]])

```

empty isimli fonksiyon ilkdeğer verilmemiş biçimde bir ndarray nesnesi oluşturur. Eğer nesneye daha sonra ilkdeğer verilecekse ilkdeğer verme zahmeti ortadan kaldırılabilir. Büyük dizilerde bu çok az da olsa bir zaman kazancı sağlayabilmektedir. Fonksiyonun parametrik yapısı şöyledir:

```
numpy.empty(shape, dtype=float, order='C')
```

Yine fonksiyonun birinci parametresi dizinin boyutsal durumunu, ikinci parametresi dtype türüne temsil eder. Örneğin:

```

>>> a = np.empty(10, dtype=np.int32)
>>> a
array([ 862335334, 1717975139, 859256109, 926166369, 1647141170,
       758604851, 929312818, 892613941, 945961062, 0])

```

Oluşturulan ndarray nesnesinin içerisindeki değerlerin çöp değerler (garbage values) olduğuna dikkat ediniz. Dizi için bellekte neresi tahsis edilmişse orada daha önceden bulunan rastgele değerler dizi elemanlarında gözükmektedir.

Aslında mademki ndarray bir sınıfıtır. Onun da başlangıç metodu vardır. İşte ndarray sınıfın başlangıç metodu (dunder init) olan ndarray fonksiyonu bize empty fonksiyonundaki gibi çöp değerlerden oluşan ndarray nesnesi verir. ndarray fonksiyonun parametrik yapısı şöyledir:

```
ndarray(shape, dtype=float, buffer=None, offset=0, strides=None, order=None)
```

Fonksiyonun ilk iki parametresi empty fonksiyonuyla aynıdır. Diğer parametreler burada ele alınmayacağından emin olun. Örneğin:

```

>>> a = np.ndarray(10, dtype=np.int32)
>>> a
array([ 862335334, 1717975139, 859256109, 926166369, 1647141170,
       758604851, 929312818, 892613941, 945961062, 0])

```

Ayrıca bir de ndarray nesnesi yaratan zeros_like, ones_like, full_like ve empty_like isminde like'lı fonksiyonlar vardır. Bu like'lı fonksiyonlar bir ndarray nesnesini alıp onun boyutlarında ama içerisinde 0'lar (zeros_like), 1'ler (ones_like), belli değerler (full_like) ve çoğ deiğerler (empty_like) olacak biçimde ndarray yaratırlar. Örneğin:

```

>>> a = np.array([[1, 2], [3, 4], [5, 6]])
>>> a
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> b = np.zeros_like(a, dtype=np.int32)
>>> b
array([[0, 0],
       [0, 0],
       [0, 0]])
>>> c = np.full_like(a, 100)
>>> c
array([[100, 100],
       [100, 100],
       [100, 100]])

```

```

        [100, 100]])
>>> d = np.ones_like(a)
>>> d
array([[1, 1],
       [1, 1],
       [1, 1]])
>>> e = np.empty_like(a)
>>> e
array([[0, 0],
       [0, 0],
       [0, 0]])

```

like'lı fonksiyonların bizden dizinin boyutlarını (shape'ini) almadığına bu boyut bilgisini ona geçirdiğimiz dizilerden aldığına dikkat ediniz. Bu like'lı fonksiyonlara geçerdiğimiz dizilerin boyut belirtmenin dışında bir amacı yoktur. Başka bir deyişle örneğin:

```

>>> b = np.zeros(a.shape)
>>> b
array([[0., 0.],
       [0., 0.],
       [0., 0.]])

```

işlemi ile:

```

>>> b = np.zeros_like(a)
>>> b
array([[0, 0],
       [0, 0],
       [0, 0]])

```

işlemi tamamen eşdeğerdir.

identity isimli fonksiyon birim matris oluşturmak için kullanılmaktadır. Fonksiyonun parametrik yapısı şöyledir:

```
numpy.identity(n, dtype=None)
```

Fonksiyonun birinci parametresi birim kare matrisin satır (ya da sütun) sayısını belirtir. İkinci parametresi yine dtype türünü belirtmektedir. Örneğin:

```

>>> a = np.identity(5)
>>> a
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])

```

eye fonksiyonu da köşegeni 1 olan bir matris oluşturur. Ancak köşegeni numarayla belirtebiliriz. 0 değeri ana kösegendir. + değerler yukarı, - değerler aşağı yön belirtir. Örneğin:

```

>>> a = np.eye(5, k = +1)
>>> a
array([[0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.],
       [0., 0., 0., 0., 0.]])
>>> b = np.eye(5, k=-2)
>>> b
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])

```

```
[0., 0., 0., 0., 0.],  
[1., 0., 0., 0., 0.],  
[0., 1., 0., 0., 0.],  
[0., 0., 1., 0., 0.]])
```

Aslında burada görmedğimiz birkaç tane daha ndarray yaratan fonksiyon vardır. Bunları NumPy dokümanlarından inceleyebilirsiniz. Son olarak burada gördüğümüz fonksiyonların bir listesini yapalım:

```
array  
zeros  
ones  
diag  
arange  
linspace  
full  
empty  
ndarray  
zeros_like  
ones_like  
full_like  
empty_like  
identity  
eye
```

ndArray Nesnesinde İndeksleme ve Dilimleme

Nasıl list, tuple, str türleri dilimleniyorsa (slicing) benzer biçimde ndarray nesnesi de indekslenebilir ve dilimlenebilir. Örneğin:

```
>>> a = np.array(range(10, 110, 10))  
>>> a[5]  
60  
>>> a[7]  
80
```

ndarray değiştirilebilir (mutable) bir sınıfıtır. Dolayısıyla biz dizi elemanlarını daha sonra değiştirebiliriz. Örneğin:

```
>>> a = np.array(range(10, 110, 10))  
>>> a  
array([ 10,  20,  30,  40,  50,  60,  70,  80,  90, 100])  
>>> a[3] = 1000  
>>> a[7] = 5000  
>>> a  
array([ 10,  20,  30, 1000,   50,   60,   70, 5000,   90, 100])
```

İndekslemede negatif değerler yine uzunlukla toplama anlamına gelmektedir. (Yani -1'inci indeks son elemanı belirtmektedir.) Örneğin:

```
>>> a = np.array(range(10, 110, 10))  
>>> a[-1]  
100  
>>> a[-2]  
90
```

Çok boyutlu ndarray nesnesinin elemanlarına erişirken köşeli parantez içerisinde birden fazla indis belirtilir. (Halbuki built-in list sınıfında birden fazla köşeli parantezin kullanıldığını anımsayınız.) Örneğin:

```
>>> a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])  
>>> a  
array([[1, 2, 3],
```

```

[4, 5, 6],
[7, 8, 9])
>>> a[1, 2]
6
>>> a[2, 2]
9
>>> a[-1, 1]
8
>>> a[-1, -1]
9

```

Çok boyutlu ndarray nesnelerinde yine negatif indeksler kullanılabilir. Her boyutun negatif indeksi yine o boyutun uzunluğu ile toplanmaktadır.

Dilimle işlemi tamamen list, tuple ve str sınıflarındaki gibidir. Örneğin:

```

>>> a = np.array(range(10, 110, 10))
>>> b = a[3:5]
>>> b
array([40, 50])
>>> type(b)
<class 'numpy.ndarray'>

```

Örneğin:

```

>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[::2]
array([0, 2, 4, 6, 8])
>>> a[::-2]
array([9, 7, 5, 3, 1])

```

Yine biz dilimleme yoluyla elemanları değiştirebiliriz. Örneğin:

```

>>> a = np.array(range(10, 110, 10))
>>> a
array([ 10,  20,  30,  40,  50,  60,  70,  80,  90, 100])
>>> a[3:5] = [1000, 2000]
>>> a
array([ 10,  20,  30, 1000, 2000,   60,   70,   80,   90,  100])

```

ndarray nesnesine dilimleme yoluyla atama yapılırken atanan dolaşılabilir nesnenin uzunluğunun dilimlemeden elde edilecek uzunluktan fazla ya da eksik olmaması gereklidir. (Halbuki list sınıfında bunun mümkün olduğunu anımsayınız.) Eğer dilimlemede atanan değer dolaşılabilir bir nesne değilse ya da tek elemanlı dolaşılabilir bir nesne ise bu değer tüm dilimlenen elemanlara atanmış kabul edilir. Örneğin:

```

>>> a = np.array(range(10, 110, 10))
>>> a
array([ 10,  20,  30,  40,  50,  60,  70,  80,  90, 100])
>>> a[3:6] = [1000, 2000, 3000]
>>> a
array([ 10,  20,  30, 1000, 2000, 3000,   70,   80,   90,  100])
>>> a[3:6] = [1000, 2000]
Traceback (most recent call last):
  File "<pyshell#137>", line 1, in <module>
    a[3:6] = [1000, 2000]
ValueError: cannot copy sequence with size 2 to array axis with dimension 3
>>> a[3:6] = [1000, 2000, 3000, 4000, 5000]
Traceback (most recent call last):

```

```

File "<pyshell#138>", line 1, in <module>
    a[3:6] = [1000, 2000, 3000, 4000, 5000]
ValueError: cannot copy sequence with size 5 to array axis with dimension 3
>>> a[3:6] = 1000
>>> a
array([ 10,   20,   30, 1000, 1000, 1000,    70,    80,    90,   100])
>>> a[0:2] = [1000]
>>> a
array([1000, 1000,   30, 1000, 1000, 1000,    70,    80,    90,   100])

```

çok boyutlu ndarray nesnesinin dilimlenmesinde her boyut için dilimleme yapılabilir. Örneğin:

```

>>> a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> a[0:2, 1:3]
array([[2, 3],
       [5, 6]])
>>> a[:2, 1:]
array([[2, 3],
       [5, 6]])

```

Aşağıdaki alt matrisi elde etmek isteyelim:

$$\begin{matrix}
 1 & 2 & 3 \\
 4 & \boxed{5} & 6 \\
 7 & 8 & 9
 \end{matrix} \quad a[1:, 1:]$$

```

>>> a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> a[1:, 1:]
array([[5, 6],
       [8, 9]])

```

Çok boyutlu ndarray dizisinin dilimlenmesinden elde edilen ürün tek boyutlu dizi olabilir:

Örneğin:

```

>>> a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> a[1, :]
array([4, 5, 6])

```

Dilimleme yoluyla elde edilen ndarray nesnesine skaler bir değer atanabilir. Bu durumda bu değer dilimlenmiş tüm elemanlara atanmaktadır. Örneğin:

```

>>> a = np.array([10, 20, 30, 40, 50])
>>> a[1:4] = -1
>>> a
array([10, -1, -1, -1, 50])

```

Biz bir ndarray nesnesini dilimleyerek ona bir dolaşılabilir nesne yoluyla atama da yapabiliriz. Ancak bu durumda atanacak dolaşılabilir nesnenin eleman sayısı atamanın yapıldığı dilimin eleman sayısı ile aynı olmak zorundadır. Örneğin:

```
>>> a = np.array([10, 20, 30, 40, 50])
>>> a[1:4] = [1, 2, 3]
>>> a
array([10, 1, 2, 3, 50])
>>> a[1:4] = [1, 2]
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    a[1:4] = [1, 2]
ValueError: cannot copy sequence with size 2 to array axis with dimension 3
```

Atama sırasında kaynak nesnenin herhangi bir dolaşılabilir nesne olabileceğine dikkat ediniz. Örneğin:

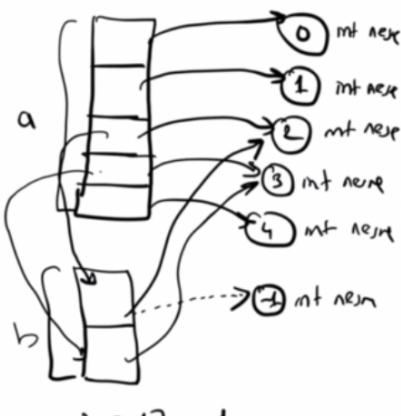
```
>>> a = np.array([10, 20, 30, 40, 50])
>>> a[1:4] = (1, 2, 3)
>>> a
array([10, 1, 2, 3, 50])
```

Dilimlerden Elde Edilen Alt Dizilerin Bir Görüntü (View) Belirtmesi Durumu

Bilindiği gibi listeler dilimlendiğinde aslında yeni bir liste nesnesi yaratılıp eski nesnedeki adresler bu yeni nesneye kopyalandı. Biz buna sağlam kopyalama (shallow copy) diyoruz. Bu nedenle dilimlemeden sonra değiştirilemez (immutable) nesnelere atama yapıldığında artık bundan asıl liste etkilenmemektedir. Örneğin:

$$a = [0, 1, 2, 3, 4]$$

$$b = a[2:4]$$



Halbuki ndarray nesnesinin dilimlenmesinde elde edilen ürün listelerdeki gibi değildir. Asıl dizinin bir görüntüsüdür. Yani biz dilimleme sonucıyla elde ettiğimiz nesnede değişiklik yaparsak her zaman asıl nesnede de değişiklik yapmış oluruz. Örneğin:

```
>>> a = np.array([0, 1, 2, 3, 4])
>>> a
array([0, 1, 2, 3, 4])
>>> b = a[2:4]
>>> b
array([2, 3])
>>> b[0] = -1
>>> a
array([0, 1, -1, 3, 4])
```

```

>>> b
array([-1,  3])
>>> a[3] = -1
>>> a
array([ 0,  1, -1, -1,  4])
>>> b
array([-1, -1])

```

Burada konuyu şöyle ele alabilirsiniz: Aslında biz ndarray dizisinde dilimleme yaptığımızda asıl dizinin o kısmını temsil eden bir dizi elde ederiz. Dilimleme sonucunda elde ettiğimiz dizi farklı bir dizi değildir. Asıl dizimin ilgili kısmını temsil eden bir dizidir. İşte buna asıl dizinin görüntüsü (view) denilmektedir. Başka bir deyişle ndarray sınıfını yazanlar aslında dilimleme sonucunda elde edilen ndarray nesnesini asıl diziye referans eden elemanlardan oluşturmuşlardır. Biz dilimlenmiş yeni nesne üzerinde işlem yaptığımızda aslında orijinal nesne üzerinde işlem yapılmaktadır. Dilimlenmiş nesnes yalnızca asıl nesnenin neresinin dilimlendiği bilgisini tutmaktadır. Aynı durum çok boyutlu ndarray dizileri için de aynı biçimde söz konusudur. Örneğin:

