

R Programlama Dili

Kurs Notları

Kaan ASLAN

C ve Sistem Programcıları Derneği

Güncelleme Tarihi: 17/03/2014

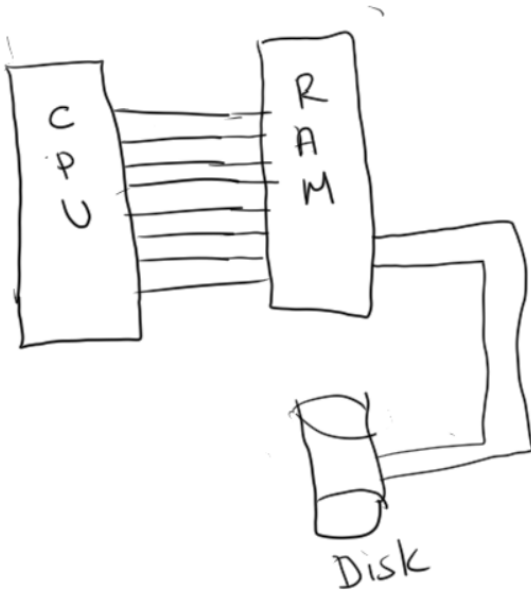
Bu kurs notları Kaan ASLAN tarafından yazılmıştır. Kaynak belirtilmek koşuluyla her türlü alıntı yapılabilir.

Temel Kavramlar

Bu bölüme R ortamına girmeden önce bazı temel kavramlarda bahsedilecektir

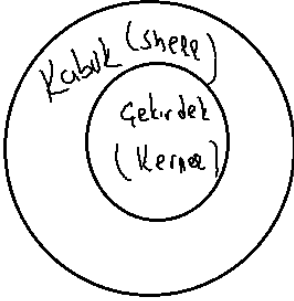
Temel Bilgisayar Mimarisi:

Bir bilgisayar mimarisi çok kabaca üç bileşenden oluşur: CPU, RAM ve Disk. Bilgisayarda tüm işlemlerin yapıldığı entegre devre biçiminde üretilmiş çiplere "mikroişlemci" ya da kavramsal olarak "CPU (Central Processing Unit)" denilmektedir. CPU elektriksel olarak RAM denilen bellek ile bağlantı halindedir. Böyle CPU'nun doğrudan elektriksel olarak bağlantı halinde olduğu belleklere kavramsal olarak "birincil bellek (primary memory)" ya da "ana bellek (main memory)" de denilmektedir. Ana bellekler entegre modülleri biçiminde üretilmektedir. Bu fiziksel modüllere RAM denir. CPU çalışırken sürekli RAM ile iletişim halindedir. Oradan bilgileri çeker, işler ve oraya geri yazar. Programlama dillerindeki değişkenlerin RAM'in içerisinde yaratılmaktadır. Bilgisayarın elektriğini kestiğimizde CPU durur. RAM'in içerisindeki bilgiler de kaybolur. Bilginin kalıcılığını sağlamak için diskler kullanılmaktadır. Diskler manyetik temeli elektromekanik biçimde olabileceği gibi tamamen yarı iletkenlerle (flash EPROM, SSD de denilmektedir) de gerçekleştirilebilmektedir. Elektromekanik olarak üretilmiş disklere kişisel bilgisayarlarda "hard disk" de denilmektedir. Bugün artık "flash EPROM" biçiminde üretilen SSD'ler hard disklerin yerini almaya başlamıştır. Ancak kavramsal olarak hard diskler, SSD'ler, CD ve DVD ROM'lar, memory sticklere "ikincil bellek (secondary memory)" denilmektedir.



İşletim Sistemi (Operating System):

İşletim sistemi makinenin donanımını yöneten, makine ile kullanıcı arasında arayüz oluşturan temel bir sistem programıdır. İşletim sistemi olmasa daha biz bilgisayarı açtığımızda bile bir şeyler göremeyiz. İşletim sistemleri iki katmandan oluşmaktadır. Çekirdek (kernel), makinenin donanımını yöneten kontrol kısmıdır. Kabuk (shell) ise kullanıcıyla arayüz oluşturan kısımdır. (Örneğin Windows'ta masaüstü kabuk görevindedir. Biz çekirdeği bakarak göremeyiz.) Tabii işletim sistemi yazmanın asıl önemli kısmı çekirdeğin yazımıdır. Çekirdek işletim sistemlerinin motor kısmıdır.



Bugün çok kullanılan bazı işletim sistemleri şunlardır:

- Windows
- Linux
- BSD'ler
- Mac OS X
- Android
- IOS
- Windows Mobile
- Solaris
- QNX

İşletim sistemlerinin bir kısmı açık kaynak kodlu bir kısmı da mülkiyete bağlıdır. Örneğin Linux, BSD açık kaynak kodlu olduğu halde Windows mülkiyete sahip bir işletim sistemidir. Mac OS X sistemlerinin çekirdeği açıktır (buna Darwin deniyor) ancak geri kalan kısmı kapalıdır.

