Bölüm

2

Döngüler ve Koşullu Yapı

Bilgisayarın ana fikri tekrarlanan işlemlerin otomasyonuna dayanır. Bu bölümde, aynı işlemi farklı durumlara uygulamak için **döngülerin** nasıl kullanılacağını öğreneceğiz. Python'da iki temel döngü çeşidi vardır, bunlar **while** ve **for** döngüleri olarak adlandırılırlar. Bu döngüleri çalışabilmek için önce iki yeni veri tipini öğrenmemiz gerekecek, **bool** ve **list** veri tipleri. Bu veri tiplerini sadece bu bölümde döngüler için değil, metnin geri kalanında bir çok konu başlığında kullanacağız. Bu bölümde ayrıca `if-else` **koşullu yapısıyla** programlarımız içinde karar alma stratejileri geliştirmeyi öğreneceğiz. Bu sayede programlarımız farklı stratejiler üzerinden çalışabilecek.

2.1 While Döngüleri

Bool Verileri

Python'da değeri sadece `True` veya `False`, yani `Doğru` veya `Yanlış` olabilen özel bir veri tipi vardır, bu nesnelere **bool** (boolean) nesnesi denir. Bool nesneleri farklı yöntemlerle oluşturulabilir, en sık kullanılan yöntem mukayese yapmaktır. İki nesne arasında bir bağıntı oluşturup buun doğruluk değerini bool verisi olarak alabiliriz. Örneğin `x < 2` bağıntısı 2'den küçük değişken değerleri için `True`, diğer durumlarda `False` olacaktır. Kullanabileceğimiz diğer bağıntılar ==, !=, <=, >= biçimindedir ve bunlar eşittir, eşit değildir, küçük eşittir, büyük eşittir anlamlarına sahiptir. Dikkat edin, = bir değişkene atama yaparken == bir bağıntı kurar!

Terminal

```
>>> x = 5
>>>
>>> x < 2
False
>>> x > 0
True
>>> x != 5
False
>>> 2*x -1 >= 10
False
```

```
>>> x**2 - 5 == 20
True
```

Bool nesneleri `and` ve `or` *bağlaçlarıyla* birleştirilip yeni bool nesneleri oluşturulabilir. Bu bağlaçlar matematikte kullandığımız **ve** ile **veya** ($\wedge$ ile $\vee$) ile aynı şekilde çalışır. Yani `and` ile bağlanmış bool verilerinden en az biri `False` ise bu verinin değeri de `False`, aksi durumda `True` olur. Benzer şekilde `or` ile bağlanmış bool nesnelerinden en az birisi `True` ise nesne `True`, aksi durumda `False` değer döndürür. Ayrıca bir bool nesnesinin önüne `not` ifadesi bu nesnenin değerini tersine çevirir.

Terminal

```
>>> from scipy import sqrt
>>>
>>> x = 2
>>> x >= 0 or x**2 != 2*x
True
>>> not x >= 0 or x**2 != 2*x
False
>>> x**2 - 2*x > 0 or sqrt(x - 1) == 0
False
>>> y = x**2 - 2*x -1
>>> sqrt(y).real <= 0 and sqrt(y).imag != 0
True
```

Son olarak bool nesneleri hakkında bir kaç teknik bilgi verelim. Aslında Python'da **her** nesnenin bir bool değeri vardır ve bu değere `bool(nesne)` komutuyla ulaşılabilir. Buradaki nesne kendisi bir bool verisi değil ise `nesne` sonucunun değeri hesaplanırken temel kural şöyledir. Sıfır sayıları (`0` veya `0.0`), boş string nesneleri (`""` veya `''`), boş list, tuple, dict gibi veriler (bunları daha sonra öğreneceğiz) `False`, diğer tüm nesneler `True` bool değerine sahiptir. Biz `x or y` veya `x and y` komutunu girdiğimizde Python aslında `True` veya `False` değerini döndürmez, arka planda `x` veya `y` nesnelerinden birini sonuç olarak döndürür, bu nesnenin de bool değeri hesaplanıp ekrana yazdırılır.

Terminal

```
>>> bool(x)
True
>>> bool(x - 2)
False
>>> x or x + 1 or 5 or "python"
2
>>> x - 2 or 0 or 2*x - 4 or x + 1 or "python"
3
>>> x - 2 or 0 or 2*x - 4 or x**2 - 4 or "python"
'python'
>>> x - 2 or 0 or 2*x - 4 or x**2 - 4
0
>>> x and 2*x and x - 1 and "python"
'python'
```

```
>>> x and 2*x and x - 2 and "python"
0
```

Anlaşılacağı üzere `or` ile bağlanmış ifadelerin en az bir bileşeni `True` ise sonuç `True` olacağından ilk `True` değeri ile karşılaşılınca gerisi değerlendirilmez. Yani `or` ile bağlı ifadelerin değeri ilk karşılaşılan `True` nesnedir, `True` nesne yoksa değeri son nesnedir. `and` ile bağlı ifadelerde durum bunun tam tersidir. Son olarak şunu da ekleyelim, bir nesnenin `bool(nesne)` ile getirilen bool değeri cebirsel işlemlerse `0` ve `1` olarak değerlendirilir.