```

>>> a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> b = a[:2, :2]
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> b
array([[1, 2],
       [4, 5]])
>>> b[0, 0] = -1; b[0, 1] = -1; b[1, 0] = -1; b[1, 1] = -1
>>> a
array([[-1, -1,  3],
       [-1, -1,  6],
       [ 7,  8,  9]])
>>> b
array([[-1, -1],
       [-1, -1]])

```

Eğer biz dilimlemeden elde edilen ürünün bir görüntü (view) olmasını istemiyorsak global copy fonksiyonuyla ya da ndarray sınıfının copy metoduyla bilinçli biçimde kopya çıkartmalıyız. Örneğin:

```

>>> a = np.array([10, 20, 30, 40, 50])
>>> b = np.copy(a[1:4])
>>> a
array([10, 20, 30, 40, 50])
>>> b
array([20, 30, 40])
>>> b[0] = 100
>>> b
array([100, 30, 40])
>>> a
array([10, 20, 30, 40, 50])

```

Tabii eğer copy metodunu kullanacaksak artık buna bir argüman vermemiz. Örneğin:

```

>>> a = np.array([10, 20, 30, 40, 50])
>>> b = a[1:4].copy()
>>> a
array([10, 20, 30, 40, 50])
>>> b
array([20, 30, 40])

```

ndarray Nesnelerinin ndarray ya da liste Nesneleriyle ve Bool Türden Nesnelerle İndesklenmesi

Bir ndarray nesnesi bir ndarray ya da liste nesneyle indekslenebilir. Ancak bu indekslemeden bir görüntü (view) nesnesi elde edilmektedir. Örneğin:

```
>>> a = np.array([10, 20, 30, 40, 50])
>>> b = [2, 4, 1]
>>> c = a[b]
>>> a
array([10, 20, 30, 40, 50])
>>> c
array([30, 50, 20])
>>> a[0] = 100
>>> a
array([100, 20, 30, 40, 50])
>>> c
array([30, 50, 20])
```

Burada da görüldüğü gibi indekslemede bir görüntü nesnesi elde edilmemiştir. Bu tür indekslemeler gerçekten birlığı kopyalamaya yol açmaktadır. Eğer bir ndarray nesnesi Bool türden bir ndarray nesnesi ya da liste nesnesi ile indekslenirse bu durumda indeks nesnesinde True olan elemanlar elde edilir. Örneğin:

```
>>> a = np.array([10, 20, 30, 40, 50])
>>> b = a[[True, False, False, True, True]]
>>> a
array([10, 20, 30, 40, 50])
>>> b
array([10, 40, 50])
```

Bool indekslemede indeks dizisinin uzunluğunun indekslenecek dizi uzunluğuyla aynı olması gereklidir. Örneğin:

```
>>> a = np.array([10, 20, 30, 40, 50])
>>> b = a[[True, False]]
Traceback (most recent call last):
  File "<pyshell#115>", line 1, in <module>
    b = a[[True, False]]
IndexError: boolean index did not match indexed array along dimension 0; dimension is 5 but
corresponding boolean dimension is 2
```

Yine Bool türden indekslemelerde elde edilen ürün bir görüntü (view) değildir.

ndarray Nesnelerinin Boyutlarının Değiştirilmesi

Bir ndarray nesnesinin boyutları global reshape fonksiyonuyla ya da ndarray sınıfının reshape metoduya değiştirilebilir. Ancak her türlü boyut değiştirmeler görüntüsüz (view) biçimde gerçekleştirilmektedir. reshape fonksiyonunun parametrik yapısı şöyledir:

```
numpy.reshape(a, newshape, order='C')
```

Fonksiyonun birinci parametresi diziyi, ikinci parametresi bir demet olarak boyutu (demet yerine skaler kullanılırsa tek boyut anlaşılmaktadır), üçüncü parametresi ise içsel yerleşimi belirtmektedir. Örneğin:

```
>>> a = np.arange(12)
>>> b = np.reshape(a, (4, 3))
>>> a
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> b
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
>>> b[1, 1] = -1
```

```

>>> a
array([ 0,  1,  2,  3, -1,  5,  6,  7,  8,  9, 10, 11])
>>> b
array([[ 0,  1,  2],
       [ 3, -1,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])

```

Aynı işlemi biz ndarray sınıfının shape metoduyla da yapabiliyoruz. Örneğin:

```

>>> a = np.arange(12)
>>> b = a.reshape((4, 3))
>>> a
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> b
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])

```

ndarray sınıfının flatten isimli metodu (bunun karşılığı olarak bir global fonksiyon yoktur) çok boyutlu diziyi görüntüsель olarak tek boyuta indirger. Örneğin:

```

>>> a = np.array([[10, 20], [30, 40], [50, 60]])
>>> b = a.flatten()
>>> a
array([[10, 20],
       [30, 40],
       [50, 60]])
>>> b
array([10, 20, 30, 40, 50, 60])

```

Global transpose isimli ve ndarray sınıfının transpose isimli metodu görüntüsель biçimde transpose matrisi verir. Örneğin:

```

>>> a = np.array([[10, 20], [30, 40], [50, 60]])
>>> a
array([[10, 20],
       [30, 40],
       [50, 60]])
>>> b = np.transpose(a)
>>> b
array([[10, 30, 50],
       [20, 40, 60]])
>>> c = a.transpose()
>>> c
array([[10, 30, 50],
       [20, 40, 60]])

```

Tek boyutlu ndarray nesneleri transpose edildiğinde bir değişiklik oluşmamaktadır.

Global vstack fonksiyonu bizden bir demet biçiminde satır vektörlerini (yani 1xn'lik matris) alarak iki boyutlu matris oluşturmaktadır. Örneğin:

```

>>> a = np.vstack(([1, 2, 3], [4, 5, 6], [7, 8, 9]))
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

```

Görüldüğü gibi fonksiyon satır vektörlerini tek boyutlu dolaşılabilir nesneler olarak alıp bize matrisel bir ndarray nesnesi vermektedir. hstack fonksiyonu tam ters işlemi yapmaktadır. Ancak burada vektörlerin sütun matrisi (yani nx1'lik bir matris) olması gereklidir. Örneğin:

```
>>> a = np.hstack((x, y, z))
>>> a
array([[1, 7, 7],
       [2, 8, 8],
       [3, 9, 9]])
>>> x = np.array([[1], [2], [3]])
>>> y = np.array([[4], [5], [6]])
>>> z = np.array([[7], [8], [9]])
>>> a = np.hstack((x, y, z))
>>> a
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
```

Sütun vektörlerinin bu biçimde oluşturulması size biraz zahmetli gelebilir. Bunu kolaylaştırmak için reshape fonksiyonunu ya da metodunu uygulayabilirsiniz. Örneğin:

```
>>> x = np.array([1, 2, 3]).reshape(3, 1)
>>> y = np.array([4, 5, 6]).reshape(3, 1)
>>> z = np.array([7, 8, 9]).reshape(3, 1)
>>> x
array([[1],
       [2],
       [3]])
>>> y
array([[4],
       [5],
       [6]])
>>> z
array([[7],
       [8],
       [9]])
>>> a = np.hstack((x, y, z))
>>> a
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
```

expand_dims isimli global fonksiyon diziye yeni bir boyut katmak için kullanılmaktadır. Örneğin:

```
>>> a = np.array([1, 2, 3])
>>> b = np.expand_dims(a, axis=1)
>>> a
array([1, 2, 3])
>>> b
array([[1],
       [2],
       [3]])
>>> c = np.expand_dims(a, axis=0)
>>> c
array([[1, 2, 3]])
```

Buradaki axis parametresi 0 için satır vektörü 1 için sütun vektörü oluşturma anlamındadır.

Global insert isimli fonksiyon ndarray nesnesine insert işlemi yapar. Ancak bu fonksiyon nesnenin kendisine değil yeni oluşturduğu kopyaya işlemi yapmaktadır. Fonksiyonun parametrik yapısı şöyledir:

```
numpy.insert(arr, obj, values, axis=None)
```

Fonksiyonun birinci parametresi insert işlemine konu olan diziyi, ikinci paarametresi insert pozisyonunu üçüncü parametresi de insert edilecek değeri belirtir. Son parametre insert edilecek boyutu belirtmektedir. Ayrıca sınıfın insert isimli bir metod yoktur. insert işlemi yalnızca global insert fonksiyonuya yapılmaktadır. Bu metot bize yeni insert edilmiş yeni bir dizi vermektedir. Bu dizinin default boyutu 1xn biçimindedir. Örneğin:

```
>>> a = np.array([0, 1, 2, 3, 4])
>>> b = np.insert(a, 3, 100)
>>> a
array([0, 1, 2, 3, 4])
>>> b
array([ 0,  1,  2, 100,  3,  4])
```

Fonksiyonun bir görüntü dizisi vermediğine kopyalanmış yeni bir dizi oluşturduğuna dikkat ediniz. Çok boyutlu dizilere insert işlemi yapılırken default olarak tek boyutlu bir dizi elde edilir. Örneğin:

```
>>> a = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
>>> b = np.insert(a, 5, 100)
>>> a
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> b
array([ 0,  1,  2,  3,  4, 100,  5,  6,  7,  8])
```

Fakat istersek axis parametresini kullanabiliriz. Örneğin:

```
>>> b = np.insert(a, 1, 100, axis=1)
>>> a
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> b
array([[ 0, 100,  1,  2],
       [ 3, 100,  4,  5],
       [ 6, 100,  7,  8]])
```

Gördüğü gibi tamamaen yeni bir sütun oluşturulup insert edilecek değer o sütuna doldurulmuştur.

delete isimli global fonksiyon insert işleminin tersini yapmaktadır. Yani diziden bir değeri silerek yeni bir dizi verir. Parametrik yapısı şöyledir:

```
numpy.delete(arr, obj, axis=None)
```

Fonksiyonun birinci parametresi delete işlemine konu olan diziyi, ikinci parametresi delete edilecek indeksi belirtmektedir. Üçüncü parametre belli bir satırın ya da sütunun delete edilmesi için kullanılır. Bu fonksiyonun da ndarray metoduna olar bir karşılığı yoktur. Örneğin:

```
>>> a = np.vstack(([0, 1, 2], [3, 4, 5], [6, 7, 8]))
>>> b = np.delete(a, 4)
>>> a
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> b
array([0, 1, 2, 3, 5, 6, 7, 8])
```

Global append isimli fonksiyon bir dizinin sonuna yeni bir eleman eklemek için kullanılmaktadır. Ancak ekleme mevcut diziye yapılmaz yeni biri dizi yaratılıp oraya yapılır. Fonksiyon bu yeni diziyle geri dönmektedir. append fonksiyonun da bir metot karşılığı yoktur. Örneğin:

```
numpy.append(arr, values, axis=None)
```

Fonksiyonun birinci parametresi söz konusu diziyi, ikinci parametresi eklenecek değeri belirtir. Örneğin:

```
>>> a = np.array([0, 1, 2, 3, 5])
>>> b = np.append(a, 100)
>>> a
array([0, 1, 2, 3, 5])
>>> b
array([ 0,  1,  2,  3,  5, 100])
```

ndarray nesnesinin kendisi üzerinde insert, delete ve append yapılmadığına yeni bir dizi yaratılarak bu işlemlerin yapılabildiğine dikkat ediniz. Bunun nedeni bir dizinin çeşitli görüntülerinin (view) oluşturulabilmesindendir. Çünkü asıl dizide değişiklik yapıldığında görüntü dizisinin güncellenmesi algoritmik bir sorundur.

ndarray Nesneleri Üzerinde Vektörel İşlemler

Biz şimdije kadar hep ndarray nesnelerinin oluşturulması ve işlenmesi üzerinde durduk. Aslında ndarray nesneleri birtakım aritmetik, mantıksal ve fonksiyonel işlemlere sokulmaktadır. Ancak bu işlemler vektörel biçimde yani dizinin her elemanı karşılıklı olacak biçimde yapılmaktadır. Matlab gibi, R gibi dillerde de temel çalışma biçimi böyledir.

Biz iki ndarray nesnesini aritmetik işlemlere soktuğumuzda dizinin karşılıklı elemanları işleme sokulur ve sonuç bir dizi biçiminde elde edilir. Örneğin:

```
>>> a = np.array([1, 2, 3, 4, 5])
>>> b = np.array([10, 20, 30, 40, 50])
>>> c = a + b
>>> a
array([1, 2, 3, 4, 5])
>>> b
array([10, 20, 30, 40, 50])
>>> c
array([11, 22, 33, 44, 55])
>>> c = a * b
>>> c
array([ 10,   40,   90,  160, 250])
>>> c = b ** a
>>> c
array([ 10,      400,    27000,  2560000, 312500000], dtype=int32)
>>> c = a / b
>>> c
array([0.1, 0.1, 0.1, 0.1, 0.1])
>>> c = a // b
>>> c
array([0, 0, 0, 0, 0], dtype=int32)
```

Yine +=, -=, *=, /= gibi operatörler asıl nesne üzerinde değişiklik yaparlar. Yani:

```
a += b
```

ile

```
a = a + b
```

listelerde olduğu gibi tamamen aynı anlama gelmemektedir. $a += b$ işleminde değişiklik mevcut a dizisi üzerinde yapılmaktadır. Halbuki $a = a + b$ işleminde yeni bir dizi yaratılmakta ve a artık bu yeni diziyi göstermektedir.

Farklı uzunluklardaki aynı boyutlu diziler üzerinde işlemler yapılamamaktadır. Örneğin:

```
>>> a = np.array([1, 2, 3, 4, 5])
>>> b = np.array([10, 20, 30])
>>> c = a + b
Traceback (most recent call last):
  File "<pyshell#70>", line 1, in <module>
    c = a + b
ValueError: operands could not be broadcast together with shapes (5,) (3,)
```

Bir dizi ile bir skaler de işleme sokulabilir. Bu durumda skaler dizinin her elemanıyla işleme sokulmuş olur. Bu işlemlerden ürün olarak yine bir dizi elde edilmektedir. Örneğin:

```
>>> a = np.array([1, 2, 3, 4, 5])
>>> b = a * 2
>>> a
array([1, 2, 3, 4, 5])
>>> b
array([ 2,  4,  6,  8, 10])
```

Örneğin eleimizde çeşitli x değerleri olsun. Biz bu x değerlerini $x^{**} 2 - 3$ işlemine sokup y değerlerini elde etmek isteyelim. Vektörel işlemler bu gibi durumlarda çok pratiklik sağlamaktadır:

```
>>> x = np.array([1.2, 4, 5.6, 6, 8.2])
>>> y = x ** 2 - 3
>>> x
array([1.2, 4. , 5.6, 6. , 8.2])
>>> y
array([-1.56, 13. , 28.36, 33. , 64.24])
```

Çok boyutlu dizilerde işlem yapıldıktan sonra (broadcasting) özelliği vardır. Örneğin bir matris ile onun bir satır ya da sütunu biçiminde tek boyutlu bir dizi işleme sokulursa işlem matrisin her satırı ya da her sütunu üzerinde yapılmalıdır. Örneğin:

```
>>> a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> b = np.array([10, 20, 30])
>>> c = a + b
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> b
array([10, 20, 30])
>>> c
array([[11, 22, 33],
       [14, 25, 36],
       [17, 28, 39]])
>>> c = a * b
>>> c
array([[ 10,  40,  90],
       [ 40, 100, 180],
       [ 70, 160, 270]])
```

Burada 3×3 'lik bir matris ile 1×3 'lük bir dizi işleme sokulmuştur. İşlem matrisin her satırı üzerinde yapılmıştır. Yani bir deyişle yukarıdaki işlemin eşdeğeri şöyledir:

```
>>> a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

```
>>> b = np.array([[10, 20, 30], [10, 20, 30], [10, 20, 30]])
>>> c = a + b
>>> c
array([[11, 22, 33],
       [14, 25, 36],
       [17, 28, 39]])
```

numpy modülündeki çeşitli fonksiyonlar bizden bir ndarray’ı alıp onun her elemanı üzerinde işlem yapıp ürün olarak bize yine bir dizi verirler. Örneğin sin, cos, tan, arcsin, arccos, arctan fonksiyonları trigonometrik işlemleri yapmaktadır. Örneğin:

```
>>> x = np.arange(0, 3.14, 0.1)
>>> x
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. , 1.1, 1.2,
       1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2. , 2.1, 2.2, 2.3, 2.4, 2.5,
       2.6, 2.7, 2.8, 2.9, 3. , 3.1])
>>> y = np.sin(x)
>>> y
array([0.          , 0.09983342, 0.19866933, 0.29552021, 0.38941834,
       0.47942554, 0.56464247, 0.64421769, 0.71735609, 0.78332691,
       0.84147098, 0.89120736, 0.93203909, 0.96355819, 0.98544973,
       0.99749499, 0.9995736 , 0.99166481, 0.97384763, 0.94630009,
       0.90929743, 0.86320937, 0.8084964 , 0.74570521, 0.67546318,
       0.59847214, 0.51550137, 0.42737988, 0.33498815, 0.23924933,
       0.14112001, 0.04158066])
>>> y = np.cos(x)
>>> y
array([-1.          , 0.99500417, 0.98006658, 0.95533649, 0.92106099,
       0.87758256, 0.82533561, 0.76484219, 0.69670671, 0.62160997,
       0.54030231, 0.45359612, 0.36235775, 0.26749883, 0.16996714,
       0.0707372 , -0.02919952, -0.12884449, -0.22720209, -0.32328957,
      -0.41614684, -0.5048461 , -0.58850112, -0.66627602, -0.73739372,
      -0.80114362, -0.85688875, -0.90407214, -0.94222234, -0.97095817,
      -0.9899925 , -0.99913515])
```

sqrt, exp, log, log2 ve log10 fonksiyonları da klasik üstel işlemleri yapmaktadır. Örneğin:

```
>>> a = np.arange(10)
>>> b = np.sqrt(a)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b
array([0.          , 1.          , 1.41421356, 1.73205081, 2.          ,
       2.23606798, 2.44948974, 2.64575131, 2.82842712, 3.          ])
```

Bunların dışında daha pek çok vektörel işlem yapan numpy fonksiyonu vardır. Örneğin abs, floor, ceil, round gibi. Bunları numpy dokümanlarından inceleyebilirsiniz.

Bazı fonksiyonlar bizden bir ndarray’ı alıp skaler değer vermektedir. Örneğin sum fonksiyonu ndarray’ın nesnesindeki değerleri toplayıp bize bunun toplamına ilişkin skaler bir değer verir. Örneğin:

```
>>> a = np.arange(1, 101)
>>> total = sum(a)
>>> total
5050
```

Örneğin biz tek halede std ve var fonksiyonlarıyla bir grup sayının standart sapmasını ve varyansını elde edebiliriz:

```
>>> a = np.array([3, 5, 2, 8, 9])
>>> a
```

```
array([3, 5, 2, 8, 9])
>>> sdev = np.std(a)
>>> sdev
2.727636339397171
```

mean fonksiyonu bir ndarray nesnesinin ortalamasını bize verir. Bu durumda örneğin biz standart sapmayı şöyle de elde edebilirdik:

```
>>> sdev = np.sqrt(np.sum((a - np.mean(a)) ** 2)/(a.size))
>>> sdev
2.727636339397171
```

prod isimli fonksiyon bir ndarray içerisindeki tüm değerlerin çarpımını bize verir. Örneğin 10 faktöryel değerini hesaplamak isteyelim:

```
>>> a = np.arange(1, 11)
>>> a
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
>>> b = np.prod(a)
>>> b
3628800
```

min ve max fonksiyonları dizinin en küçük ve en büyük elemanlarını bize verir. Örneğin:

```
>>> a = np.array([3, 6, 34, 32, 17])
>>> min = np.min(a)
>>> max = np.max(a)
>>> min
3
>>> max
34
```

Tüm bu fonksiyonlar aslında çok boyutlu dizilerle de işleme sokulabilmektedir. Ancak bunlar çok boyutları dizileri sanki tek boyutlu olmuş gibi ele alırlar. Yani biz bir matrisin toplamını da sum fonksiyonuyla elde edebiliriz.

all fonksiyonu bir dizideki tüm elemanlar True ise (hangi değerlerin True olarak ele alındığını biliyorsunuz) any fonksiyonu herhangi bir değer True ise True değerine değilse False değerine geri döner. Bu fonksiyonların geri dönüş değeri 0 ya da sıfır dışı bir değer olabilir. Örneğin:

```
>>> a = np.array([12.3, 0, True, None])
>>> np.all(a)
0
>>> np.any(a)
12.3
```

NumPy ile Lineer Cebir İşlemleri

Lineer cebir işlemleri için hem numpy modülündeki hem de numpy.linalg modülündeki fonksiyonlar kullanılabilmektedir. numpy modülündeki dot isimli fonksiyon matris çarpımı yapar. Yani matrisin karşılıklı elemanlarını değil matemetikteki matris çarpımını gerçekleştirer. Örneğin aşağıdaki iki matrisi çarpmak isteyelim:

$$\begin{bmatrix} 1 & 5 & 4 \\ 3 & 8 & 2 \\ 7 & 10 & 3 \end{bmatrix} \begin{bmatrix} 3 \\ 6 \\ 1 \end{bmatrix} = \begin{bmatrix} \dots \\ b \end{bmatrix}$$

a *b*
numpy.dot(a, b)

```
>>> a = np.array([[1, 5, 4], [3, 8, 2], [7, 10, 3]])
>>> b = np.array([[3], [6], [1]])
>>> c = np.dot(a, b)
>>> c
array([[37],
       [59],
       [84]])
```

Ters matris almak için linalg modülündeki inv fonksiyonu kullanılmaktadır. Örneğin yukarıdaki matrisin tersini alacak olalım:

```
>>> a = np.array([[1, 5, 4], [3, 8, 2], [7, 10, 3]])
>>> np.linalg.inv(a)
array([[-0.05333333, -0.33333333,  0.29333333],
       [-0.06666667,  0.33333333, -0.13333333],
       [ 0.34666667, -0.33333333,  0.09333333]])
```

Şimdi aşağıdaki lineer denklem sistemini çözmeye çalışalım:

$$\begin{aligned} 3x_1 + x_2 - 2x_3 &= 3 \\ x_1 + x_2 + x_3 &= 2 \\ -2x_1 + 2x_2 - x_3 &= -7 \end{aligned}$$

Bunu matrisel olarak şöyle ifade edebiliriz:

$$\begin{aligned} 3x_1 + x_2 - 2x_3 &= 3 \\ x_1 + x_2 + x_3 &= 2 \\ -2x_1 + 2x_2 - x_3 &= -7 \end{aligned}$$

$\begin{bmatrix} 3 & 1 & -2 \\ 1 & 1 & 1 \\ -2 & 2 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \\ -7 \end{bmatrix}$

İki tarafı ters matris çarparsak çözümü elde deriz. Örneğin:

```
>>> a = np.array([[3, 1, -2], [1, 1, 1], [-2, 2, -1]])
>>> b = [[3], [2], [-7]]
>>> np.dot(np.linalg.inv(a), b)
array([[ 2.],
       [-1.],
       [ 1.]])
```

Göründüğü gibi çözüm $x_1 = 2$, $x_2 = -1$ ve $x_3 = 1$ biçiminde bulunmuştur. Aslında bu işlemi tek hamlede yapan numpy.linalg.solve fonksiyonu vardır. Örneğin:

```
>>> a = np.array([[3, 1, -2], [1, 1, 1], [-2, 2, -1]])
```

```
>>> b = [[3], [2], [-7]]
>>> np.linalg.solve(a, b)
array([[ 2.],
       [-1.],
       [ 1.]])
```

Determinant hesabı için yine numpy.linalg.det fonksiyonu kullanılmaktadır. Örneğin:

```
>>> a = np.array([[1, 5, 4], [3, 8, 2], [7, 10, 3]])
>>> np.linalg.det(a)
-74.999999999997
```

Verilerin Dosyalarından Okunması

Nümerik analizlerde verilerin kod içerisinde oluşturulmasıyla seyrek karşılaşılmaktadır. Aslında analiz edilecek edilecek veriler ya dosyalardan okunmakta ya da başka bir ortamdan (örneğin bir veritabanı, web servis, soket gibi) alınmaktadır. Tabii en yaygın karşılaşılan durum verilerin bir text dosyadan alınmasıdır. Text dosyalarda veriler ya tablosal biçimde yani satır satır ya da düz biçimde bulunmaktadır. Tablosal biçimde veriler dosyalarda genellikle aralarına ',' karakteri ya da TAB karakter getirilerek satır satır bulundurulmaktadır. Bilindiği gibi aralarına ',' karakteri getirilerek tablosal biçimde oluşturulmuş olan dosyalara CSV dosyaları denilmektedir.

Python'da normal dosyalardan ve CSV dosyalarından okuma yapmak için çeşitli fonksiyonlar zaten bulunmaktadır. Örneğin standart kütüphanedeki csv modülünde reader isimli fonksiyon bizden bir dolaşılabilir nesneyi alır onu CSV olarak okur. Her satırı bize bir liste olarak verir. Dosya nesnelerinin de dolaşılabilir nesneler olduğunu anımsayınız. Örneğin:

```
import csv

with open('test.txt') as file:
    reader = csv.reader(file)
    for l in reader:
        print(l)
```

Yukarıdaki programın örnek çıktısı şöyledir:

```
['10', ' 20', ' 30']
['40', ' 50', ' 60']
```

reader fonksiyonun aldığı dolaşılabilir nesne dolaşıldığından bize CSV satırları vermesi gerekmektedir.

Ancak Python'ın standart csv.reader fonksiyonu NumPy için kullanışlı değildir. Çünkü biz NumPy'da okuduğumuz değerleri bir string listesi biçiminde değil bir ndarray nesnesi biçiminde elde etmek isteriz. Bunun için NumPy modülünde ayrı fonksiyonlar bulundurulmuştur.

numpy.loadtxt fonksiyonu CSV tarzı text dosyalarını okuyarak bize içeriğini ndarray olarak vermektedir. Fonksiyonun parametrik yapısı şöyledir:

```
numpy.loadtxt(fname, dtype=<class 'float'>, comments='#', delimiter=None, converters=None,
skiprows=0, usecols=None, unpack=False, ndmin=0, encoding='bytes')
```

Default durumda fonksiyon boşluk karakterlerini ve New Line karakterlerini ayıraç olarak almaktadır. Fonksiyon bize tablosal biçimde verileri bir matris olarak verir. Örneğin:

```
import numpy as np

result = np.loadtxt('test.txt')
print(result)
```

Buradaki test.txt dosyasının içeriği şöyle olsun:

```
10 20 30  
40 50 60
```

Çıktı şöyle olacaktır:

```
[[10. 20. 30.]  
 [40. 50. 60.]]
```

Default dtype'in float64 olduğuna dikkat ediniz. Tabii bu dtype parametresi kullanılarak değiştirilebilir. Örneğin:

```
import numpy as np  
  
result = np.loadtxt('test.txt', dtype=np.int32)  
print(result)
```

CSV okumaları için delimiter parametresini ',' biçiminde girmek gereklidir. (Default durumda yukarıda da belirtildiği gibi delimiter bir ya da biren fazla boşluk karakterleri biçimindedir.) Örneğin:

```
import numpy as np  
  
result = np.loadtxt('test.txt', delimiter=',', dtype=np.int32)  
print(result)
```

Buradaki test.txt dosyası şöyledir:

```
10,20,30,40  
50,60,70,80  
90,100,110,120
```

Çıktı da şöyle elde edilecektir:

```
[[ 10   20   30   40]  
 [ 50   60   70   80]  
 [ 90  100  110  120]]
```

Burada ',' karakterlerinin solunda ve sağında yine istenildiği kadar SPACE ve TAB karakteri bulunabilmektedir.