Bazı işletim sistemleri diğer sistemlerin kodları değiştirilerek gerçekleştirilmiştir. Bazıları sıfırdan (orijinal kod tabanına sahip) yazılmışlardır. Orijinal kod tabanına sahip olan yani sıfırdan yazılmış olan işletim sistemlerinden bazıları şunlardır:

- Microsoft Windows
- Linux
- BSD'ler
- Solaris

Android Linux işletim sisteminin çekirdek kodları alınarak bazı modüllerin atılması ve mobil cihazlara yönelik bazı işlevselliklerin eklenmesiyle gerçekleştirilmiş bir sistemdir. Benzer biçimde IOS da Mac OS X kodlarından devşirilmiştir. Darwin çekirdeği Free BSD ve Mach isimli çekirdeklerin birleştirilmesiyle oluşturulmuş hibrit bir çekirdektir.

Çok kullanılan masaüstü işletim sistemlerinin mobil versiyonları da vardır. Örneğin Windows'un mobil versiyonu Windows CE (türevlerinden biri Windows Mobile)'dir. MAC OS X'in mobil versiyonu IOS'tur. Android'e Linux çekirdeğinin mobil hale getirilmiş bir versiyonu gözüyle bakılabilir.

Maalesef her mobil ortamın doğal program geliştirme ortamı farklıdır. Windows mobil aygıtlarının doğal geliştirme ortamı .NET'tir. Android'in Java'dır. IOS'un da Objective-C ve Swift'tir. Ancak C# ile Android (Monodroid ortamı) ve IOS'ta (Mono Touch ortamı) geliştirme yapılabilir.

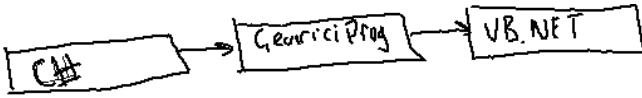
Bugün masaüstü (laptoplar da dahil olmak üzere) işletim sistemlerinde Windows %70-%80 arası bir kullanıma sahiptir. Mac OS X'in kullanım oranı %10 civarındadır. Linux'un gündelik yaşamda kişisel bilgisayar olarak kullanım oranı ise %1 civarındadır. Ancak server dünyasında UNIX türevi işletim sistemlerinin payları %60'ın üzerindedir. Yani UNIX/Linux sistemleri server dünyasında en çok kullanılan sistemlerdir. 2017 yılı itibarıyla Android %60, IOS ise %30 civarında civarıyla bir yaygınlığa sahiptir. Windows Mobile sistemlerinin kullanım oranı %2.5 civarındadır.

Gömülü Sistemler (Embedded Systems)

Asıl amacı bilgisayar olmayan fakat bilgisayar devresi içeren sistemlere genel olarak gömülü sistemler denilmektedir. Örneğin elektronik tartılar, biyomedikal aygıtlar, GPS cihazları, turnike geçiş sistemleri, müzik kutuları vs. birer gömülü sistemdir. Gömülü sistemlerde en çok kullanılan programlama dili C'dir. Ancak son yıllarda Raspberry Pi gibi, Banana Pi gibi, Orange Pi gibi güçlü ARM işlemcilerine sahip kartlar çok ucuzlamıştır ve artık gömülü sistemlerde de doğrudan kullanılabilir hale gelmiştir. Bu kartlar tamamen bir bilgisayarın işlevselliğine sahiptir. Bunlara genellikle Linux işletim sistemi ya da Android işletim sistemi yüklenir. Böylece gömülü yazılımların güçlü donanımlarda ve bir işletim sistemi altında çalışması sağlanabilmektedir. Örneğin Raspberry Pi'ya biz Mono'yu yükleyerek C#'ta program yazıp onu çalıştırabiliriz.

Çevirici Programlar (Translators), Derleyiciler (Compilers) ve Yorumlayıcılar (Interpreters)

Bir programlama dilinde yazılmış olan programı eşdeğer olarak başka bir dile dönüştüren programlara çevirici programlar (translators) denilmektedir. Çevirici programlarda dönüştürülmek istenen programın diline kaynak dil (source language), dönüşüm sonucunda elde edilen programın diline hedef dil (target/destination language) denilmektedir. Örneğin:



Burada kaynak dil C#, hedef dil VB.NET'tir.

Eğer bir çevirici programda hedef dil aşağı seviyeli bir dil ise (saf makina dili, arakod ve sembolik makine dilleri alçak seviyeli dillerdir) böyle çevirici programlara derleyici (compiler) denir. Her derleyici bir çevirici programdır fakat her çevirici program bir derleyici değildir. Bir çeviri programa derleyici diyebilmek için hedef dile bakmak gerekir. Örneğin arakodu gerçek makine koduna dönüştüren CLR bir derleme işlemi yapmaktadır. Sembolik makine dilini saf makina diline dönüştüren program da bir derleyicidir.

Bazı programlar kaynak programı alarak hedef kod üretmeden onu o anda çalıştırırlar. Bunlara yorumlayıcı (interpreter) denilmektedir. Yorumlayıcılar birer çevirici program değildir. Yorumlayıcı yazmak derleyici yazmaktan daha kolaydır. Fakat programın çalışması genel olarak daha yavaş olur. Yorumlayıcılarda kaynak kodun çalıştırılması için onun başka kişilere verilmesi gerekir. Bu da kaynak kod güvenliğini bozar.