Terminal

```
>>> bool(x) + 3
4
>>> 2*bool(x > 0) + 3
5
>>> bool(x - 2) + 10
10
```

Döngü Yapısı

İstatistik hesabında normal dağılımı tanımlamak için kullanılan ve **Gauss Fonksiyonu** olarak adlandırılan

$$f(x) = \frac{1}{\sqrt{2\pi}s}\exp\left[-\frac{1}{2}\left(\frac{x-m}{s}\right)^2\right]$$

fonksiyonunu ele alalım, burada m ve $s > 0$ reel parametrelerdir. Amacımız $m = 0$ ve $s = 1$ seçip $x = -3, -2, -1, 0, 1, 2, 3$ için $f(x)$ değerlerini bir tablo halinde yazdırmaktır. Bunu aşağıdaki program yapabilir.

```
1  from math import sqrt, pi, exp
2
3  m = 0.0
4  s = 1.0
5  K = (1/(sqrt(2*pi)*s))
6
7  x=-3.0; f=K*exp(-(1.0/2)*((x-m)/s)**2); print "x=%2d f(x)=%g" % (x, f)
8  x=-2.0; f=K*exp(-(1.0/2)*((x-m)/s)**2); print "x=%2d f(x)=%g" % (x, f)
9  x=-1.0; f=K*exp(-(1.0/2)*((x-m)/s)**2); print "x=%2d f(x)=%g" % (x, f)
10 x=0.0; f=K*exp(-(1.0/2)*((x-m)/s)**2); print "x=%2d f(x)=%g" % (x, f)
11 x=1.0; f=K*exp(-(1.0/2)*((x-m)/s)**2); print "x=%2d f(x)=%g" % (x, f)
12 x=2.0; f=K*exp(-(1.0/2)*((x-m)/s)**2); print "x=%2d f(x)=%g" % (x, f)
13 x=3.0; f=K*exp(-(1.0/2)*((x-m)/s)**2); print "x=%2d f(x)=%g" % (x, f)
```

Yukarıdaki yazımdan fark ettiğiniz üzere Python'da komutlar farklı satırlar yerine, noktalı virgül ile ayrılmak şartıyla, aynı satırda da yazılabilir. Şimdi yukarıdaki programı çalıştırırsak aşağıdaki gibi bir çıktı alırız.

Terminal
```
Terminal > python gauss1.py
x=-3 f(x)=0.00443185
x=-2 f(x)=0.053991
x=-1 f(x)=0.241971
x= 0 f(x)=0.398942
x= 1 f(x)=0.241971
x= 2 f(x)=0.053991
x= 3 f(x)=0.00443185
```

Bu çıktı tam istediğimiz gibi, fakat programın kodları üzerinde biraz durmamız gerekiyor. Buradaki sorun şu, aynı (veya benzer) komutlar bir çok defa tekrarlanıyor. Aynı işlemleri farklı değişken değerlerine defalarca uygulamak için bir **döngü** kullanabiliriz.

Python'da **while** döngüleri, belirli bir **koşul** sağlandığı sürece bir işlemi tekrarlamaya yarar. Buradaki koşul bir bool verisi ile belirtilir, dolayısı ile bu bool nesnesinin değeri `True` olduğu sürece belirtilen bir işlem tekrarlanır. Bu döngünün söz dizimini aşağıdaki örnek üzerinde gözlemleyelim.

```
1  from math import sqrt, pi, exp
2
3  m = 0.0
4  s = 1.0
5  K = (1/(sqrt(2*pi)*s))
6
7  x = -3
8  while x < 4:
9      f = K*exp(-(1.0/2)*((x - m)/s)**2)
10     print "x=%2d f(x)=%g" % (x, f)
11     x = x + 1
```

Bu program da yukarıdaki çıktının aynısını üretir fakat çok daha kompakt ve modern bir kod ile. Şimdi bu kodları açıklayalım, `x = -3` satırına kadar önceki kodlar ile aynıdır. Bu satırda tanımladığımız `x` değişkenine kuracağımız döngü için bir **sayaç** değişkeni deriz, aslında bu değişken bizim Gauss fonksiyonumuzun ana değişkenidir. Daha sonra bir sonraki satıra `while` kelimesini takip eden bir bool nesnesi ve hemen sonrasında iki nokta üst üste yazılıp alt satıra geçilir. Tekrarlanmasını istediğimiz işlemleri bir veya birkaç satırda belirtiriz fakat bu satırların `while` satırı ile aynı paragraf hizasında başlamaması gerekiyor. Tekrar işlemleri belirten satırların hepsi aynı hizada ama `while` satırından daha sağdan başlamalı. Yukarıdaki örnekte tekrarlanmasını istediğimiz işlemleri dört karakter boşluk (bir tab uzunluğu kadar) bıraktıktan sonra başlattık. Bu boşluk sayısının dört olması zorunlu değildir, istediğimiz uzunluğu seçebiliriz ama dört karakter seçimi Python'da bir gelenek olmuştur. Bir `while` döngüsünün amacı, altındaki kendinden farklı hizada bulunan tüm satırları (kendi ile aynı hizada bir satır görene kadar bütün satırları) belirtilen bool nesnesi `True` olduğu sürece tekrarlamaktır. Dolayısıyla sonsuz bir döngüye girmemek için bu satırlar içinde belirtilen bool nesnesini etkileyecek işlemler de yapmalıyız. Yukarıdaki örnekte döngüden önce `x = -3` değişkenini tanımladık, daha sonra `while` döngüsünde `x < 4` olduğu sürece bu `x` değişkenini kullanarak `f` değerini hesaplayıp yazdırmasını istedik. Döngü içinde bu yazım işleminden sonra `x` değişkeni değerini 1 arttırdık, döngünün sonraki adımına bu şekilde gitmezsek döngü aynı `x` değeri ile aynı `f` değerini yazdırır ve bunları tekrarlayarak hiç bir zaman

durmazdı. Ama bu şekilde değişkeni her tekrarda arttırınca belirli sayıda tekrardan sonra `x < 4` bool verisi `False` olacaktır ve döngü sona erecektir. Bundan sonra programımız döngüden sonraki kodlarla devam edecektir.