Örneğin biz iki a.txt ve b.txt dosyalarında bulunan iki matrisi çarpmak isteyelim. İşlemi aşağıdaki gibi yapabiliriz:

```
import numpy as np  
  
a = np.loadtxt('a.txt')  
b = np.loadtxt('b.txt')  
c = np.dot(a, b)  
print(c)
```

Bazen tablosal bir dosyada yalnızca belirli sütunları elde etmek isteyebiliriz. Bunun için fonksiyonun usecols parametresi kullanılır. Bu parametre bir demet biçiminde girilmelidir. Örneğin:

```
import numpy as np  
  
result = np.loadtxt('test.txt', delimiter=',', usecols=(1, 2))  
print(result)
```

Buradaki test.txt dosyasının içeriği de şöyledir:

Ali Serçe, 10, 20, 30
Veli Akar, 40, 50, 60
Selami Paydaş, 70, 80, 90

Ekran çıktısı da şöyle olacaktır:

```
[[10. 20.]  
 [40. 50.]  
 [70. 80.]]
```

genfromtxt fonksiyonu da loadtxt fonksiyonuyla çok benzerdir. Ancak loadtxt fonksiyonundan daha yeteneklidir. Parametrik yapısı oldukça genişştir:

```
numpy.genfromtxt(fname, dtype=<class 'float'>, comments='#', delimiter=None, skip_header=0,  
skip_footer=0, converters=None, missing_values=None, filling_values=None, usecols=None,  
names=None, excludelist=None, deletechars=None, replace_space='_', autostrip=False,  
case_sensitive=True, defaultfmt='f%i', unpack=None, usemask=False, loose=True,  
invalid_raise=True, max_rows=None, encoding='bytes')
```

Örneğin:

```
import numpy as np  
  
result = np.genfromtxt('test.txt', delimiter=',', usecols=(1, 2), skip_header=1,  
dtype=np.int32, filling_values=-10)  
print(result)
```

test.txt dosyasının içeriği şöyledir:

```
Adı Soyadı No1 No2 No3  
Ali Serçe, 10,, 30  
Veli Akar, 40, 50, 60  
Selami Paydaş, 70, 80, 90
```

Ekran çıktısı da şöyledir:

```
[[ 10 -10]  
 [ 40  50]  
 [ 70  80]]
```

Sembolik Matematiksel İşlemler ve SymPy Paketi

NumPy Paketi sayısal değerleri üzerinde işlemler yapmaktadır. Ancak matematikte simbolîlî değerleri üzerinde işlemler de çok sık yapılmaktadır. Yani örneğin biz belli bir aralıktâ belirli integrali numpy ve pandas kullanarak hesaplayabiliriz. Ancak simbolik işlemleri bu paketlerle yapamayız. Sembolik işlemler için SymPy paketi kullanılmaktadır. SymPy diğer paketlerde olduğu gibi indirilip kurulmaktadır.

SymPy paketinin dokümantasyonuna aşağıdaki URL'den erişilebilir:

<https://www.sympy.org/en/docs.html>

SymPy işlemlerinde biz genellikle paketi aşağıdaki gibi import edeceğiz:

```
import sympy as sp
```

SymPy'da ifadelerin daha matematiksel bir biçimde görüntülenmesi isteniyorsa işin başında bir kez `sympy.init_printing()` fonksiyonunun çağrılması gerekir.

SymPy işlemlerinde `x`, `y`, `z` gibi değerlere sembol denilmektedir. Bir sembol herhangi bir değeri alan bir sembolik bir değişkendir. Semboller birkaç biçimde yaratılabilirmektedir. En çok kullanılan yöntem `Symbol` sınıfının başlangıç metodu olan `Symbol` fonksiyonunu kullanmaktadır. Bu fonksiyon bizden sembolün ismini string olarak alır ve `Symbol` sınıfı türünden bir nesne verir. Örneğin:

```
>>> x = sp.Symbol('x')
>>> x
x
>>> type(x)
<class 'sympy.core.symbol.Symbol'>
```

Bir sembol yaratılırken onun hakkında bazı bilgiler varsa o bilgiler de sembole ilişirilebilir. Sonra istenirse `Symbol` sınıfının örnek öznitelikleriyle bu bilgilerin olup olmadığı elde edilebilir. Sembole ilişirilecek bilgiler şunlardır:

Parametre	Test Özniteliği
<code>real</code>	<code>is_real</code>
<code>imaginary</code>	<code>is_imaginary</code>
<code>positive</code>	<code>is_positive</code>
<code>negative</code>	<code>is_negative</code>
<code>odd</code>	<code>is_odd</code>
<code>even</code>	<code>is_even</code>
<code>prime</code>	<code>is_prime</code>
<code>finite</code>	<code>is_finite</code>
<code>infinite</code>	<code>is_infinite</code>

İlgili özellik `Symbol` fonksiyonunda aynı isimli bool parametre ile belirtilmektedir. Örneğin:

```
>>> x = sp.Symbol('x', real=True)
>>> x
x
>>> type(x)
<class 'sympy.core.symbol.Symbol'>
>>> x.is_real
True
```

Örneğin:

```
>>> y = sp.Symbol('y', positive=True)
>>> y
y
>>> type(y)
<class 'sympy.core.symbol.Symbol'>
>>> y.is_positive
True
```

Çok sayıda sembol hızlı bir biçimde yaparılacaksa `symbols` isimli fonksiyon tercih edilebilir. Bu fonksiyon bizden sembollerin isimlerini virgüllerle ayrılmış tek bir yazı biçiminde ister. Bu yazılımı parse ederek semboller oluşturur ve onları bize bir demet biçiminde verir. Örneğin:

```
>>> sp.symbols('x, y, z, t')
(x, y, z, t)
```

Burada `symbols` fonksiyonu `x`, `y`, `z` ve `t` sembollerini oluşturup bize bunu bir demet biçimde vermiştir. Yani yukarıdaki işlemin eşdeğeri şöyle yazılabilir:

```
(sp.Symbol('x'), sp.Symbol('y'), sp.Symbol('z'), sp.Symbol('t'))  
(x, y, z, t)
```

Tabii biz aynı zamanda açım (unwrap) yapabiliriz. Örneğin:

```
>>> x, y, z, t = sp.symbols('x, y, z, t')
```

symbols fonksiyonunda biz yine özellik belirtebiliriz. Ancak bunlar tüm semboller için geçerli olur. Örneğin:

```
>>> x, y, z, t = sp.symbols('x, y, z, t', positive=True)  
>>> x.is_positive  
True  
>>> y.is_positive  
True
```

Yalnızca Semboller değil sayısal değerler de SYmPy'da özel sınıflarla temsil edilmektedir. SymPy'in sayısal sınıfları Python'ın orijinal sınıflarına benzemekte birlikte sembolik işlemler için özelleştirilmiştir. Integer isimli sınıf tamsayı değerler tutar. Yine bu sınıfın da tuttuğu tamsayı değerlerin bir sınırı yoktur. Örneğin:

```
>>> x = sp.Integer(100)  
>>> type(x)  
<class 'sympy.core.numbers.Integer'>  
>>> x  
100
```

Benzer biçimde Float sınıfı da noktalı sayısal değerleri tutmak için düşünülmüştür. Örneğin:

```
>>> a = sp.Float(12.3)  
>>> type(a)  
<class 'sympy.core.numbers.Float'>  
>>> a  
12.3000000000000
```

sympy.Float sınıfı Python'ın standart Float sınıfından farklı olarak değerleri IEEE754 formatına göre tutmaz. Dolayısıyla biz istediğimiz duyarlılıkta sayıları bu sınıfa tutturabiliriz. Tabii noktalı sayılar Python'da Float türden olduğuna göre artık bu tür sayıları bizim sınıfa string olarak vermemiz gereklidir. Örneğin:

```
>>> a = sp.Float('12.234567899003445555555333')  
>>> a  
12.2345678990034455555555333
```

Bu biçimde sp.Float sınıfının sayıları yuvarlama hatası olmadan tuttuğuna dikkat ediniz. İstersek biz Float sınıfına ikinci parametre vererek belli bir duyarlılıkta sayıyı saklamasını siteyebiliriz. Örneğin:

```
>>> a = sp.Float('12.234567899003445555555333')  
>>> b = sp.Float(a, 10)  
>>> a  
12.2345678990034455555555333  
>>> b  
12.23456790
```

SymPy'da bazı özel değerler sembolik biçimde kullanılabilmektedir. Örneğin pi sayısı sembolik biçimde sympy.pi biçiminde, e sayısı sympy.E biçiminde kullanılabilir. Sonsuz değeri ise sembolik olarak sympy.oo ile kullanılır.

SimPy'da İfadeler (Expressions)

SymPy'daki ifadeler sembollerin, sabit değerlerin operatörlerle birleşiminden elde edilmektedir. Örneğin:

```

>>> x = sp.Symbol('x')
>>> y = 2 * x ** 2 - 3 * x - 5
>>> y
2*x**2 - 3*x - 5
>>> print(type(y))
<class 'sympy.core.add.Add'>

```

Burada $2x^2-3x-5$ biçiminde bir ifade oluşturulmuştur. Bu ifadenin türüne bakıldığından Add sınıfı bir sınıf türünde olduğu görülmektedir. Add sınıfı aslında ifadelerin toplanması ve çıkartılmasını temsil eden bir sınıfıdır. Örneğin:

```

>>> y = 2 * x
>>> y
2*x
>>> print(type(y))
<class 'sympy.core.mul.Mul'>

```

Burada da ifadenin Mul isimli bir sınıfta temsil edildiğini görüyorsunuz. Bir ifadenin terimleri args property'si ile bir demet biçiminde elde edilebilir. Örneğin:

```

>>> y = 2 * x
>>> y.args
(2, x)

```

Örneğin:

```

>>> y = 2 * x ** 2 - 3 * x - 5
>>> y.args
(-5, -3*x, 2*x**2)
>>> y.args[2].args
(2, x**2)
>>> y.args[2].args[1].args
(x, 2)

```

Bu örneklerden ne anlaşılmaktadır? SymPy'da ifadeler (expressions) bir ağaç oluşturmaktadır. İfadeler içerisindeki operatörler ve sabitler ayrı birer sınıfta temsil edilmektedir. Bu sınıflar bir ifade ağaçını oluşturmaktadır. Örneğin:

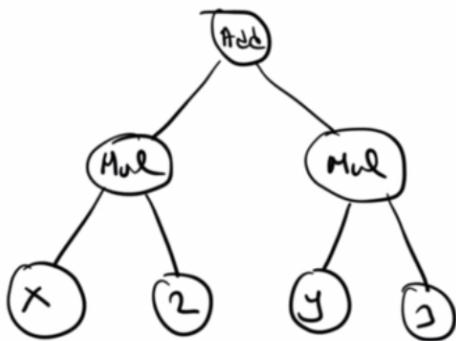
```

>>> x, y = sp.symbols('x, y')
>>> e = x * 2 + y * 3
>>> e
2*x + 3*y
>>> type(e)
<class 'sympy.core.add.Add'>
>>> e.args
(2*x, 3*y)
>>> e.args[0]
2*x
>>> type(e.args[0])
<class 'sympy.core.mul.Mul'>
>>> type(e.args[1])
<class 'sympy.core.mul.Mul'>

```

Buradaki ağaç aşağıdaki gibi ifade edilmiştir:

$$e = x^2 + y^3$$



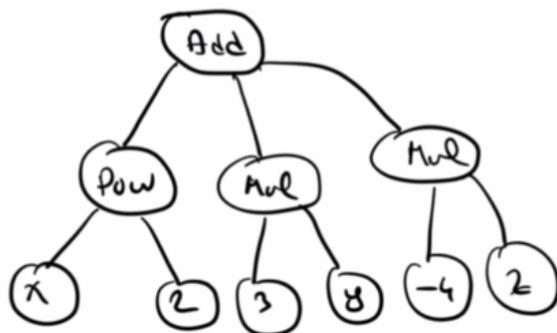
Örneğin:

```

>>> x, y, z = sp.symbols('x, y, z')
>>> e = x**2 + 3*y - 4*z
>>> e
3*y + z**2 - 4*z
  
```

Bu ifadenin ağacı da şöyledir:

$$e = x^2 + 3y - 4z$$



SymPy'da yukarıdaki ifade sınıfları değiştirilemez (immutable) sınıflardır. Yani örneğin bir Add, Mul, Pow nesnesi yaratıldıkten sonra artık içerik bakımından değiştirilemez.

Bir ifadeyi oluşturan Add, Pow, Mul gibi sınıfların hepsi Aslında Expr isimli bir sınıfından türetilmiştir.

İfadelerin Sadeleştirilmesi (Simplify)

Karmaşık ifadeler sympy.simplify fonksiyonuyla ya da ifade sınıflarının simplify metotlarıyla sadeleştirilebilir. Bu fonksiyon ve metot bize sadeleştirilmiş yeni bir ifade vermektedir. Örneğin:

```

>>> x = sp.Symbol('x')
>>> e = x**2 + x
>>> e
x**2 + x
>>> s = sp.simplify(e)
>>> s
x*(x + 1)
>>> s = e.simplify()
>>> s
x*(x + 1)
  
```

Örneğin:

```
>>> e = (x ** 2 - 1) / (x + 1)
>>> e.simplify()
x - 1
```

Örnek:

```
>>> e = 2 * sp.sin(x) * sp.cos(x)
>>> e.simplify()
sin(2*x)
```

Örneğin bir ÖSS sınavındaki sadeleştirme sorusuna bakalım:

$$\frac{x^2 + x - 6}{x^2 + 3x - 10} \cdot \frac{x^2 - xy + 5x - 5y}{x^2 + xy + 3x + 3y}$$

İfadelerinin en sade şekli aşağıdakilerden hangisidir?

- A) $\frac{x+5}{x-3}$ B) $\frac{x-y}{x-5}$ C) $\frac{x-y}{x+y}$
D) $x^2 + x - 6$ E) 1

Çözümü SymPy ile elde edebiliriz:

```
>>> x, y = sp.symbols('x, y')
>>> e = ((x ** 2 + x - 6) / (x ** 2 + 3 * x - 10)) * ((x ** 2 - x * y + 5 * x - 5 * y) / (x ** 2 + x * y + 3 * x + 3 * y))
>>> e
(x**2 + x - 6)*(x**2 - x*y + 5*x - 5*y)/((x**2 + 3*x - 10)*(x**2 + x*y + 3*x + 3*y))
>>> e.simplify()
(x - y)/(x + y)
```

İfadelerim Açılması (Expand)

Açım sadeleştirmenin tersidir. Yani çarpansal ifadeler çarpılarak toplamsal hale getirilir. Bu işlem sympy.expand isimli isimli fonksiyonla ya da ifade sınıflarının expand metoduyla yapılmaktadır. Örneğin:

```
>>> x = sp.Symbol('x')
>>> e = (x + 1) * (x - 3)
>>> sp.expand(e)
x**2 - 2*x - 3
```

expand fonksiyonun çeşitli biçimleri de vardır. Bu biçimler açım işlemini neye göre yapılacağını belirtir. Normal expand duruma göre bunlardan birini kullanmaktadır. expand fonksiyonlarının listyesi şöyledir:

```
expand_log
expand_mul,
expand_multinomial
expand_complex
expand_trig
expand_power_base
expand_power_exp
expand_func
```

Örneğin trigonometrik açılımları expand_trig fonksiyonuyla yapabiliriz:

```
>>> x = sp.Symbol('x')
>>> e = sp.sin(2 * x)
>>> sp.expand_trig(e)
2*sin(x)*cos(x)
```

Örneğin:

```
>>> x, y = sp.symbols('x, y')
>>> e = sp.sin(x + y)
>>> sp.expand_trig(e)
sin(x)*cos(y) + sin(y)*cos(x)
```

İfadelerin Çarpanlara Ayrılması

İfadelerin çarpanlarına ayrılması için sympy.factor fonksiyonu ya da ifade sınıflarının factor metodunu kullanılmaktadır.