Bazı diller yalnızca derleyicilere sahiptir (C, C++, C#, Java gibi). Bazıları yalnızca yorumlayıcılara sahiptir (R, PHP, Perl gibi). Bazılarının hem derleyicileri hem de yorumlayıcıları vardır (Basic gibi). Genel olarak belli bir alana yönelik (domain specific) dillerde çalışma yorumlayıcılar yoluyla yapılmaktadır. Genel amaçlar diller daha çok derleyiciler ile derlenerek çalıştırılırlar.

IDE (Integrated Development Environment)

Derleyiciler komut satırından çalıştırılan programlardır. Bir programlama faaliyetinde program editör denilen bir program kullanılarak yazılır. Diske save edilir. Sonra komut satırından derleme yapılır. Bu yorucu bir faaliyettir. İşte yazılım geliştirmeyi kolaylaştıran çeşitli araçları içerisinde barındıran (integrated) özel yazılımlara IDE denilmektedir. IDE'nin editörü vardır, menüleri vardır ve çeşitli araçları vardır. IDE'lerde derleme yapılırken derlemeyi IDE yapmaz. IDE derleyiciyi çalıştırır. IDE yardımcı bir araçtır, mutlak gerekli bir araç değildir.

Microsoft'un ünlü IDE'sinin ismi "Visual Studio"dur. Apple'ın "X-Code" isimli IDE'si vardır. Bunların dışında başka şirketlerin malı olan ya da "open source" olan pek çok IDE mevcuttur. Örneğin "Eclipse" ve "Netbeans" yaygın kullanılan cross-platform "open source" IDE'lerdir. Linux altında Mono'da "Mono Develop" isimli bir IDE tercih edilmektedir. Bu IDE'nin Windows versiyonu da vardır.

R ortamı için RStudio isimli şirket tarafından geliştirilmiş olan RStudio IDE'si yaygın olarak kullanılmaktadır. RStudio hem Windows, hem Mac OS X hem de Linux sistemlerinde kullanılabilen "cross platform" bir IDE'dir. Açık kaynak kodlu ve ticari sürümleri vardır. Kursumuzda RStudio IDE'sinin açık kaynak kodlu sürümü kullanılacaktır.

Mülkiyete Sahip Yazılımlar, Özgür ve Açık Kaynak Kodlu Yazılımlar

Yazılımların çoğu bir firma tarafından ticari amaçla yazılırlar. Bunlara mülkiyete bağlı yazılım (proprietary) denilmektedir. 1980'li yılların ortalarında Richard Stallman tarafından "Özgür Yazılım (Free Software)" hareketi başlatılmıştır. Bunu daha sonra "Open Source (Açık Kaynak Kod)" ve türevleri izlemiştir. Bunların çoğu birbirine benzer lisanslara sahiptir. Özgür yazılımın ve açık kaynak kodlu yazılımın temel prensipleri şöyledir:

- Program yazıldığında yalnızca çalıştırılabilen (executable) dosyalar değil, kaynak kodlar da verilir
- Kaynak kodlar sahiplenilemez.
- Bir kişi bir kaynak kodu değiştirdiğinde ve geliştirdiğinde o da ürününü açmak zorundadır.
- Program istenildiği gibi dağıtılıp kullanılabilir.

Bugün Linux dünyasındaki ürünlerin çoğu bu kapsamdadır. Biz bir yazılımı ya da bileşeni kullanırken onun lisansına dikkat etmeliyiz.

Bugün özgür yazılım ve açık kaynak kod hareketi çok ilerlemiştir. Neredeyse popüler pek çok ürünün açık kaynak kodlu bir versiyonu vardır.

Dil (Language) Kavramı

Dil karmaşık bir olgudur. Tek bir cümleyle tanımını yapmak pek mümkün değildir. Fakat kısaca "iletişim için kullanılan semboller kümesidir" denebilir. Bir dilin tüm kurallarına gramer denir. Gramerin en önemli iki alt alanı sentaks (syntax) ve semantik (semantic)'tir. Bir dili oluşturan en yalın öğelere atom ya da sembol (token) denilmektedir. Örneğin doğal dillerde atomlar sözcüklerdir.

Bir olgunun dil olabilmesi için asgari sentaks ve semantik kurallara sahip olması gerekir. Sentaks doğru dizilime ve doğru yazıma ilişkin kurallardır. Örneğin:

"I school to am going"

Burada İngilizce için bir sentaks hatası vardır. Sözcükler doğrudur fakat dizilimleri yanlıştır. Örneğin:

"Herkez çok mutluydu"

Burada da bir sentaks hatası vardır. Türkçe'de "herkez" biçiminde bir sözcük (yani sembol) yoktur. Örneğin:

"if a > 10)"

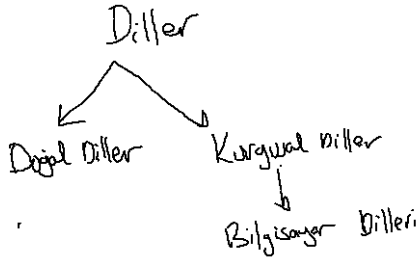
Burada da R'ca bir sentaks hatası yapılmıştır.