Bu döngü içinde kullandığımız `x = x + 1` ifadesine dikkat edelim. Bu ifade matematiksel olarak bir çelişki ise de Python dilinde sıklıkla kullanılan bir ifadedir. Python'da = ile bir değişkene değer atanırken önce eşitliğin sağ tarafındaki ifade hesaplanıp sonucu bellekte tutulur, daha sonra bu bellekteki değere eşitliğin solundaki isim atanır. Burada ele aldığımız örnekte önce `x + 1` değeri hesaplanıp belleğe alınır, hali hazırda bellekte `x` adında bir değişkenin değeri vardı. Daha sonra bu yeni hesaplanan değere `x` ismi atanır, daha önceden bu isimle başka bir değer vardı ve bu yeni atamadan sonra bu ismin atıf verdiği eski değer bellekten silinir. Böyle işlemler Python döngüleri içinde sıklıkla yapıldığı için kısa yolları vardır, `x = x + 1` ifadesi ile `x += 1` ifadesi aynı atamayı yapar. Diğer cebirsel işlemler için de `-=, *=, /=, **=` kısaltmaları kullanılabilir. Dolayısıyla aşağıdaki program da yukarıdakilerle aynı çıktıyı verecektir.

```
from math import sqrt, pi, exp

m = 0.0
s = 1.0
K = (1/(sqrt(2*pi)*s))

x = -3
while x < 4:
    f = K*exp(-(1.0/2)*((x - m)/s)**2)
    print "x=%2d f(x)=%g" % (x, f)
    x += 1
```

Sonlu Toplamlar

Bilimsel hesaplamalarda $\sum x_n$ biçiminde toplamlarla çok sık karşılaşırız, böyle **sonlu** toplamları hesaplarken döngüleri kullanırız. Uygun bir ilk terim tanımladıktan sonra kurulan döngünün her adımında toplamın bir terimi hesaplanıp ilk terime eklenir ve bu terimin değeri güncellenir. Ayrıca bir de sayaç değişkeni tanımlanıp toplam kaç adım hesaplanacaksa ona göre bir bool verisi oluşturulur. Örneğin

$$f(t) = \begin{cases} 1, & 0 < t < T/2 \\ 0, & t = T/2 \\ -1, & T/2 < t < T \end{cases}$$

fonksiyonunun bir t noktasındaki yaklaşık değerlerini

$$S(t;n) = \frac{4}{\pi} \sum_{i=1}^{n} \frac{1}{2i-1} \sin\left(\frac{2(2i-1)\pi t}{T}\right)$$

toplamı ile hesaplayabiliriz. Bu tip serilere literatürde **Fourier serileri** denir ve bu örnekteki $f(t)$ için $n \to \infty$ için $S(t;n) \to f(t)$ olduğu gösterilebilir, belirli şartları sağlamayan fonksiyonların Fourier serileri yakınsak olmayabilir. Bu tür serilerin elektronik mühendisliği, sinyal ve resim işleme, akustik, kuantum mekaniği gibi bir çokalanda kullanım alanı vardır. Buradaki toplamı aşağıdaki programda görülen döngü ile hesaplayabiliriz.

```
from math import sin, pi

t = pi/2
i = 1
n = 10000
T = 2*pi
s = 0

while i <= n:
    terim = (1.0/(2*i - 1))*sin((2*(2*i - 1)*pi*t)/T)
    s += terim
    i += 1

toplam = (4/pi)*s
print "f(t)=%.16f  (yaklasik hesap, n=%d)" % (toplam, n)
```

Bu programda $f(\pi/2)$ değerini hesaplamak için yukarıdaki seriyi $(0, 2\pi)$ aralığında 10.000 adım ile kullandık. Fonksiyon tanımından $f(\pi/2) = 0$ olduğu görülüyor, bu hesaplama ile de yakın bir sonuç bulacağımızı umuyoruz. Programın çıktısı aşağıdaki gibi olur.

```
Terminal
Terminal > python fourier1.py
f(t)=0.9999681690114585  (yaklasik hesap, n=10000)
```