Örneğin:

```
>>> x = sp.Symbol('x')
>>> e = x ** 2 - 1
>>> e.factor()
(x - 1)*(x + 1)
```

Örneğin:

```
>>> x = sp.Symbol('x')
>>> e = x ** 3 - 1
>>> e.factor()
(x - 1)*(x**2 + x + 1)
```

Örneğin:

```
>>> x = sp.Symbol('x')
>>> e = x **3 + 6 * x ** 2 + 5 * x
>>> e.factor()
x*(x + 1)*(x + 5)
```

İfadelerin Değerlerinin Elde Edilmesi

Sympy sembolik bir matematiksel işlem sunmaktadır. Ancak biz istersek belli bir ifadenin değerini elde edebiliriz. Bunun için ifade sınıflarının subs metodunu kullanılmaktadır. subs metodunun isim olarak aslında "substitute (yer değiştirme)" sözcüğünden kısaltma yapılarak uydurulmuştur. subs metodunu belli bir sembolü belli bir sembol ya da değerle yer değiştirmek için kullanılmaktadır. Örneğin:

```
>>> x, y = sp.symbols('x, y')
>>> e = x ** 2 - 1
>>> e.subs(x, y)
y**2 - 1
```

Burada ifade içerisindeki x'ler y yapılmıştır. Örneğin:

```
>>> x, y = sp.Symbol('x')
>>> e = x ** 2 - 1
>>> e.subs(x, 2)
3
```

Burada subs metodunu ile x yerine 2 yerleştirmiştir. Pekiyi birden fazla değişken varsa ne olacak? Bu durumda seçeneklerden biri birden fazla subs metodunu uygulamaktır. Örneğin:

```

>>> x, y = sp.symbols('x, y')
>>> e = x ** 2 - 3 * x * y + 2
>>> e
x**2 - 3*x*y + 2
>>> e.subs(x, 1)
-3*y + 3
>>> e.subs(x, 1).subs(y, 2)
-3

```

Bunun diğer bir yolu subs metoduna ikili elemanlardan oluşan bir dolaşılabilir nesne vermektir. Örneğin:

```

>>> x, y = sp.symbols('x, y')
>>> e = x ** 2 - 3 * x * y + 2
>>> e.subs([(x, 1), (y, 2)])

```

Diğer bir seçenek ise subs metoduna bir sözlük nesnesi vermektir. Sözlüğün anahtarları sembol isimlerinden değerleri yer değiştirecek değerlerden oluşmalıdır. Örneğin:

```

>>> x, y = sp.symbols('x, y')
>>> e = x ** 2 - 3 * x * y + 2
>>> e.subs({'x': 1, 'y': 2})
-3

```

İfadelerin değerlerini elde etmenin diğer bir yolu da ifade sınıflarının evalf metodunu kullanmaktır. Örneğin:

```

>>> e = sp.pi / 2
>>> e
pi/2
>>> e.evalf()
1.57079632679490

```

evalf metodu subs gibi yer değiştirme yapmaz. Yalnızca sembolik birtakım değerleri gerçek değerlerine dönüştürür. Örneğin sympy.pi sembolik pi değeridir. Ancak biz evalf ile bunu gerçek bir değere dönüştürebiliriz. evalf metodu parametre alabilir. Bu durumda metodun parametresi sembolik değer açılımlarının kaç basamak olacağını belirtir. Örneğin:

```

>>> e = sp.pi / 2
>>> e.evalf(20)
1.5707963267948966192

```

sympy.lamdfy isimli netot bizden bir sembol ve bir ifade ister. Bize bu ifadeyi normak bir Python fonksiyonu olarak verir. Yani artık biz subs yerine normal fonksiyon çağrıma ile değer elde edebiliriz. Örneğin:

```

>>> x = sp.Symbol('x')
>>> e = x ** 2 - 2 * x + 1
>>> f = sp.lamdfy(x, e)
>>> f(10)
81
>>> f(2)
1

```

Eğer ifadede birden fazla sembol varsa lambdşy metodunda semboller bir demet biçiminde verilmelidir. Örneğin:

```

>>> x, y = sp.symbols('x, y')
>>> e = x ** 2 + y ** 2
>>> f = sp.lamdfy((x, y), e)
>>> f(1, 2)
5

```

Burada f artık iki parametreli bir fonksiyondur.

Türev ve İntegral İşlemleri

Bir ifadenin türevi `sympy.diff` isimli fonksiyonla elde edilir. Fonksiyonun birinci parametresi türevi alınacak ifadeyi ikinci parametresi ise hangi sembole göre türevi alınacağını belirtmektedir. Örneğin:

```
>>> x = sp.Symbol('x')
>>> e = x ** 2
>>> sp.diff(e, x)
2*x
```

Burada e ifadesinin x 'e göre türevi alınmıştır. Örneğin:

```
>>> x = sp.Symbol('x')
>>> e = sp.sin(x)
>>> sp.diff(e, x)
cos(x)
```

Örneğin aşağıdaki ÖSS sorusunu çözelim:

$$f(x) = \left[1 + (x + x^2)^3 \right]^4$$

olduğuna göre, $f'(x)$ türev fonksiyonunun $x = 1$ deki değeri kaçtır?

A) $2^3 \cdot 3^5$ B) $2^3 \cdot 3^7$ C) $2^4 \cdot 3^6$

D) $2^4 \cdot 3^8$ E) $2^5 \cdot 3^{10}$

2009 ÖSS 2

Yanıtı şöyle elde edebiliriz:

```
>>> x = sp.Symbol('x')
>>> f = (1 + (x + x ** 2) ** 3) ** 4
>>> sp.diff(f, x).subs(x, 1)
104976
>>> 2 ** 4 * 3 ** 8
104976
```

Çok dereceli türevde `diff` fonksiyonuna daha fazla argüman girilir. Örneğin $x^{** 2}$ fonksiyonunun ikinci türevini alalım:

```
>>> x = sp.Symbol('x')
>>> f = x ** 2
>>> sp.diff(f, x, x)
2
```

Tabii çok değişkenli fonksiyonların da türevlerini alabiliriz:

```
>>> x, y = sp.symbols('x, y')
>>> f = x ** 2 - 2*x*y + 4
>>> sp.diff(f, x, y)
-2
```

İntegral alma işlemi de benzer biçimde `integrate` isimli fonksiyonla yapılmaktadır. Örneğin:

```
>>> x = sp.Symbol('x')
>>> f = x ** 2
>>> sp.integrate(f)
x**3/3
```

Örneğin:

```
>>> x = sp.Symbol('x')
>>> f = sp.sin(x)
>>> sp.integrate(f)
-cos(x)
```

Matematiksel Grafik Çizimleri

Python'da matematiksel grafikleri çizmek için çeşitli kütüphaneler bulunuyorsa da bunlardan en yaygın kullanılanlardan biri matplotlib isimli kütüphanedir. matplotlib aslında bir paket görünümündedir. Çizim işlemleri için bu paketin pyplot denilen modülü kullanılmaktadır. Biz de bu bölümde bu kütüphanenin temel kullanımını ele alacağız. Tabii matplotlib'in kütüphanesi de arka planda numpy kütüphanesini kullanmaktadır. Yani genellikle Python programcıları bu iki kütüphaneyi birlikte kullanırlar.

Matplotlib standart bir kütüphane olmadığı için onun da kurulumu gerekmektedir. Kurulum pip yoluyla aşağıdaki gibi yapılabilir:

```
python install -m pip matplotlib
```

Benzer biçimde PyCharm IDE'sinde de aynı şlem Setting menüsü kullanılarak yapılabilir.

Matplotlib kütüphanesinin ana dokümantasyonu aşağıdaki adreste bulunmaktadır:

<https://matplotlib.org/contents.html>

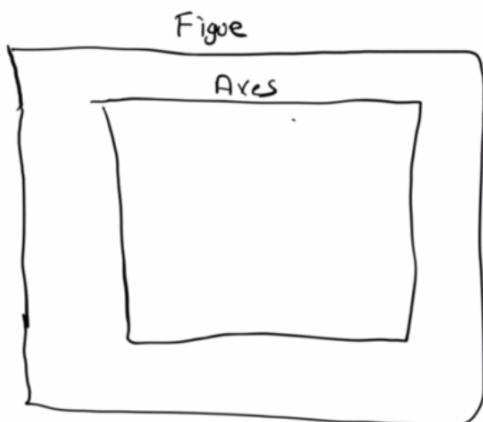
Kütüphaneniyi kullanırken aşağıdaki iki import işlemini yapacağız:

```
import numpy as np
import matplotlib.pyplot as plt
```

Matplotlib.pyplot kütüphanesinde çizim için iki önemli kavram söz konusudur:

- 1) Figure (figure) Kavramı
- 2) Eksen (axes) Kavramı

Figure grafiğin tamamını temsil eder. Eksen ise onun içerisindeki grafin çizildiği asıl alanı belirtmektedir.



O halde bizim grafiği çizmemiz için önce figure ve sonra da en az bir eksen yaratmamız gereklidir. Eksen figürün içerişindedir. Aslında bir figür birden fazla eksen içerebilir.

Pyplot modülünde çizimler tek tek global fonksiyonlar çağrılarak da sınıfların metotları çağrılarak da kullanılabilir. Başka bir deyişle kütüphane hem prosedürel hem de nesne yönelimli teknikle kullanılabilecek biçimde tasarlanmıştır. Biz burada da çok metotlar kullanarak (yani nesne yönelimli biçimde) kütüphaneyi kullanacağız.

Bir figür nesnesi yaratmak için pyplot modülündeki figure fonksiyonu kullanılır. Örneğin:

```
>>> fig = plt.figure()  
>>> fig  
<Figure size 640x480 with 0 Axes>  
>>> type(fig)  
<class 'matplotlib.figure.Figure'>
```

Aslında figure fonksiyonun pek çok default değer alan parametresi vardır. Dokümantasyonda figure fonksiyonu şöyle açıklanmıştır:

```
matplotlib.pyplot.figure  
matplotlib.pyplot.figure(num=None, figsize=None, dpi=None, facecolor=None, edgecolor=None, frameon=True, FigureClass=<class 'matplotlib.figure.Figure'>, clear=False, **kwargs)
```

Bir eksen Figure sınıfının add_axes isimli metoduna (tabii aslında global bir fonksion da vardır) figüre eklenmektedir. Örneğin:

```
>>> fig = plt.figure(facecolor='gray')  
>>> ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
```

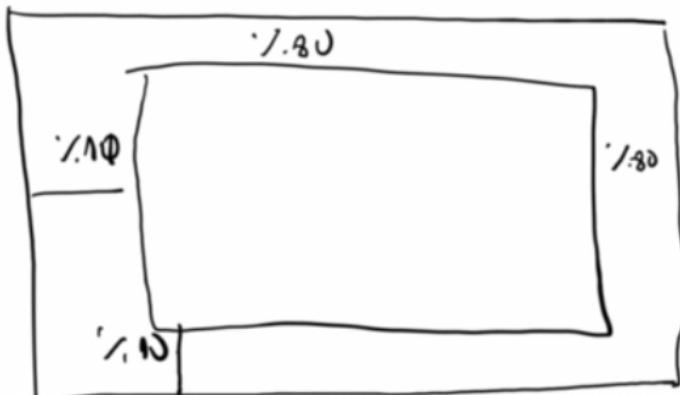
add_axes fonksiyonunun parametrik yapısı şöyledir:

```
add_axes(rect, projection=None, polar=False, **kwargs)
```

Fonksiyonun rect parametresi feksenin oransal olarak figürün neresinde görüntüleneceğini belirlemekte kullanılır. Bu parametre dört elemanlı dolaşılabilir bir nesne biçiminde girilmelidir. Bu dört elemanın anlamı şöyledir:

```
[left, bottom, width, height]
```

Bu değerler 0 ile 1 arasında oransal değerler olması gerekmektedir. Örneğin left değerinin 0.1 olması demek eksenin tüm figüre alanının sol tarafından %10'dan itibaren başlaması demektir. Örneğin [0.1, 0.1, 0.8, 0.8] değerleri şu anlama gelir:



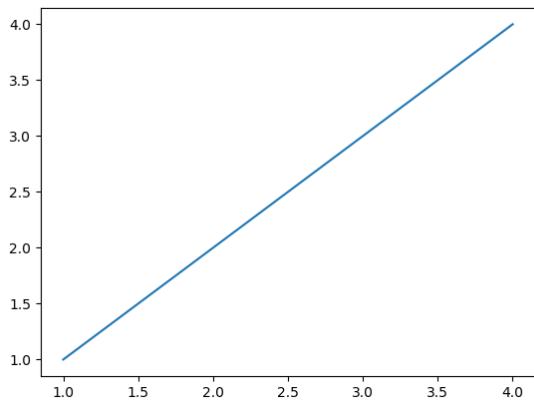
`add_axes` metodu bize yaratılan eksen nesnesini vermektedir.