Semantik doğru yazılmış ve dizilmiş öğelerin ne anlam ifade ettiğine ilişkin kurallardır. Yani bir şey doğru yazılmıştır fakat ne anlama gelmektedir? Örneğin:

" I am going to school"

sentaks bakımından geçerlidir. Fakat burada ne demek istenmiştir? Bu kurallara semantik denilmektedir.

Diller doğal ve kurgusal (ya da yapay) olmak üzere ikiye ayrılabilir. Doğal dillerde sentaksın tam bir fomülasyonu yoktur. Kurgusal diller insanlar tarafından formüle edilebilecek biçimde tasarlanmış dillerdir. Bilgisayar dilleri kurgusal dillerdendir.



Kurgusal dillerde istisnalar ya yoktur ya da çok azdır. Sentaks ve semantik tutarlıdır. Doğal dillerde pek çok istisna vardır. Doğal dillerin zor öğrenilmesinin en önemli nedenlerinden birisi de istisnalardır.

Bilgisayar Dilleri ve Programlama Dilleri

Bilgisayar bilimlerinde kullanılan dillere bilgisayar dilleri (computer languages) denir. Bir bilgisayar dilinde akış varsa ona aynı zamanda programlama dili de (programming language) denilmektedir. Örneğin HTML bir bilgisayar dilidir. Fakat HTML'de bir akış yoktur. Bu nedenle HTML bir programlama dili değildir. HTML'de de sentaks ve semantik kuralları vardır. Oysa R'da bir akış da vardır. R bir programlama dilidir.

Programlama Dillerinin Sınıflandırılması

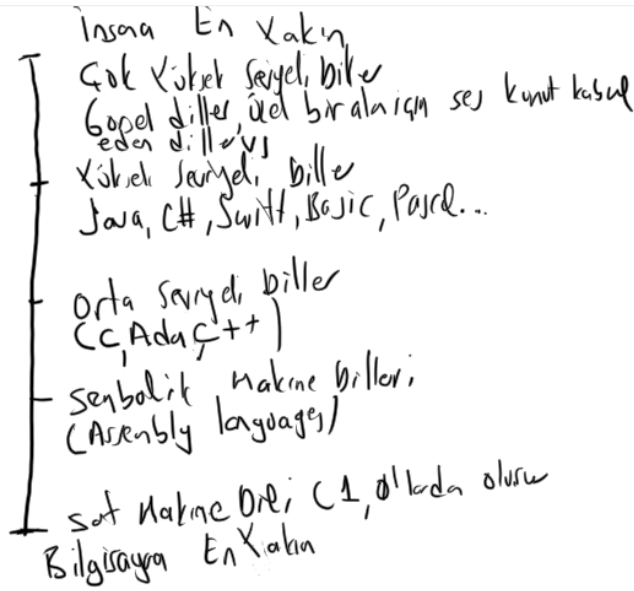
Programlama dilleri yaygın olarak üç biçimde sınıflandırılmaktadır:

- 1) Seviyelerine Göre sınıflandırma
- 2) Kullanım Alanlarına Göre Sınıflandırma
- 3) Programlama Modeline Göre Sınıflandırma

Seviyelerine Göre Sınıflandırma: Seviye (level) bir programlama dilinin insan algısına yakınlığının bir ölçüsüdür. Yüksek seviyeli diller kolay öğrenilebilen insana yakın dillerdir. Alçak seviyeli diller bilgisayara yakın dillerdir. Olabilecek en alçak seviyeli diller 1'lerden 0'lardan oluşan saf makine dillerdir. Bunun biraz yukarısında sembolik makina dilleri (assembly languages) bulunur. Biraz daha yukarıda orta seviyeli diller bulunmaktadır. Daha yukarıda ise yüksek seviyeli, en yukarıda da "çok yüksek seviyeli diller" vardır. Örneğin:

- Java, C#, Pascal, Basic , R "yüksek seviyeli" dillerdir.
- C "orta seviyeli" bir dildir.
- C++ orta ile yüksek seviye arasındadır.

Tabii aslında dillerin seviyelerini sürekli çizgi üzerinde noktalar biçiminde düşünebiliriz. İki yüksek seviyeli dil bu çizgide aynı yerde bulunmak zorunda değildir. Seviyelerine göre dilleri sınıflandırırken genellikle bir seviye çizgisinden faydalanılır:



Kullanım Alanlarına Göre Sınıflandırma: Bu sınıflandırma biçimi dillerin hangi amaçla daha çok kullanıldığına yöneliktir. Tipik sınıflandırma şöyle yapılabilir:

- Bilimsel ve Mühendislik Diller: C, C++, Java, C#, Fortran, Pascal, R, Matlab gibi...
- Veritabanı Yoğun İşlemlerde Kullanılan Diller: SQL, Foxpro, Clipper, gibi...
- Web Dilleri: PHP, C#, Java
- Animasyon Dilleri: Action Script gibi...
- Yapay Zeka Dilleri: Lisp, Prolog, C, C++, C#, Java, R gibi...
- Sistem Programlama Dilleri: C, C++, Sembolik Makina Dilleri
- Genel Amaçlı Diller: C, C++, Pascal, C#, Java, Python, Basic gibi...
- Özel Amaçlı (Domain Specific) Diller: Bunlar genel amaçlı dillerin aksine belli bir alanda kullanılmak üzere tasarlanmış dillerdir. Örneğin sed, awk ve R dillerini by gruba dahil edebiliriz.