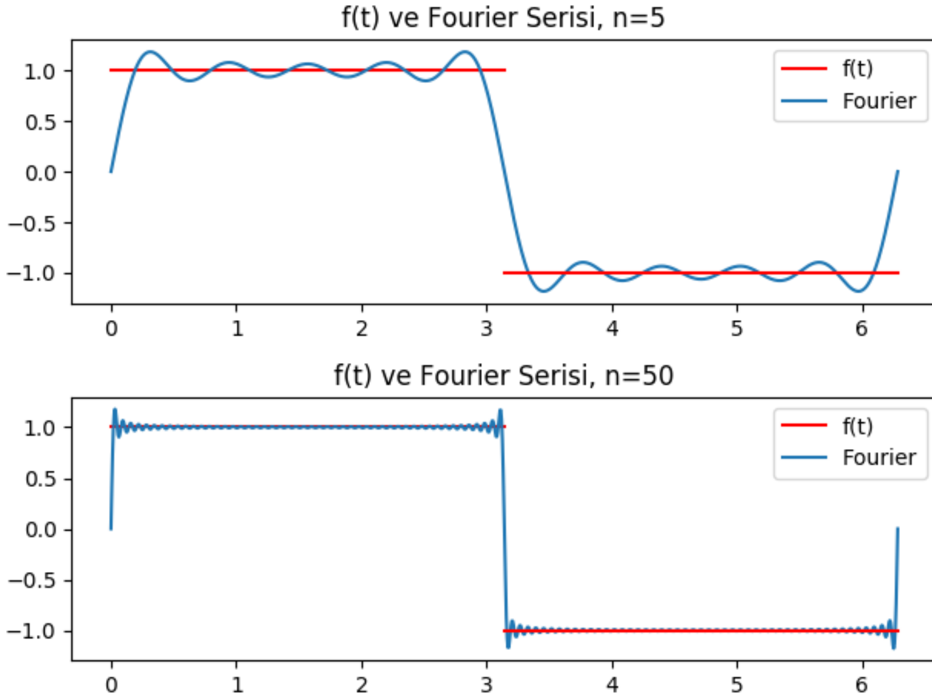
Görüldüğü gibi sonuç gerçek değere yakın, adım sayısı arttırılarak daha da yakın sonuçlar elde etmeyi umabiliriz. Gerçekten $t = \pi/2$ noktasında böyle olacaktır ama süreksizlik noktaları olan $t = 0$, $t = \pi$ ve $t = \pi/2$ noktalarına yakın noktalarda adım sayısını arttırmak çok işe yaramaz. Aşağıda verilen grafikte bu durumu gözlemleyebilirsiniz. Bu durum Fourier serilerinin doğasıyla alakalı bir durumdur ve literatürde **Gibbs olgusu** olarak adlandırılır.

2.2 For Döngüleri

List Verileri

Programlarımızda sıklıkla tek bir nesne ile çalışmak yerine birden fazla nesneyi bir grup haline getirerek bütün bir grup ile işlemler yaparız. Python'da böyle nesne grupları oluşturmanın bir yolu `list` (**liste**) verileri oluşturmaktır. Python'da `list` verisi oluşturmanın bir yolu bazı nesneleri köşeli parantez içinde belirli bir **sıra** dahilinde virgülle ayırarak yazmaktır.

```
Terminal
>>> L = [0, 2, 4, 6, 8, 10]
>>> print L
[0, 2, 4, 6, 8, 100]
>>> type(L)
<type 'list'>
```



Bu tanımladığımız list nesnesinin tüm elemanları birer `int` nesnesidir ama genelde böyle bir kural yoktur, elemanlar farklı tipten veriler de olabilir. Yukarıda tanımladığımız `L` değişkeni, altı tane veriden oluşan list tipi veriyi temsil ediyor.

Bir listenin her bir öğesinin bir indisi (**index**) vardır, ilk öğenin indisi 0, sonrakinin 1, ve böyle devam eder. Bir listenin tekil bir elemanına ulaşmak için o öğenin indisini kullanarak `liste[indis]` komutunu gireriz. Python'da alternatif olarak farklı bir indisleme sistemi daha vardır. Bu sisteme göre listenin son elemanının indisi -1, bir öncekinin indisi -2, ve diğerleri de böyle devam eder. İstenilirse baştan istenilirse sondan indisleme sistemi kullanılarak işlem yapılabilir. Bu şekilde indisleme yöntemiyle bir `list` nesnesinin elemanları üzerinde manipülasyon yapabiliriz, elemanları veya sıralarını değiştirebiliriz.

Terminal

```
>>> L
>>> [0, 2, 4, 6, 8, 10]
>>> L[0]
0
>>> L[1]
2
>>> L[4]
8
>>> L[-1]
10
>>> L[-5]
2
>>> L[-1] = 100
```

```
>>> L
[0, 2, 4, 6, 8, 100]
```

Bu L nesnesinin **sonuna** yeni bir eleman eklemek için `L.append(nesne)` komutunu gireriz. Herhangi bir `i` indisli konuma veni bir nesne eklemek için se `L.insert(i, nesne)` komutunu kullanırız, bu durumda daha sonraki elemanların indislerin değişeceğine dikkat edin. Bu L nesnesinin bir elemanının indis numarasını öğrenmek için `L.index(nesne)` komutundan faydalanırız, herhangi bir `i` indisli elemanı silmek için `del L[i]` komutunu kullanırız. Ayrıca bir listenin eleman sayısı bilgisi `len(L)` komutu ile getirilebilir. `x in L` ifadesi bir bool verisidir ve `x` nesnesi `L` listesinin bir elemanı ise `True`, aksi taktirde `False` değerini döndürür.

Terminal

```
>>> L
[0, 2, 4, 6, 8, 100]
>>> len(L)
6
>>> L.append("python")
>>> L
[0, 2, 4, 6, 8, 100, 'python']
>>> L.index("python")
6
>>> "python" in L
True
>>> del L[-1]
>>> "python" in L
False
>>> L
[0, 2, 4, 6, 8, 100]
L.insert(1, "python")
>>> L
[0, 'python', 2, 4, 6, 8, 100]
```

Python `list` nesneleri üzerinde toplama, çıkarma ve çarpma gibi bazı aritmetik işlemler de tanımlanmıştır, aşağıdaki örneklerde bunların nasıl davrandığını gözlemleyebilirsiniz.