Aslında figür ve ekseni yaratmak tek hamlede `pyplot` modülünün `subPlots` metoduyla da yapılmaktadır. Genellikle bu metod tercih edilmektedir. Fonksiyonun parametrik yapısı şöyledir:

```
matplotlib.pyplot.subplots(nrows=1, ncols=1, sharex=False, sharey=False, squeeze=True,
                           subplot_kw=None, gridspec_kw=None, **fig_kw)
```

Fonksiyon bize figure ve eksenden oluşan bir demet vermektedir. Fonksiyonun ilk iki parametresi olan `nrows` ve `ncols` matrisel biçimde kaç tane eksen yaratılacağını belirtir. Örneğin:

```
>>> fig, ax = plt.subplots()
>>> ax.plot([1, 2, 3, 4], [1, 2, 3, 4])
[<matplotlib.lines.Line2D object at 0x000001EC41D77A58>]
>>> fig.show()
```



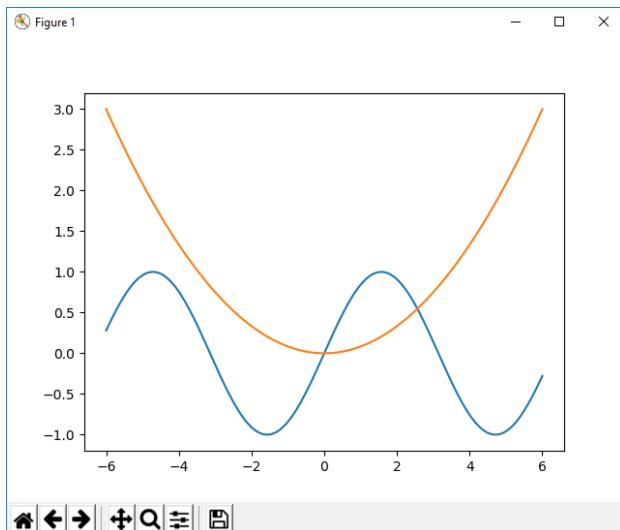
Asıl çizim işlemi axes nesnesinin `plot` fonksiyonlarıyla yapılmaktadır. `plot` fonksiyonun parametrik yapısı şöyledir:

```
Axes.plot(*args, scalex=True, scaley=True, data=None, **kwargs)
```

Fonksiyon değişken sayıda dizilim alabilmektedir. Tabii normal olarak x ve y değerleri için iki ayrı parametrenin girilmesi gereklidir. `plot` fonksiyonu birden fazla kez çağrırlırsa aynı eksen üzerinde eklemeler yapar. Örneğin:

```
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
x = np.linspace(-6, 6, 200)
y = np.sin(x)
ax.plot(x, y)
x = np.linspace(-6, 6, 200)
y = x ** 2 / 12
ax.plot(x, y)
plt.show()
```



plot fonksiyonun `**`li parametresine dikkat ediniz. Bu fonksiyon parametre isimlerinde olmayan çok sayıda isimli parametreyi `**` parametresiyle kabul etmektedir. Şimdi bu isimli parametrelerin bazlarını görelim:

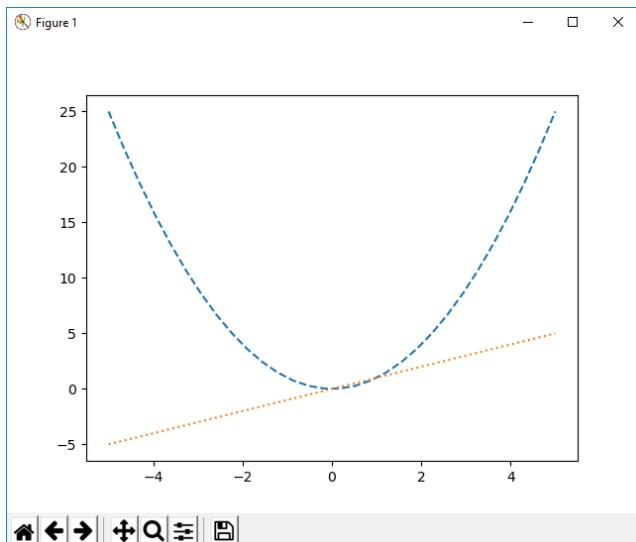
- `linestyle` isimli parametre bir string almaktadır. Çizgilerin stilini belirtir. Stiller şunlardır:

<code>linestyle</code>	<code>description</code>
'-' or 'solid'	solid line
'--' or 'dashed'	dashed line
'-. ' or 'dashdot'	dash-dotted line
::' or 'dotted'	dotted line
'None'	draw nothing
' '	draw nothing
' '	draw nothing

Örneğin:

```
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
x = np.linspace(-5, 5, 30)
y = x ** 2;
ax.plot(x, y, linestyle='--')
y = x
ax.plot(x, y, linestyle='::')
plt.show()
```

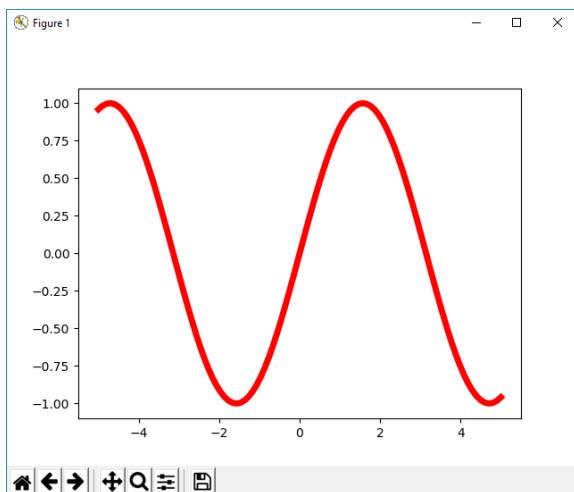


- linewidth parametresi ile çizgilerin kalınlığı ayarlanabilir.
- color parametresi çizgi rengini ayarlamak için kullanılabilir.

Örneğin:

```
import numpy as np
import matplotlib.pyplot as plt

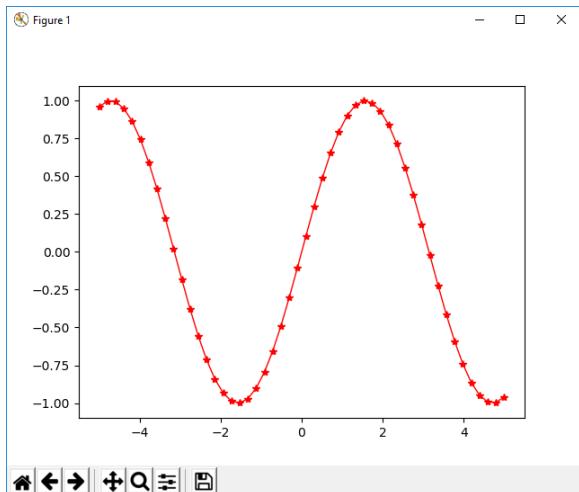
fig, ax = plt.subplots()
x = np.linspace(-5, 5, 100)
y = np.sin(x)
ax.plot(x, y, linewidth=5, color='red')
plt.show()
```



- marker parametresi grafiği oluşturan noktalar için basılacak küçük şekillerin ne olacağını belirtir. Bunun için matplotlib.pyplot dokümanlarına bakınız. Örneğin:

```
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
x = np.linspace(-5, 5, 50)
y = np.sin(x)
ax.plot(x, y, linewidth=1, color='red', marker='*')
plt.show()
```



plot fonksiyonun diğer parametreelerini dokümanlardan inceleyiniz.

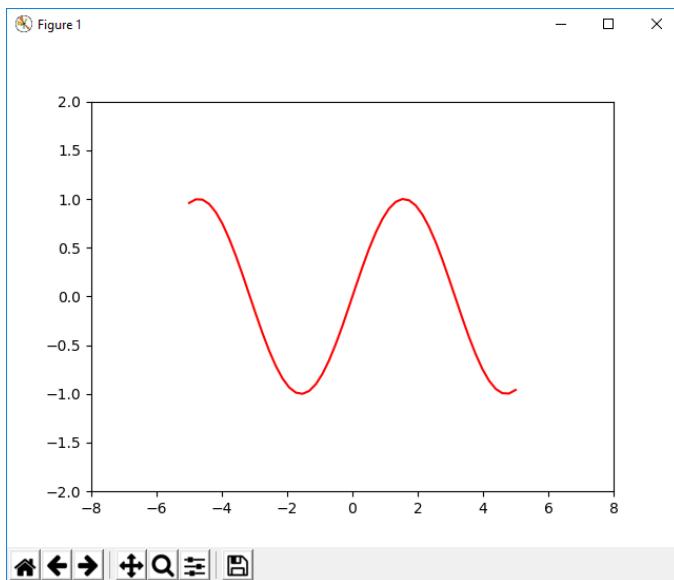
Grafik çizilirken x ve y eksenlerinin limit değerleri set_xlim ve set_ylim fonksiyonlarıyla ayarlanabilir. Örneğin:

```
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

ax.set_xlim(-8, 8)
ax.set_ylim(-2, 2)
x = np.linspace(-5, 5, 50)
y = np.sin(x)
ax.plot(x, y, color='red')

plt.show()
```



Axes sınıfının set_title isimli metodu grafiğe başlık eklemek için kullanılır. Örneğin:

```
ax.set_title('Sinüs grafiği')
```

Yine bu fonksiyonda da ** parametresine dikkat ediniz. Bu parametre sayesinde başlık yazısı çeşitli biçimlerde özelliklendirilebilir. Benzer biçimde eksenlere isimler vermek de grafiklerin okunabilirliğini artırmaktadır. Bunun için Axes sınıfının set_xlabel ve set_ylabel metotları kullanılır. Örneğin:

```

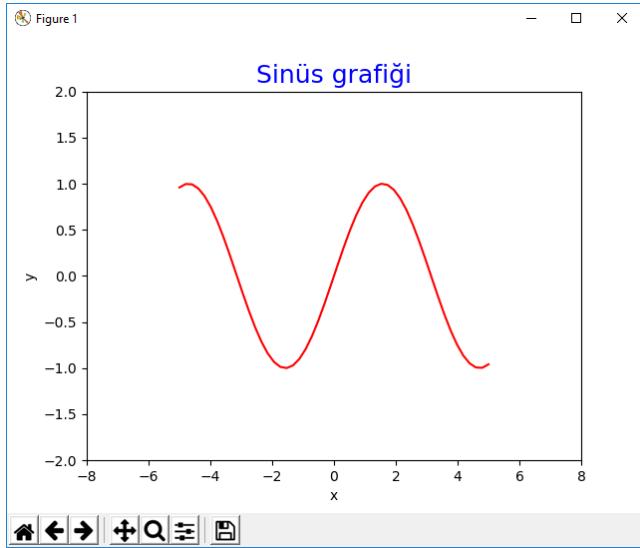
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

ax.set_title('Sinüs grafiği', color='blue', size=18)
ax.set_xlim(-8, 8)
ax.set_ylim(-2, 2)
ax.set_xlabel('x')
ax.set_ylabel('y')
x = np.linspace(-5, 5, 50)
y = np.sin(x)
ax.plot(x, y, color='red')

plt.show()

```



Eksenlere tick ataması set_xticks ve set_yticks metodlarıyla yapılmaktadır. Örneğin:

```

import numpy as np
import matplotlib.pyplot as plt

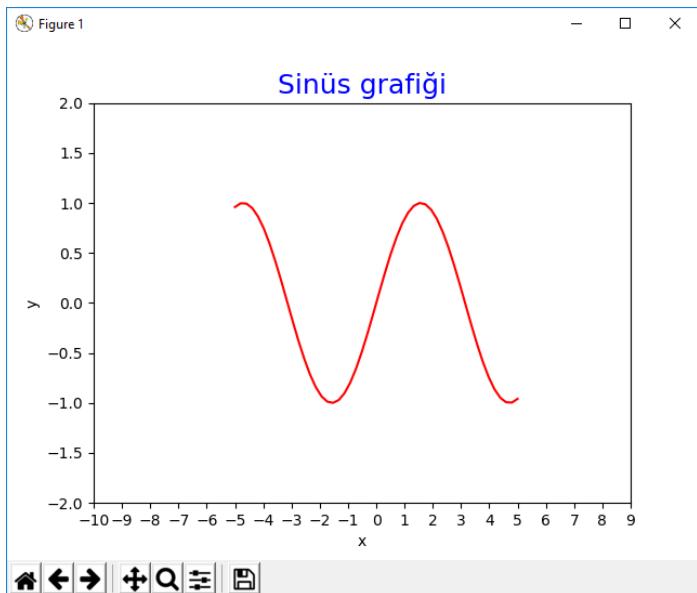
fig, ax = plt.subplots()

ax.set_title('Sinüs grafiği', color='blue', size=18)
ax.set_xlim(-8, 8)
ax.set_ylim(-2, 2)
ax.set_xticks(range(-10, 10))
ax.set_yticks([-2, -1.5, -1, -0.5, 0, 0.5, 1, 1.5, 2])

ax.set_xlabel('x')
ax.set_ylabel('y')
x = np.linspace(-5, 5, 50)
y = np.sin(x)
ax.plot(x, y, color='red')

plt.show()

```



Eksen alanını grid ile kaplayabiliriz. Bunun için Axes sınıfının grid isimli metodu kullanılmaktadır. Bu metodun parametrik yapısı şöyledir:

```
grid(b=None, which='major', axis='both', **kwargs)
```

Fonksiyon yine pek çok isimli parametreye sahiptir. Örneğin:

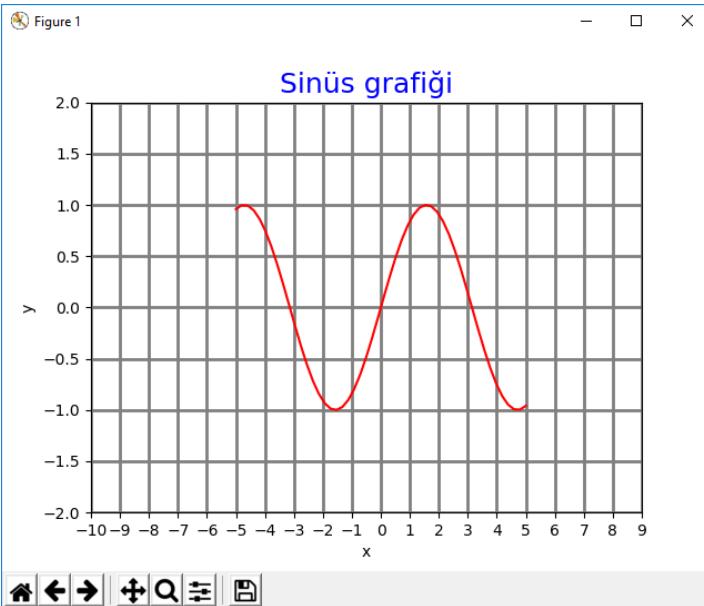
```
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

ax.grid(color='gray', linestyle='-', linewidth=2)
ax.set_title('Sinüs grafiği', color='blue', size=18)
ax.set_xlim(-8, 8)
ax.set_ylim(-2, 2)
ax.set_xticks(range(-10, 10))
ax.set_yticks([-2, -1.5, -1, -0.5, 0, 0.5, 1, 1.5, 2])

ax.set_xlabel('x')
ax.set_ylabel('y')
x = np.linspace(-5, 5, 50)
y = np.sin(x)
ax.plot(x, y, color='red')

plt.show()
```



Biz burada şimdiye kadar yalnızca plot fonksiyonunu gördük. plot fonksiyonu çizgi grafiği çizmektedir. Halbuki plot yerine kullanılabilecek başka fonksiyonlar da vardır. Bu tür çizim metotları bazıları şunlardır:

```
plot
step
bar
hist
errorbar
scatter
fill_between
quiver
pie
```

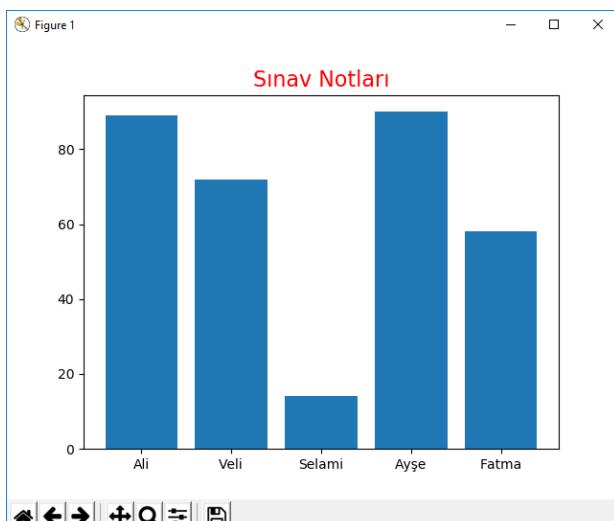
Aslında burada belirtilmeyen daha pek çok grafik türü ve o grafiği çizen çizim metotları vardır.