Programlama Modeline Göre Sınıflandırma: Program yazarken hangi modeli (paradigm) kullandığımıza yönelik sınıflandırmadır. Altprogramların birbirlerini çağırmasıyla program yazma modeline "prosedürel programlama modeli (procedural programming paradigm)" denilmektedir. Bir dilde yalnızca alt programlar oluşturabiliyorsak, sınıflar oluşturamıyorsak bu dil için "prosedürel programlama modeline uygun olarak tasarlanmış" bir dil diyebiliriz. Örneğin klasik Basic, Fortran, C, Pascal prosedürel dillerdir. Bir dilde sınıflar varsa ve program sınıflar kullanılarak yazılıyorsa böyle dillere "nesne yönelimli diller (object oriented languages)" denilmektedir. Eğer program formül yazar gibi yazılıyorsa bu modelde fonksiyonel model (functional paradigm), bu modeli destekleyen dillere de fonksiyonel diller denilmektedir. Bazı dillerde program görsel olarak fare hareketleriyle oluşturulabilmektedir. Bunlara görsel diller denir. Bazı diller birden fazla programlama modelinin kullanılmasına olanak sağlayacak biçimde tasarlanmıştır. Bunlara da çok modellenli diller (multiparadigm languages) denilmektedir. C++, Swift ve R gibi diller çok modellenlidir. Örneğin R'da biz hem fonksiyoneli hem prosedürel hem de nesne yönelimli teknikleri kullanabilmekteyiz.

Tüm bunlar ışığında R için şunları söylenebiliriz: R yüksek seviyeli, matematiksel ve istatistiksel veri analizinde, bilimsel ve mühendislik uygulamalarda ve yapay zeka uygulamalarında kullanılabilen kullanılabilen fonksiyonel prosedürel ve nesne yönelimli bir programlama dilidir.

1.9. Klavyedeki Karakterlerin İngilizce İsimleri

Programlama faaliyetlerinde klavye üzerinde gördüğünüz pek çok karakter şu ya da bu biçimde kullanılabilmektedir. Aşağıda bu karakterlerin İngilizce karşılıkları verilmiştir:

Sembol	İsim
+	plus

-	minus, hyphen, dash
*	asterisk
/	slash
\	back slash
.	period, dot
,	comma
:	colon [ko:lın]
;	semicolon
"	double quote [dabil kvot]
'	single quote
(...)	paranthesis [pıran(th)isi:s] left, right, opening, closing
[...]	(square) bracket left, right, opening, closing
{...}	brace [breys] left, right, opening, closing
=	equal sign [i:kvıl sayn]
&	ampersand
~	tilda
@	at
<...>	less than, greater than, angular bracket
^	caret [k(ea)rıt]
	pipe [payp]
_	underscore [andırsko:r]
?	question mark
#	sharp, number sign, hashtag
%	percent sign [pörsınt sayn]
!	exclamation mark [eksklemeyşın mark]
\$	dollar sign [dalır sayn]
...	ellipsis [elipsis]

R Nasıl Bir Dil ve Ortamdır?

R istatistiksel ve matematiksel veri analizinde kullanılan fonksiyonel, prosedürel, nesne yönelimli bir programlama dilidir. R özellikle istatistikçiler ve veri madencileri tarafından tercih edilmektedir. R GPL lisansına sahip olan açık kaynak kodlu (open source) bir yazılımdır. Dolayısıyla aynı zamanda bedavadır. R Programlama Dilinin ve paketlerinin gerçekleştirilmesinde ağırlıklı C Programlama Dili kullanılmıştır. Ancak Fortran'dan da faydalanılmıştır. Bazı paketler ise doğrudan R'ın kendisi ile yazılmıştır.

R yorumlayıcı temelli (interpreted) bir programlama dilidir. Hem komut satırlı karşılıklı etkileşimi (interactive) hem de script tabanlı bir çalışmayı destekler. Çeşitli konulara yönelik onlarca pakete (kütüphaneye) sahiptir. R "cross platform" bir ortamdır. Hem Windows sistemlerinde, hem Linux sistemlerinde hem de Mac OS X sistemlerinde yüklenerek çalıştırılabilmektedir. Güçlü grafik özelliklere sahiptir.

R Auckland Üniversitesinde Ros Ihaka ve Robert Gentleman tarafından geliştirilmiştir. R'in geliştirilmesine 1992 yılında başlanmıştır. Stabil ilk versiyonu 1995 yılında oluşturulmuştur. R aslında John Chambers tarafından 70'li yılların ortalarında geliştirilmiş olan S isimli programlama dilinin biraz daha geliştirilmiş bir biçimidir. Scheme Dilinden de biraz etkilenmiştir.