Terminal

```
>>> L
[0, 'python', 2, 4, 6, 8, 100]
>>> L1 = [0]
>>> L2 = 5*L1
>>> L2
[0, 0, 0, 0, 0]
>>> L3 = L + L2
>>> L3
[0, 'python', 2, 4, 6, 8, 100, 0, 0, 0, 0, 0]
```

Sıklıkla elemanları belirli bir düzene göre sıralanmış sayı listelerine ihtiyacımız olacak, böyle listeleri yukarıdaki örneklerde yaptığımız gibi açıkça tanımlamak yerine bir döngü yardımıyla kolayca oluşturabiliriz. Bunun için önce boş bir liste oluşturup döngü içinde her bir elemanı hesaplayıp listenin sonuna ekleriz.

```
L = [] #bos bir list

a = 0
b = 30

while a <= b:
    L.append(a)
    a += 3

print L
```

Bu program 0 ile 30 arasında (0 ve 30 dahil) 3'ün katı olan tamsayıların bir listesini oluşturup yazdırır. Programın çıktısı aşağıdaki gibi olacaktır.

```
Terminal
>>> python list1.py
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
```

Bu işlemin çok sık yapıldığı için Python'da bu işlem için hazır bir `range(a, b, s)` fonksiyonu vardır. Bu fonksiyon `a` ile `b` sayıları arasında (`b` dahil **değil**) `s` aralıklı sayıların bir listesini oluşturur. Bu komutun bir kaç farklı kullanım yöntemi daha vardır.

- `range(n)` = `range(0, n, 1)` = `[0, 1, 2, .., n-1]`
- `range(n, k)` = `range(n, k, 1)` = `[n, n+1, n+2, .., k-1]`

Bunları aşağıdaki örnekleri uygulayarak gözlemleyebilirsiniz.

```
Terminal
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(3, 8)
[3, 4, 5, 6, 7]
>>> range(-5, 12, 3)
[-5, -2, 1, 4, 7, 10]
```

Tuple Verileri

Python'da listelere çok benzeyen bir de `tuple` verileri vardır. Bu veri tipi `list` veri tipine çok benzerdir, bunlar da birden fazla elemanı olan sıralı verilerdir. Köşeli parantez yerine normal parantez içinde yazılırlar ve tanımlandıktan sonra elemanları **değiştirilemez**. Bu veriler de list verileri gibi kullanılırlar, indisleme ve döngü içinde kullanımları tamamen aynıdır. `tuple` verileri üzerinde işlemler genelde `list` verilerine çok daha hızlıdır, dolayısıyla elemanların sonradan güncellenmesi gerekmiyorsa bu verileri kullanmak program verimliliği açısından

önemlidir. Ayrıca daha sonra değişmemesi gerekn verileri korumak için `list` yerine `tuple` verileri kullanılabilir.

Terminal
```
>>> T = (1, 3, 5, 7, 9)
>>> type(T)
<type 'tuple'>
>>> T[2]
5
>>> T[-2]
7
>>> 3 in T
True
>>> T.append(11)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```

Dilimleme ve Birleştirme

Listenin bir kısmının seçilmesiyle oluşan yeni listelere bir alt liste (**sublist**) veya dilim (**slice**) denir. Bir listeden bir parçayı seçerek bu yeni listeleri oluşturma yöntemine dilimleme (**slicing**) denir.

- `A[i:]` komutu `A` listesinin `i` indisli elemanından başlayıp listenin sonuna kadar tüm elemanların listesini oluşturur.
- `A[:j]` komutu `A` listesinin başından başlayıp `j` indisli elemana kadar (j dahil **değil**) tüm elemanların listesini oluşturur.
- `A[:]` komutu `A` listesinin tüm elemanlarından oluşan bir liste oluşturur.
- `A[1:-1]` komutu `A` listesnin ilk ve son elemanı hariç diğer tüm elemanlarından oluşan bir liste oluşturur.
- `A[i:j:s]` komutu `A` listesnin `i` ve `j` indisli elemanları arasında (`j` dahil **değil**) `s` adım aralıklı olarak bir liste oluşturur.

Terminal
```
>>> A = range(15)
>>> A
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> A[:7]
[0, 1, 2, 3, 4, 5, 6]
>>> A[8:]
[8, 9, 10, 11, 12, 13, 14]
>>> A[:]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> A[2: 9: 2]
[2, 4, 6, 8]
```

```
>>> A[2: 9: 2][:2]
[2, 4]
```

Python'da bazen birden fazla listenin elemanlarıyla senkronize bir biçimde çalışmamız gerekebilir, böyle durumlarda `zip(list1, list2)` komutunu kullanırız. Bu komut her bir elemanı `(x1, y1)` `tuple` nesneleri olan bir list üretir, bu `tuple` nesnelerinin ilk elemanı `list1` ve diğer elemanı da `list2` listesinin elemanıdır.

```
1  from math import sin, pi
2
3  L1 = [5, 10, 15, 20]
4  L2 = [pi, 2*pi, 3*pi, 4*pi]
5  L3 = zip(L1, L2)
6
7  k = 0
8  while k < len(L1):
9      A = L3[k][0]
10     B = L3[k][1]
11     print "A=%2d, B=%7.4f, A*sin(B)=%9.2e" % (A, B, A*sin(B))
12     k += 1
```

Bu programın çıktısı aşağıdaki gibi olacaktır. Burada `zip()` komutunu kullanmadan daha basit şekilde de bu çıktıyı alabilirdik. Bunun için 5. satırı silip 8 ve 9 numaralı satırları sırasıyla `A = L1[k]` ve `B = L2[k]` olarak değiştiririz, böylece `zip()` komutunu kullanamdan da ayı işi yapmış oluruz.

Terminal