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

ax.set_title('Sınav Notları', size=16, color='red')
ax.bar(['Ali', 'Veli', 'Selami', 'Ayşe', 'Fatma'], height=[89, 72, 14, 90, 58])

plt.show()
```



Tabii bar fonksiyonun da pek çok isimli parametresi vardır.

pie metodu pasta dilimi grafiği çizmek için kullanılmaktadır. pie metodunun parametrik yapısı şöyledir:

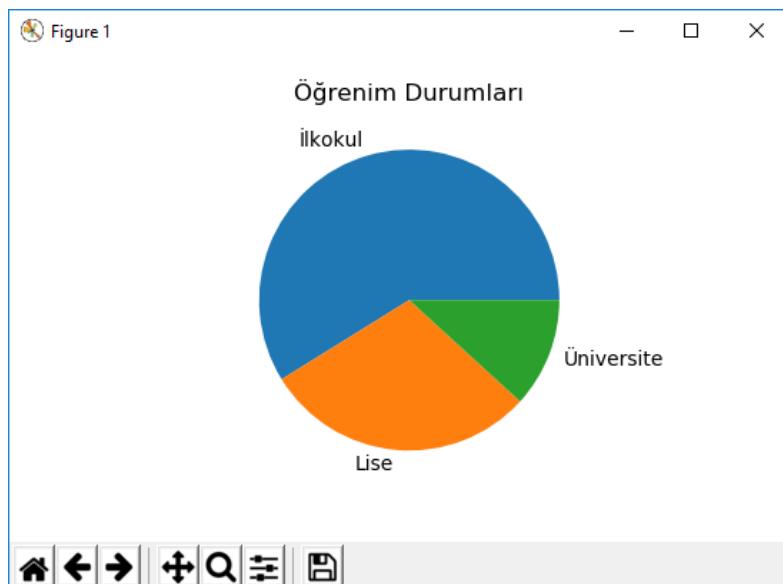
```
pie(x, explode=None, labels=None, colors=None, autopct=None, pctdistance=0.6, shadow=False,
labeldistance=1.1, startangle=None, radius=None, counterclock=True, wedgeprops=None,
textprops=None, center=(0, 0), frame=False, rotatelabels=False, *, data=None)
```

Metodun x parametresi pasta dilimindeki yüzdeleri belirtir. Metot tüm değerlerin toplamına orantı yaparak pasta dilimini çizmektedir. labels parametresi pasta diliminin ne anlam ifade ettiğine ilişkin yazılardır. Örneğin:

```
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

ax.set_title('Öğrenim Durumları')
ax.pie([10, 5, 2], labels=['İlkokul', 'Lise', 'Üniversite'])
plt.show()
```



Fonksiyonun colors parametresi ile biz istediğimiz dilime istediğimiz rengi atayabiliriz. Örneğin:

```
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

ax.set_title('Öğrenim Durumları')
ax.pie([10, 5, 2], labels=['İlkokul', 'Lise', 'Üniversite'], colors=['red', 'green', 'blue'])
plt.show()
```



Histogram çizmek için hist isimli metot kullanılır. Bilindiği gibi histogram sıklık grafiğidir. Biz histogramma bir grup veri veririz. Histogram da bunlara ilişkin bir sıklık grafiği bize çizer. hist fonksiyonunun parametrik yapısı şöyledir:

```
hist(x, bins=None, range=None, density=None, weights=None, cumulative=False, bottom=None,
histtype='bar', align='mid', orientation='vertical', rwidth=None, log=False, color=None,
label=None, stacked=False, normed=None, *, data=None, **kwargs)
```

Burada x tüm değerleri belirtmektedir. bins parametresi aralıkların kaçça bölüneceği ile ilgilidir. Burada girilen değer en büyük ve en küçük x değeri arasındaki farka bölünür ve aralıklar ona göre belirlenir. Örneğin merkezi limit teoremini ispatlamak için bir histogram çizecek olalım. Merkezi limit teoremine göre ana kütle (population) dağılımı ne olursa olsun o ana kütleden çekilen örneklemelerin ortalaması normal dağılmaktadır. Örneğin ana kütle 0 ile 100 arasındaki değerlerden oluşsun. Biz de bu ana kütleden 5'lük örneklemeler çekelim ve bunların ortalamalarını hesapyalım. Örnek bir çözüm şöyle olabilir:

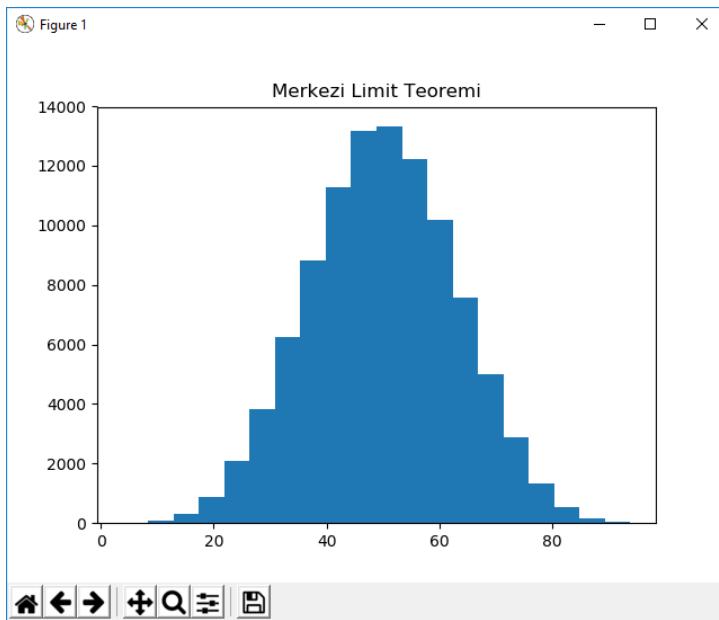
```
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

total = 100000
ssize = 5

sarray = [np.random.sample(5) * 100 for i in range(total)]
marray = [np.mean(x) for x in sarray]

ax.set_title('Merkezi Limit Teoremi')
ax.hist(marray, 20)
plt.show()
```



Aslında yukarıda da belirtildiği gibi bir figura üzerinde birden fazla eksen oluşturulabilir. Bu eksenlerin her birine farklı grafikler yerleştirilebilir. Bunun için subplots fonksiyonunda ilk iki parametreye matrisel biçimde eksen sayılarını girmek gereklidir. Örneğin:

```
fig, ax = plt.subplots(1, 3)
```

Burada 1×3 'luk bir eksen dizisi elde edilecektir. Yani ax burada artık çok boyutlu her elemanı Axes türünden olan bir ndarray nesnesi belirtmektedir.

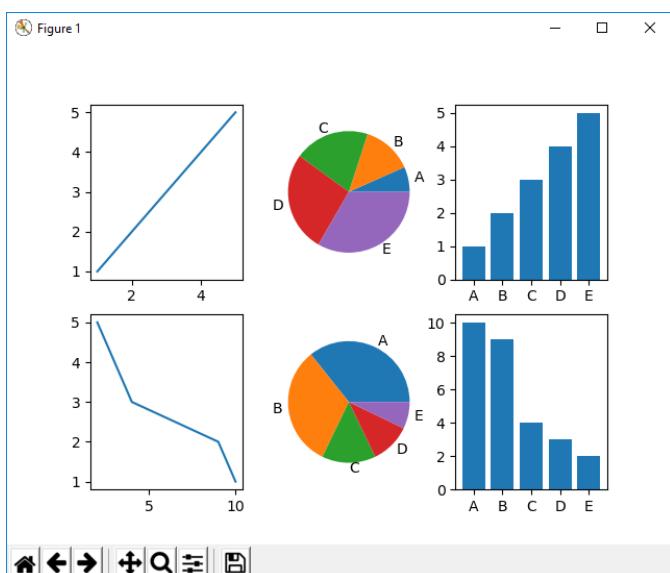
```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(2, 3)

ax[0, 0].plot([1, 2, 3, 4, 5], [1, 2, 3, 4, 5])
ax[0, 1].pie([1, 2, 3, 4, 5], labels=['A', 'B', 'C', 'D', 'E'])
ax[0, 2].bar(['A', 'B', 'C', 'D', 'E'], height=[1, 2, 3, 4, 5])

ax[1, 0].plot([10, 9, 4, 3, 2], [1, 2, 3, 4, 5])
ax[1, 1].pie([10, 9, 4, 3, 2], labels=['A', 'B', 'C', 'D', 'E'])
ax[1, 2].bar(['A', 'B', 'C', 'D', 'E'], height=[10, 9, 4, 3, 2])

plt.show()
```



Pandas Kütüphanesinin Kullanımı

Pandas kütüphanesi adeta Python'ı R benzeri bir dil gibi kullanabilme yönünde birtakım özellikler sunmaktadır. Tabii Pandas da aslında numpy üzerine oturtulmuş yüksek seviyeli bir kütüphanedir. Bu sayede biz sütunsal bilgiler üzerinde kolaylıkla işlemler yapabilmekteyiz. Pek çok istatistiksel analiz böyle sütunsal bilgiler üzerinde yürütülmektedir.

Pandas kütüphanesi de ayrıca install edilmelidir. Install işlemi aşağıdaki gibi komut satırından yapılabilir:

```
python -m pip install pandas
```

Pandas kütüphanesi için ana kaynak aşağıdaki sitedir:

<http://pandas.pydata.org/>

Burada kütüphanenin tüm dokümantasyonu elde edilebilir. Örneklerimizde aşağıdaki iki import işlemini yapacağız:

```
import numpy as np
import pandas as pd
```

Nasıl numpy kütüphanesinin en önemli veri türü ndarray ise Pandas kütüphanesinde de en önemli veri türü Series isimli türdür. Bir Series nesnesi Series isimli global fonksiyonla oluşturulabilir. Örneğin:

```
>>> import pandas as pd
>>> s = pd.Series([12, 34, 23, 67, 56])
>>> s
0    12
1    34
2    23
3    67
4    56
dtype: int64
>>> type(s)
<class 'pandas.core.series.Series'>
```

Bir Series nesnesinin iki sütun biçiminde görüntüülendiğine dikkat ediniz. Sütunlardan ilki bir indeks belirtmektedir. Diğer sütun ise gerçek verileri belirtir.

Series nesnesi içeriisndeki index ayrıca elde edilebilir. Bunun için index isimli örnek özniteliği kullanılmaktadır. Örneğin:

```
>>> i = s.index
>>> type(i)
<class 'pandas.core.indexes.range.RangeIndex'>
>>> i
RangeIndex(start=0, stop=5, step=1)
```

Yine Series içeriisndeki değerler de bağımsız olarak values isimli örnek özniteliği ile elde edilebilir. Örneğin:

```
>>> v = s.values
>>> type(v)
<class 'numpy.ndarray'>
>>> v
array([12, 34, 23, 67, 56], dtype=int64)
```

Tabii biz bir Series nesnesinin içeriisndeki belli bir indekste bulunan bilgiyi elde edebiliriz. Örneğin:

```
>>> s[0]
12
>>> s[1]
34
>>> s[2]
23
```

İstenirse indeks'ler sayısal olmaktan çıkartılıp yazısal biçimde getirilebilir. Bunun için index özniteliğine str nesnelerinden oluşan dolaşılabilir bir nesne atamak gereklidir. Örneğin:

```
>>> s = pd.Series([12, 34, 23, 67, 56])
>>> s
0    12
1    34
2    23
3    67
4    56
dtype: int64
>>> s.index = ['Ali', 'Veli', 'Selami', 'Ayşe', 'Fatma']
>>> s
Ali      12
Veli     34
Selami   23
Ayşe     67
Fatma    56
dtype: int64
```

Artık biz indekslemede sayılar yerine bu eiketleri kullanabiliyoruz. Örneğin:

```
>>> s['Ali']
12
>>> s['Veli']
34
>>> s[0]
12
>>> s[1]
34
```

Burada hala bizim sayısal indeksleri kullanabildiğimize dikkat ediniz. Ayrıca bir seride isimsel indeksler verilmişse biz sanki o isimleri bir örnek özniteliğimiş gibi de kullanabiliyoruz. Örneğin:

```
>>> s = pd.Series([1, 2, 3, 4, 5], index=['Ali', 'Veli', 'Selami', 'Ayşe', 'Fatma'])
>>> s.Ali
1
>>> s.Selami
3
```

Aslında Series fonksiyonunda biz index parametresiyle de indekslemeyi aynı anda yapabiliyoruz. Örneğin:

```
>>> s = pd.Series([1, 2, 3, 4, 5], index=['Ali', 'Veli', 'Selami', 'Ayşe', 'Fatma'])
>>> s
Ali      1
Veli     2
Selami   3
Ayşe     4
Fatma    5
dtype: int64
```

Bir seride isim de verilebilir. Bunun için sınıfın name özniteliği kullanılır. Örneğin:

```
>>> s = pd.Series([1, 2, 3, 4, 5], index=['Ali', 'Veli', 'Selami', 'Ayşe', 'Fatma'])
```

```

>>> s.name = 'Öğrenciler'
>>> s
Ali      1
Veli     2
Selami   3
Ayşe    4
Fatma   5
Name: Öğrenciler, dtype: int64

```

Nesne yaratılırken de aslında name parametresiyle isim verilebilir. Örneğin:

```

>>> s = pd.Series([1, 2, 3, 4, 5], index=['Ali', 'Veli', 'Selami', 'Ayşe', 'Fatma'],
name='Öğrenciler')
>>> s
Ali      1
Veli     2
Selami   3
Ayşe    4
Fatma   5
Name: Öğrenciler, dtype: int64

```

Bir bir seriyi indekslerken birden fazla indeks girebiliriz. Genel olarak bireden fazla indeks dolaşılabilir bir nesnelerle ifade edilebilir. Bu durumda alt bir seri elde edilir. Ancak bu seriyle diğerleri arasında bir bağlantı yoktur. Tabii yine sığ kopyalama yapılmaktadır. Örneğin:

```

>>> s = pd.Series([1, 2, 3, 4, 5], index=['Ali', 'Veli', 'Selami', 'Ayşe', 'Fatma'])
>>> k = s[['Ali', 'Selami', 'Fatma']]
>>> k
Ali      1
Selami   3
Fatma   5
dtype: int64
>>> s['Ali'] = 1000
>>> k
Ali      1
Selami   3
Fatma   5
dtype: int64

```

Bir seri üzerinde birtakım istatistiksel ve matematiksel işlemler yapabiliriz. Örneğin:

```

>>> s.max(), s.min()
(1000, 2)

```

Örneğin:

```

>>> s.sum(), s.mean(), s.median()
(1014, 202.8, 4.0))

```

Örneğin:

```

>>> s.std(), s.var()
(445.64975036456605, 198603.70000000004)

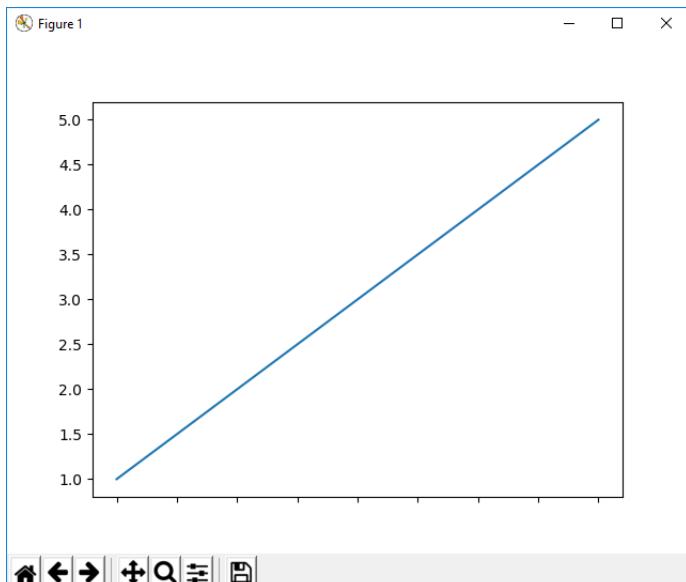
```

Örneğin bir grafik doğrudan series sınıfının metodlarıyla da çizdirilebilir. Örneğin:

```

>>> s = pd.Series([1, 2, 3, 4, 5], index=['Ali', 'Veli', 'Selami', 'Ayşe', 'Fatma'])
>>> s.plot()
<matplotlib.axes._subplots.AxesSubplot object at 0x000002FB38D2B9B0>
>>> plt.show()

```



Data Frame'ler

Data Frame tablosal bir veriyi ifade etmek için kullanılmaktadır. Data frame'ler sütunlardan oluşur. Her sütun farklı türlerden olabilmektedir. Bir data frame DataFrame fonksiyonuyla (yani bu sınıfın başlangıç metoduya) iç içe dolaşılabilir bir nesne verilerek yaratılabilir. (Örneğin liste listesi, liste demeti, demet listesi, demet demeti gibi). Örneğin:

```
>>> df = pd.DataFrame([['M', 28], ['F', 42], ['M', 56], ['M', 62]])
>>> df
   0   1
0  M  28
1  F  42
2  M  56
3  M  62
```

Burada iki sütundan oluşan tablosal bir bilgi vardır. Satırların ve sütunların indekslendiğine dikkat ediniz. Biz istersek yine index metoduya satırındaki indekslemeyi değiştirebiliriz. Örneğin:

```
>>> df = pd.DataFrame([['M', 28], ['F', 42], ['M', 56], ['M', 62]])
>>> df.index = ['Ali', 'Ayşe', 'Selami', 'Fuat']
>>> df
   0   1
Ali    M  28
Ayşe   F  42
Selami  M  56
Fuat   M  62
```

Benzer biçimde sütun indeksleri de columns özniteliği ile değiştirebiliriz. Örneğin:

```
>>> df.columns = ['Gender', 'Grade']
>>> df
      Gender  Grade
Ali        M    28
Ayşe      F    42
Selami    M    56
Fuat      M    62
```

Tabii genellikle satır indeksini değiştirmek yerine sütunlara anlamlı isimler atamak daha çok tercih edilen bir durumdurdur.

index ve columns özniteliklerini ayrı ayrı kullanmak yerine DataFrame fonksiyonunda isimli parametre biçiminde de kullanabiliriz. Örneğin:

```
>>> df = pd.DataFrame([['M', 28], ['F', 42], ['M', 56], ['M', 62]], index=['Ali', 'Ayşe', 'Selami', 'Fuat'], columns=['Gender', 'Grade'])
>>> df
   Gender  Grade
Ali      M     28
Ayşe    F     42
Selami   M     56
Fuat     M     62
```

Biz belli bir sütunu data frame içerisindeinde indeksleme yöntemiyle elde edebiliriz. İndekslemede tipik olarak sütun isimleri kullanılmaktadır. İndeksleme sonucunda -eğer tek bir sütun belirtilmişse- bize ürün olarak bir Series nesnesi verilecektir. Örneğin:

```
>>> df = pd.DataFrame([['M', 28], ['F', 42], ['M', 56], ['M', 62]], index=['Ali', 'Ayşe', 'Selami', 'Fuat'], columns=['Gender', 'Grade'])
>>> df
   Gender  Grade
Ali      M     28
Ayşe    F     42
Selami   M     56
Fuat     M     62
>>> s1 = df['Gender']
>>> type(s1)
<class 'pandas.core.series.Series'>
>>> s1
   Ali      M
   Ayşe    F
   Selami  M
   Fuat    M
Name: Gender, dtype: object
>>> s2 = df['Grade']
>>> type(s2)
<class 'pandas.core.series.Series'>
>>> s2
   Ali     28
   Ayşe    42
   Selami  56
   Fuat    62
Name: Grade, dtype: int64
```

Aslında sütun indekslemesi yaparken eğer sütunlara isim vermemişsek sütun isimleri yerine doğrudan onların indeks değerlerini de belirtebiliriz. Ancak sütuna isim verilmişse artık indekslemede sütun numarası kullanılamamaktadır. Örneğin:

```
>>> df = pd.DataFrame([['M', 28], ['F', 42], ['M', 56], ['M', 62]])
>>> df[1]
0    28
1    42
2    56
3    62
Name: 1, dtype: int64
```

İstersek bir data frame nesnesinin birden fazla sütununu da indeksleyebiliriz. Bu durumda elde edeceğimiz ürün Series değil DataFrame nesnesi olur. Birden fazla sütunun çekilmesi için köşeli parantez içeisinde dolaşılabılır bir nesne vermek gereklidir (örneğin tipik olarak bir liste). Tabii bu dolaşılabılır nesne eğer sütunlara isim verilmişse sütun isimlerinden, sütunlara isim verilmemişse sütun indekslerinden oluşacaktır.

Örneğin:

```
>>> df = pd.DataFrame([['Adana', 2216475, 1], ['İstanbul', 15029231, 34], ['Kocaeli', 1883270, 41]])
>>> c = df[[0, 1]]
>>> c
   0      1
0  Adana  2216475
1  İstanbul  15029231
2  Kocaeli  1883270
>>> type(c)
<class 'pandas.core.frame.DataFrame'>

>>> df.columns = ['Şehir', 'Nüfus', 'PlakaKodu']
>>> df
    Şehir      Nüfus  PlakaKodu
0  Adana    2216475          1
1  İstanbul  15029231        34
2  Kocaeli  1883270        41

>>> c = df[['Şehir', 'Nüfus']]
>>> type(c)
<class 'pandas.core.frame.DataFrame'>
>>> c
    Şehir      Nüfus
0  Adana    2216475
1  İstanbul  15029231
2  Kocaeli  1883270
```

Bir data frame satırlara göre de indekslenebilir. Yani data frame'den belli satırlar çekilebilir. Bunun için eğer sayısal temelde (yani index numarası ile) indeksleme yapılacaksa iloc isimli property, eğer isimsel temelde indeksleme yapılacaksa loc isimli property kullanılmaktadır. (Eski den ix property'si kullanılıyordu. Ancak pandas'ın ileri sürümlerinde bu property "deprecated" yapılmıştır.) Örneğin:

```
>>> df = pd.DataFrame([['Adana', 2216475, 1], ['İstanbul', 15029231, 34], ['Kocaeli', 1883270, 41]], columns=['Şehir', 'Nüfus', 'PlakaKodu'])
>>> r = df.iloc[2]
>>> r
Şehir      Kocaeli
Nüfus    1883270
PlakaKodu     41
Name: 2, dtype: object
>>> type(r)
<class 'pandas.core.series.Series'>
```

Yine birden fazla satırı çektiğimizde bize o satırlar Series olarak değil DataFrame nesnesi olarak verilecektir. Örneğin:

```
>>> r = df.iloc[[1, 2]]
>>> r
    Şehir      Nüfus  PlakaKodu
1  İstanbul  15029231        34
2  Kocaeli  1883270        41
>>> type(r)
<class 'pandas.core.frame.DataFrame'>
```

Aslında tıpkı Series nesnesinde olduğu gibi eğer sütunlara isim verilmişse sanki o isimler bir property gibi de kullanılabilir. Örneğin:

```
>>> df = pd.DataFrame([['Adana', 2216475, 1], ['İstanbul', 15029231, 34], ['Kocaeli', 1883270, 41]], columns=['Şehir', 'Nüfus', 'PlakaKodu'])
>>> df.Şehir
```

```

0      Adana
1    İstanbul
2   Kocaeli
Name: Şehir, dtype: object
>>> df.Nüfus
0    2216475
1   15029231
2   1883270
Name: Nüfus, dtype: int64

```

DataFrame sınıfının info isimli metodu bize o data frame ile ilgili ayrıntılı bilgiler vermektedir. Örneğin:

```

>>> df = pd.DataFrame([['Adana', 2216475, 1], ['İstanbul', 15029231, 34], ['Kocaeli', 1883270, 41]], columns=['Şehir', 'Nüfus', 'PlakaKodu'])
>>> df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 3 columns):
Şehir            3 non-null object
Nüfus            3 non-null int64
PlakaKodu       3 non-null int64
dtypes: int64(2), object(1)
memory usage: 152.0+ bytes

```

Tıpkı numpy'in ndarray nesnesinde olduğu gibi Series nesnelerinde ve DataFrame nesnelerinde de belli sütunlar üzerinde vektörel işlemler yapılabilmektedir. Örneğin:

```

>>> df = pd.DataFrame([['Adana', 2216475, 1], ['İstanbul', 15029231, 34], ['Kocaeli', 1883270, 41]], columns=['Şehir', 'Nüfus', 'PlakaKodu'])
>>> df['Nüfus'] / 10
0    221647.5
1   1502923.1
2   188327.0
Name: Nüfus, dtype: float64
>>> df[['Nüfus', 'PlakaKodu']] + 10
   Nüfus  PlakaKodu
0  2216485     11
1  15029241     44
2  1883280     51

```

Yani Series nesneleri ve DataFrame nesneleri de vektörel işlem yeteneğine sahiptir. Böylece típkı ndarray nesnelerinde olduğu gibi biz bu nesnelerde de Bool türden indeksleme yoluyla belli kayıtları seçebiliriz. Örneğin:

```

>>> df['Nüfus'] > 2000000
0    True
1    True
2   False
Name: Nüfus, dtype: bool
>>> df[df['Nüfus'] > 2000000]
   Şehir  Nüfus  PlakaKodu
0  Adana  2216475        1
1  İstanbul  15029231      34

```

Pandas özellikle istatistiksel amaçla çok yoğun kullanılmaktadır. İstatistiksel veriler de genellikle program içerisinde elle girilmek yerine bir dosyadan okunurlar. İşte Pandas'ın read_csv isimli fonksiyonu CSV (comma separated vector) biçimindeki text dosyalarını okuyarak onların içeriğini bize DataFrame nesnesi olarak verir. read_csv fonksiyonun pek çok default parametresi vardır. Bu default parametreler CSV dosyalarındaki farklılıklarını ele almak için kullanılmaktadır. Standart CSV dosyalarında sütunlar yalnızca ',' karakteri ile ayrılmaktadır. Ancak bazı CSV dosyalarında ',' karakterinden sonra 'SPAVE' karakteri de bulunmaktadır. Örneğin aşağıdaki CSV dosyasının içeriği ("oscar_age_male.csv") aşağıdaki gibidir:

```
"Index", "Year", "Age", "Name", "Movie"
1, 1928, 44, "Emil Jannings", "The Last Command, The Way of All Flesh"
2, 1929, 41, "Warner Baxter", "In Old Arizona"
3, 1930, 62, "George Arliss", "Disraeli"
4, 1931, 53, "Lionel Barrymore", "A Free Soul"
5, 1932, 47, "Wallace Beery", "The Champ"
6, 1933, 35, "Fredric March", "Dr. Jekyll and Mr. Hyde"
7, 1934, 34, "Charles Laughton", "The Private Life of Henry VIII"
....
```

Burada standart olmayan (yani ',' den sonra SPACE olan) CSV dosyası okunurken skipinitialspace parametresi True geçilmelidir. Okuma işlemi şöyle yapılabilir:

```
import pandas as pd

df = pd.read_csv(r'D:\Dropbox\Kurslar\Python-App\Src\Sample\oscar_age_male.csv',
skipinitialspace=True)
print(df[['Age', 'Year']])
```

Göründüğü gibi çok büyük dosyalar bu biçimde bir DataFrame olarak elde edilebilmektedir. Artık bundan sonra programcı bu sütunlar üzerinde çeşitli istatistiksel işlemleri yapabilir. Örneğin Oscar ödülü alanların yaş ortalamasını şöyle bulabiliyoruz:

```
import pandas as pd

df = pd.read_csv(r'D:\Dropbox\Kurslar\Python-App\Src\Sample\oscar_age_male.csv',
skipinitialspace=True)

print(df['Age'].mean())
```