R'in geliştirilmesi ve sürdürümü "R Foundations" isimli kurum tarafından yapılmaktadır (www.r-project.org). R Foundations "Free Software Foundation (FSF)" kurumunun bir parçası durumundadır. Bu anlamda bir GNU dağıtımı olduğu söylenebilir.

R Ortamının Kurulumu

R'in kurulumu oldukça basittir. Kurulum dosyası www.cran.r-project.org adresinden indirilebilir. R'in kurulumu ile birlikte kullanıcıya hem bir yorumlayıcı hem de karşılıklı etkileşimli bir ortam sunulmaktadır. Linux sistemlerinde R komut satırından paket yöneticileriyle kurulabilir. Örneğin Ubuntu ve türevlerinde komut satırından aşağıdaki komutla doğrudan tban (base) R sistemini kurabiliriz:

```
sudo apt-get install r-base r-base-dev
```

Tabii Linux sistemlerinde istenirse yine kurulum paketi indirilerek de kurulum yapılabilir. Mac OS X sistemlerinde kurulum yine çok kolaydır. İlgili dmg dosyası indirilerek kurulum yapılabilir.

Ayrıca R için bazı IDE benzeri ortamlar da oluşturulmuştur. Bunların bazıları ücretlidir. RStudio isimli şirket tarafından geliştirilmiş olan RStudio isimli IDE'nin ücretsiz ve ücretli biçimleri Windows, Linux, Mac OS X sistemlerinde kullanılabilir. RStudio dışında R için başa IDE'ler de vardır. Örneğin Microsoft'un Visual Studio IDE'sine yine Microsoft tarafından hazırlanmış olan R için eklenti (Visual Studio Tools for R) kurulabilmektedir. Ancak biz kursumuzda RStudio IDE'sini kullanacağız. RStudio IDE'si <https://www.rstudio.com/products/rstudio/download/> adresinden indirilip kurulabilir. Ancak RStudio kurulmadan önce R ortamının kurulmuş olması gerekmektedir. Bunun dışında R için ismine "RStudio Server" denilen bir "server" program da oluşturulmuştur. Bu server sayesinde biz Web tarayıcılarıyla RStudio ortamını da kullanabiliriz.

2. R Programla Dilinin ve Ortamının Temel Özellikleri

Bu bölümde R Programlama Diline ve ortamına hızlı bir giriş yapacağız. Sonraki bölümlerde R Programlama Dili daha teknik biçimde ele alınacaktır.

R'da Değişkenler, Vektörler ve Atama İşlemleri

R dizisel (vektörel) tabanlı bir dildir. R'da değişkenler birden fazla değeri tutabilen vektör biçimindedir. İleride de göreceğimiz gibi R'da vektörlerin elemanlarına indeksleme yoluyla erişilebilmektedir. Vektör elemanları aynı türden olabilir ya da farklı türlerden olabilir. R'da elemanları aynı türden olan vektörlere "atomik vektörler", elemanları farklı türden olabilen vektörlere ise "listeler (lists)" denilmektedir.

Pek çok programlama dilinde bir değişken tek bir değer tutar. Bir değişkenin tek bir değeri tuttuğu durum aslında R'da tek elemanlı bir vektör gibi düşünülmelidir. R'da C, Java, C# gibi dillerdeki gibi bir bildirim kavramı yoktur. R dinamik türlü (dynamic typed) bir dildir. Yani değişkenlerin türleri sabit değildir. Onlara atanan değerle birlikte onların türleri de değişmektedir.

R'da atama işlemi için diğer pek çok dildeki gibi ' = ' operatörü kullanılabilir. Ancak R'da atama için R'a özgü ok operaörü de kullanılabilir. Biz kursumuzda atama işlemlerini bu operatörle yapacağız. Örneğin:


```
a <- 10
```

Atama işleminde atamanın yapıldığı değişken sağ tarafta da olabilir. Bu durumda okun yönü de değiştirilmelidir:

```
10 -> a
```

R büyük harf küçük harf duyarlılığı olan bir dildir. Yani değişken isimlerinde büyük harflerle küçük harfler tamamen farklı karakterlermiş gibi ele alınmaktadır. (Örneğin count isimli değişken ile Count isimle değişken aynı değişken değildir.) Değişken isimlendirme kuralları pek çok programlama dilinde olduğu gibidir:

- Değişken isimleri sayısal karakterlerle başlatılamazlar. Ancak alfabetik karakterlerle başlatılıp sayısal karakterlerle devam ettirilebilirler.

- Değişken isimleri boşluk karakterlerini içeremez.

- Değişken isimleri operatör sembollerini içeremez. Ancak değişken isimlendirmesinde '.' karakteri kullanılabilir.

- Değişken isimlendirmesinde UNICODE alfabetik karakterler kullanılabilir. Örneğin değişkenlerimize Türkçe isimler verebiliriz.