```
Terminal > python list2.py
A= 5, B= 3.1416, A*sin(B)= 6.12e-16
A=10, B= 6.2832, A*sin(B)=-2.45e-15
A=15, B= 9.4248, A*sin(B)= 5.51e-15
A=20, B=12.5664, A*sin(B)=-9.80e-15
```

Döngü Yapısı

Bir `list` veya `tuple` nesnesinin tüm elemanlarına aynı işlemi uygulayamak için `for` döngülerinden faydalanırız. Bu döngü çeşidini bir örnek üzerinde çalışalım. Farz edelim ki başarı olasılığı p (dolayısıyla başarısızlık olasılığı $1-p$) olsun. Bu durumda n deneme yapalım ve her bir denemenin sonucu önceki denemelerden bağımsız olsun. Bu n deneme sonucunda x tane başarı (dolayısıyla $n-x$ tane başarısızlık) elde etme olasılığımız

$$B(x,n,p) = \frac{n!}{x!(n-x)!}p^x(1-p)^{n-x}$$

formülü ile verilir, bu formüle Binom dağılımı denir. Bir madeni parayı art arda fırlattığımızı düşünelim, tura gelmesi olayını başarı kabul edersek bu durumda bu olayın gerçekleşme olasılığı $p = 1/2$ olacaktır. Örneğin beş denemede iki defa tura gelme olasılığı $B(2,5,0.5)$ değerine eşittir. Amacımız $n = 10,20,30,40,50$ deneme sonucunda başarı olasılıklarını hesaplayıp

yazdıran bir program yazmaktır. Burada $n!$ ifadesi n doğal sayısının **faktöriyel** değeridir ve bu hesaplamayı `math` modülü içinde tanımlanmış `math.factorial()` fonksiyonu yardımıyla yapabiliriz.

```
from math import factorial as fact

nlist = range(10, 51, 10) #[10, 20, 30, 40, 50]
p = 0.5
x = 2.0

for n in nlist:
    B = fact(n)/(fact(x)*fact(n-x))*p**x*(1-p)**(n-x)
    print "%d deneme -> %5.3e ihtimal" % (n, B)
```

Programda kullandığımız `for` döngüsü adımlarını açıklayalım. Döngüyü `for a in L` biçiminde başlatıyoruz, bunun anlamı aşağıda farklı paragraf hizasından başlayan tüm satırlar `L` listesinin her bir elemanı için ayrı ayrı çalıştırılacak demektir. Daha sonra iki nokta üst üste koyup altında hizalanmış biçimde hesaplamaları ve komutları yerleştiriyoruz. Bu programın çıktısı aşağıdaki gibi olacaktır.

Terminal

```
Terminal > for1.py
10 deneme -> 4.395e-02 ihtimal
20 deneme -> 1.812e-04 ihtimal
30 deneme -> 4.051e-07 ihtimal
40 deneme -> 7.094e-10 ihtimal
50 deneme -> 1.088e-12 ihtimal
```

Bu programı daha kompakt ve verimli bir biçimde aşağıdaki gibi de kodlayabiliriz, bu da aynı çıktıyı üretecektir.

```
from math import factorial as fact

p = 0.5; x = 2.0

for n in xrange(10, 51, 10):
    B = fact(n)/(fact(x)*fact(n-x))*p**x*(1-p)**(n-x)
    print "%d deneme -> %5.3e ihtimal" % (n, B)
```

Burada kullandığımız `xrange()` fonksiyonu bu örnekte aynı işi yapmasına rağmen aslında `range` fonksiyonundan oldukça farklıdır. `range()` fonksiyonu istenilen değerleri içeren bir liste oluşturup bellekte tutarken `xrange()` fonksiyonu açıkça bir liste oluşturup bellekte yer kaplamaz, döngünün her bir adımında sıradaki elemanını çıktı olarak döndürür. Dolayısıyla döngülerde bu fonksiyonun kullanımı çok daha verimlidir, eğer oluşturulan listeyi daha sonra başka işlemlerimizde kullanmayacaksak `xrange()` fonksiyonunu tercih etmeliyiz. Python 3 versiyonunda bulunan `xrange()` adında bir fonksiyon yoktur çünkü orada `range()` fonksiyonu Python 2'deki `xrange()` gibi davranır.

Terminal
```
>>> a = range(10, 51, 10)
>>> a
[10, 20, 30, 40, 50]
>>> type(a)
<type 'list'>
>>> b = xrange(10, 51, 10)
>>> b
xrange(10, 60, 10)
>>> type(b)
<type 'xrange'>
```

Her `for` döngüsünü bir `while` döngüsü olarak yeniden kodlayabiliriz. Bunun için önce `n = 0` biçiminde bir sayaç değişkeni tanımlarız, daha sonra `for a in L` ifadesini `while n < len(L)` ifadesiyle değiştiririz. Döngü komutlarında ise üzerinde işlem yapacağımız elemanı `x = L[n]` komutuyla alıp üzerinde gerekli işlemleri yaparız. Son olarak döngü içinde sayaç değerini arttırırız. Örneğin yukarıdaki program bir `while` döngüsü kullanılarak aşağıdaki gibi yazılabilir.