R'da pek çok hazır fonksiyon bulunmaktadır. Fonksiyonlar bir işi yapan alt programlardır. Fonksiyon çağırmanın genel biçimi şöyledir:

```
fonksiyon_ismi([argüman listesi])
```

R'da vektör oluşturmak için c isimli fonksiyon kullanılır. Vektörün elemanları bu fonksiyona argüman olarak verilmelidir. Örneğin:

```
> a <- c(10, 20, 30)
> a
[1] 10 20 30
```

R komut satırında bir değişkenin ismini yazıp ENTER tuşuna bastığımızda değişkenin içerisindeki değerin yazdırıldığına dikkat ediniz.

Sabitler de bir elemanlı vektör olarak düşünülmelidir. Bu durumda örneğin:

```
a <- 10
```

ile

```
a <- c(10)
```

işlevsel olarak eşdeğerdir. c fonksiyonunda argüman olarak vektörler de kullanılabilir. Örneğin:

```
> a <- c(1, 2, 3, 4, 5)
> a <- c(1, 2, 3)
> b <- c(7, 8, 9)
> c <- c(a, 4, 5, 6, b)
> c
[1] 1 2 3 4 5 6 7 8 9
```

Vektör elde etmenin diğer bir yolu ':' operatörünü kullanmaktır. Bu operatör iki operandlı araek bir operatördür. Sol tarafındaki değerden başlayarak sağ tarafındaki değere kadar (kapalı aralık) birer artırılmış değerlerden oluşan bir vektör elde etmemizi sağlar. Örneğin:

```
> a <- 1:10
> a
[1] 1 2 3 4 5 6 7 8 9 10
> b <- 1:10
> a * b
[1] 1 4 9 16 25 36 49 64 81 100
```

Aslında soldaki değer sağdaki değerden küçük olmak zorunda değildir. Yani vektör ters sırada da elde edilebilir:

```
> a <- 10:1
> a
[1] 10 9 8 7 6 5 4 3 2 1
```

‘:’ operatörünün operandlarının tamsayı olması gerekmez. Fakat her zaman 1 artırım yapılmaktadır:

```
> a <- 1.5:10
> a
[1] 1.5 2.5 3.5 4.5 5.5 6.5 7.5 8.5 9.5
```

‘:’ operatöründe artırım miktarı her zaman birdir bunu değiştiremeyiz. Ancak seq isimli fonksiyon bize istediğimiz artımı vermeye olanak sağlar. Örneğin:

```
> a <- seq(1, 10, 2)
> a
[1] 1 3 5 7 9
```

R’da default argüman kullanımı vardır. Yani bazı fonksiyonlarda biz bazı değerleri belirtmediğimizde onlar için default birtakım değerler alınmaktadır. Örneğin seq fonksiyonunda biz artırımı belirtmek zorunda değiliz. Bu durumda artırım default olarak 1 alınmaktadır. Örneğin:

```
> a <- seq(1, 10)
> a
[1] 1 2 3 4 5 6 7 8 9 10
```

Bir vektörün uzunluğu (yani eleman sayısı) length fonksiyonuyla elde edilebilir. Örneğin:

```
> x <- seq(-3.14,+3.14, 0.01)
> length(x)
[1] 629
```

R’da vektör oluşturma’nın diğer bir yolu da vector isimli fonksiyonu kullanmaktır. vektör fonksiyonunda biz vektör elemanlarının türünü (modunu) ve uzunluğunu veririz. Örneğin:

```
> v <- vector("numeric", 10)
> v
[1] 0 0 0 0 0 0 0 0 0 0
```

Bu biçimde yaratılan vektörün elemanlarının başlangıçta sıfır değerinde olduğuna dikkat ediniz. vector fonksiyonu başka konular içerisinde daha ayrıntılı olarak ele alınacaktır.

Vektörlerin Operatörlerle İşleme Sokulması

Bir işleme yol açan ve işlem sonucunda belli bir değer üretilmesini sağlayan +, -, *, /, >, < gibi sembollere operatör denilmektedir. R’da değişkenler (yani vektörler) aritmetik operatörlerle ve karşılaştırma operatörleriyle işlemlere sokulabilirler. Vektörler operatörlerle işleme sokulduğunda vektörün karşılıklı elemanları işleme sokulmuş olur. Örneğin:

```
> a <- c(1, 2, 3)
```

```
> b <- c(4, 5, 6)
> c <- a + b
> c
[1] 5 7 9
```

Burada $a + b$ işleminin sonucunda 3 elemanlı bir vektör elde edilmiştir. Elde edilen vektörün elemanları $[1+4, 2+5, 3+6]$ değerlerinden oluşmaktadır. Pekiyi bu işlemlerde vektörler aynı uzunlukta değilse ne olur? İşte bu durumda R'da "döngüye sokma (recycling)" denilen bir kural uygulanmaktadır. Bu kurala göre uzunluğu kısa olan vektör yinelenerek uzun olanla aynı uzunluğa getirilerek işleme sokulur. Örneğin:

```
> a <- c(1, 2, 3, 4, 5, 6)
> b <- c(1, 2)
> c <- a + b
> c
[1] 2 4 4 6 6 8
```

Burada kısa olan vektördeki 1 2 değerleri 1 2 1 2 1 2 durumuna getirilerek işleme sokulmuştur.

```
[1, 2, 3, 4, 5, 6] + [1, 2, 1, 2, 1, 2] = [2, 4, 4, 6, 6, 8]
```