```
1  from math import factorial as fact
2
3  nlist = range(10, 51, 10) #[10, 20, 30, 40, 50]
4  p = 0.5
5  x = 2.0
6
7  i = 0
8  while i < len(nlist):
9      n = nlist[i]
10     B = fact(n)/(fact(x)*fact(n-x))*p**x*(1-p)**(n-x)
11     print "%d deneme -> %5.3e ihtimal" % (n, B)
12     i += 1
```

Python'da sıklıkla bir listenin öğelerini sayaç olarak kullanarak başka bir liste oluşturma işlemi yapılır ve dolayısıyla bunun için bir kısa yol vardır. Bu işlemi birkaç örnek üzerinde görelim, aşağıda görüleceği gibi bu işlemi köşeli parantezler içinde tek satırda yazılmış bir for döngüsü gibi değerlendirebiliriz. Bu işleme liste kapsama (**list comprehension**) denir.

Terminal
```
>>> from math import factorial
>>> N = [1, 3, 5, 7, 9]
>>> L = [factorial(n) for n in N]
>>> L
[1, 6, 120, 5040, 362880]
```

Bazen tablosal bir veriyi saklayıp yazdırmamız gerekebilir, bunu **iç içe** listeler kullanarak yapabiliriz. Yani elemanları yine birer liste olan bir liste oluşturabiliriz. Genelde bir tabloyu bellekte tutmak için tablonun her satırını ayrı birer liste olarak kaydedip daha sonra bu satırları eleman olarak kabul eden bir ana liste oluştururuz. Dolayısıyla bu son liste tüm tablo

verisini içerecektir. Örnek olarak Celsius ve Fahrenheit derece arası dönüşüm formülü olan

$$F = \frac{9}{5}C + 32$$

formülünü ele alalım. Bir sütunu Celsius, diğer sütunu da bunlara karşılık gelen Fahrenheit büyüklüklerini içeren bir tablo yazdıralım. Bunu aşağıdaki program ile sağlayabiliriz.

```
Clist = range(10, 36, 5)                        #[10, 15, 20, ... , 35]
Flist = [9.0/5*C + 32 for C in Clist]           #F degerleri
L = [[C, F] for C, F in zip(Clist, Flist)]      #ana tablo

for C, F in L:
    print C, F
```

Burada yazdırma aşamasında `C` ve `F` değerlerini aynı anda `L` listesinden nasıl aldığımıza dikkat edin. `L` listesinin her bir elemanı bir `[C, F]` ikilisi biçiminde, dolayısıyla döngünün her bir adımı böyle bir ikiliyi ziyaret ediyor ve iki değişkeni birden alıp yazdırıyor. Programın çıktısı aşağıdaki gibi olacaktır.

```
Terminal
Terminal > for4.py
10 50.0
15 59.0
20 68.0
25 77.0
30 86.0
35 95.0
```

2.3 If-Else Koşullu Yapısı

Bilgisayar programlamada sıklıkla programnın dallanması gerekir. Yani bir koşul sağlanırsa bir iş, başka bir koşul sağlanırsa başka bir iş yapılır. Matematiksel olarak **Heaviside** fonksiyonu olarak adlandırılan ve aşağıdaki gibi tanımlanam fonksiyonu bu duruma örnek göstere-biliriz.

$$H(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$$

Basit Dallanmalar

Python programlarında dallanmalar `if-else` blokları yardımıyla kodlanır. Aşağıdaki programda basit bir `if-else` yapısının kullanımı örnekleniyor.

```
x = 1E-15

if x < 0:
    H = 0
else:
    H = 1
```

```
7
8 print "x = %g\tH(x)=%g" % (x, H)
```

Şimdi yukarıdaki programda kullandığımız kodları açıklayalım. Programda dallanma 5-7 numaralı satırlarda oluşmaktadır. Bir `if-else` yapısı `if` kelimesini takip eden bir `bool` nesnesinden oluşan bir satırla başlar. Bunun altında bulunan ve daha sağdan hizalanmış satırlarda bulunan kodlar, eğer belirtilen bool nesnesi `True` değerine sahip ise çalıştırılır. Eğer bool nesnesi `False` ise bu blokta aynı hizada bulunan kodlar çalıştırılmaz. Bu bloktan sonra `if` kısmı ile aynı şekilde hizalanmış ilk satırda `else:` ifadesi ile karşılaşılırsa, belirtilen bool nesnesi değeri `False` olması durumunda bu satırın altındaki farklı hizalı satırlardaki kodlar çalıştırılır. `else` kısmı zorunlu değildir, eğer bu kısım yoksa bool nesnesi `True` ise `if` kısmında belirtilen kodlar çalıştırılır fakat bool nesnesi değeri `False` ise hiç bir şey yapılmaz. Örneğin aşağıdaki program da yukarıdaki ile aynı çıktıyı üretir.

```
1 x = 1E-15
2 H = 1
3
4 if x < 0:
5     H = 0
6
7 print "x = %g\tH(x)=%g" % (x, H)
```

Yukarıdaki programda `print` ifadesinde kullandığımız `\t` komutu Python'da özel bir karakterdir ve bir tab uzunluğunda boşluk vermek için kullanılır. Bu tür karakterlere kaçış (**escape**) karakterleri denir, diğer bir sık kullanılan kaçış karakteri `\n`'dir ve yeni bir satıra atlamayı sağlar. Yukarıdaki programlar aşağıdaki gibi bir sonuç üretir.

Terminal

```
Terminal > python ifelse1.py
x = 1e-15       f(x)=1
```

Bu fonksiyonun değerlerini farklı x değerleri için yazdıran bir program yazalım, bunu bir `for` döngüsü ile yapabiliriz.

```
1 xlist = [-1 + k*0.2 for k in range(10)]
2
3 for x in xlist:
4     if x < 0:
5         H = 0
6     else:
7         H = 1
8
9     print "x = %4.1f\tH(x)=%g" % (x, H)
```

Bu programın çıktısı aşağıdaki gibi olacaktır. Burada `print` ifadesinin bulunduğu satırı `for` döngüsü içinde yazdık, bundan dolayı her bir tekrarda yazdırma işlemi yapılıyor.

Terminal
```
Terminal > ifelse3.py
x = -1.0        H(x)=0
x = -0.8        H(x)=0
x = -0.6        H(x)=0
x = -0.4        H(x)=0
x = -0.2        H(x)=0
x =  0.0        H(x)=1
x =  0.2        H(x)=1
x =  0.4        H(x)=1
x =  0.6        H(x)=1
x =  0.8        H(x)=1
```

Yukarıda ele aldığımız Heaviside fonksiyonu matematikte yaygın olarak kullanılmasına rağmen bazı dezavantajları vardır, örneğin bu fonksiyon $x = 0$ noktasında süreksizdir. Bazı uygulamalarda bu fonksiyon yerine

$$H_\epsilon(x) = \begin{cases} 0, & x < -\epsilon \\ \frac{1}{2} + \frac{x}{2\epsilon} + \frac{1}{2\pi}\sin\left(\frac{\pi x}{\epsilon}\right), & -\epsilon \le x \le \epsilon \\ 1, & x > \epsilon \end{cases}$$

olarak tanımlanan H_ϵ fonksiyonu kullanılabilir. Bu fonksiyon yeterince küçük ϵ değerleri için Heaviside fonksiyonuna yakın davranır hem kendisi hem de türevi tüm reel eksende süreklidir. Böyle bir dallanmayı Python'da yapmak için iki koşullu yapı kullanabiliriz.

```
1  from math import sin, pi
2
3  x = 0.5
4  epsilon = 1E-12
5
6  if x < -epsilon:
7      H = 0
8  else:
9      if x <= epsilon:
10         H = 0.5 + x/(2*epsilon) + 1.0/(2*pi)*sin(pi*x/epsilon)
11     else:
12         H = 1
13
14 print "x = %4.1f\tH(x)=%g" % (x, H)
```

Çoklu Dallanmalar

Böyle çoklu dallanmalarla sık karşılaşıldığı için Pythpn'da bunlar için kısa bir yol vardır, bu da `elif` (**else if**) yapısıdır. Kullanımını aşağıdaki örnek üzerinde gözlemleyelim, bu örnek ile yukarıdaki program aynı şekilde çalışır.

```
1  from math import sin, pi
2
3  x = 0.5
```

```
4  epsilon = 1E-12
5
6  if x < -epsilon:
7      H = 0
8  elif -epsilon <= x <= epsilon:
9          H = 0.5 + x/(2*epsilon) + 1.0/(2*pi)*sin(pi*x/epsilon)
10 else:
11     H = 1
12
13 print "x = %4.1f\tH(x)=%g" % (x, H)
```

Yukarıdaki program için de bir döngü kullanarak bir tablo yazdıarbiliriz, bunu aşağıdaki kodlarla sağlarız.

```
1  from math import sin, pi
2
3  epsilon = 1E-12
4  xlist = [-.1 + k*0.02 for k in range(10)]
5
6  for x in xlist:
7      if x < -epsilon:
8          H = 0
9      elif -epsilon <= x <= epsilon:
10         H = 0.5 + x/(2*epsilon) + 1.0/(2*pi)*sin(pi*x/epsilon)
11     else:
12         H = 1
13     print "x = %4.1f\tH(x)=%g" % (x, H)
```

Bu kodlar aşağıdaki çıktıyı üretecektir. Görüleceği gibi H_ϵ fonksiyonu bire bir H ile aynı değerler vermiyor, ama oldukça yakın bir davranıştadır.

Terminal

```
Terminal > heaviside_eps3.py
x = -0.1        H(x)=0
x = -0.1        H(x)=0
x = -0.1        H(x)=0
x = -0.0        H(x)=0
x = -0.0        H(x)=0
x =  0.0        H(x)=0.5
x =  0.0        H(x)=1
x =  0.0        H(x)=1
x =  0.1        H(x)=1
x =  0.1        H(x)=1
```

Başka bir örnek olarak

$$N(x) = \begin{cases} 0, & x < 0 \\ x^2 \sin(x), & 0 \le x < 1 \\ 3 - x, & 1 \le x < 2 \\ 0, & x \ge 2 \end{cases}$$

fonksiyonu işlemini aşağıdaki dallanma örneği ile elde edebiliriz.

```
from math import sin, pi

x = pi/2

if x < 0:
    N = 0
elif 0 <= x < 1:
    N = x**2*sin(x)
elif 1 <= x <2:
    N = 3 - x
else:
    N = 0

print "x = %4.3f\tH(x)=%g" % (x, N)
```

Burada $(0,2]$ aralığı dışında fonksiyonun 0 değer aldığını kullanarak bu kodları biraz kısaltabiliriz.

```
from math import sin, pi

x = pi/2
N = 0

if 0 <= x < 1:
    N = x**2*sin(x)
elif 1 <= x <2:
    N = 3 - x

print "x = %4.3f\tH(x)=%g" % (x, N)
```

Bu iki programın da çıktısı aşağıdaki gibi olacaktır.

```
Terminal
Terminal > python ifelse5.py
x = 1.571        H(x)=1.4292
```

Satır İçi Dallanma

Programlarda `if-else` bloklarını çok sık kullandığımıdan Python'da bunun tek satırlık bir kısa yolu vardır. `H = (0 if x < 0 else 1)` komutu `H` değişkenine Heaviside fonksiyonunun sonucu olan sayıyı atar.

```
xlist = [-1 + k*0.2 for k in range(10)]
for x in xlist:
    H = (0 if x < 0 else 1)
    print "x = %4.1f\tH(x)=%g" % (x, H)
```