Ancak "döngüye sokma" işleminde büyük olan vektörün uzunluğu küçük olanın uzunluğunun katı değilse uyarı (warning) oluşur. Tabii işlem yine yapılır. Örneğin:

```
> a <- c(1, 2, 3, 4, 5)
> b <- c(1, 2)
> c <- a + b
Warning message:
In a + b : longer object length is not a multiple of shorter object length
```

Örneğin:

```
> a <- c(1, 2, 3, 4, 5)
> b <- 10
> c <- a * b
> c
[1] 10 20 30 40 50
```

Burada görüldüğü gibi a'nın her elemanı 10 ile çarpılmıştır. Aslında burada yine "döngüye sokma" kuralının işletildiğine dikkat ediniz.

Bool Türü ve Değerleri

R'da doğru ya da yanlış olabilen bir bool türü vardır. Doğru değer TRUE ile yanlış değer de FALSE ile belirtilir. Ancak bu değerler kısaca T ve F harfleriyle de gösterilebilmektedir. Örneğin:

```
> a <- T
> a
[1] TRUE
> a <- TRUE
```

Pek çok programlama dilinde olduğu gibi R'da da karşılaştırma işlemleri için $>$, $<$, $>=$, $<=$, $=$ ve $!=$ operatörleri kullanılmaktadır. R'da karşılaştırma operatörleri bool vektör üretmektedir. Örneğin:

```
> a <- c(10, 20, 30, 40)
> b <- a > 20
> b
[1] FALSE FALSE TRUE TRUE
```

Şüphesiz döngüye sokma işlemi karşılaştırma işlemlerinde de geçerlidir. Örneğin:

```
> a <- c(-4, 3, -7, -9, 4, 7)
> b <- c(0, 2)
> c <- a > b
> c
[1] FALSE TRUE FALSE FALSE TRUE TRUE
```

R'da bool türü tıpkı C ve C++'ta olduğu gibi aritmetik işlemlere sokulabilir. Bool türü aritmetik işlemere sokulduğunda TRUE için 1, FALSE için 0 olarak işleme girer. Örneğin:

```
> 3 + T
[1] 4
> b <- c(T, F, T, T, T)
> sum(b)
[1] 4
```

2.4. Stringler ve Stringler Türünden Vektörler

R'da string oluşturmak için tek tırnak ya da çift tırnak kullanılır. Tek tırnak ya da çift tırnak arasında bu bağlamda hiçbir fark yoktur. Örneğin:

```
> "Kaan Aslan"
[1] "Kaan Aslan"
> 'Kaan Aslan'
[1] "Kaan Aslan"
```

String türünden vektörler de söz konusu olabilir. Örneğin:

```
> cities = c("Ankara", "İzmir", "Adana", "Bursa", "Eskişehir")
> cities
[1] "Ankara"      "İzmir"       "Adana"       "Bursa"       "Eskişehir"
```

Boş string söz konusu olabilir. Örneğin:

```
> str = c("Ali", "", "Veli")
> str
[1] "Ali" "" "Veli"
```

İki string + operatörü ile toplanamaz. (Bazı dillerde bu yapılabilmektedir) paste isimli, fonksiyon iki yazıyı birleştirmekte kullanılmaktadır. Örneğin:

```
> a <- "Ali"
> b <- "Veli"
> c <- paste(a, b)
> c
[1] "Ali Veli"
```

Üs, Mod ve Tamsayı Bölme Operatörleri

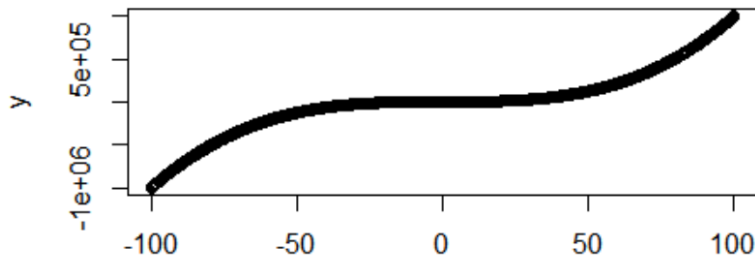
R'da üs alma işlemi için ^operatörü kullanılmaktadır. Örneğin:

```
> a <- 2 ^ 6
> a
[1] 64
```

Örneğin:

```
> x <- seq(-100, 100, 0.1)
```

```
> y <- x ^ 3
> plot(x, y)
```



R'da bölümden elde edilen kalan işlemi için %% operatörü kullanılmaktadır. Örneğin:

```
> a <- 1:10
> b <- a[a %% 3 == 0]
> b
[1] 3 6 9
```

%% operatörü R'da tamsayı bölme anlamına gelir. Yani bu operatörde önce bölme işlemi yapılır daha sonra elde edilen değerin noktadan sonraki kısmı atılır. Örneğin:

```
> 10 / 3
[1] 3.333333
> 10 %% 3
[1] 3
```

Örneğin:

```
> 3.8 %% 2
[1] 1
```

Fonksiyon Kavramı ve Fonksiyonların Çağırılması

Belli bir işlemi gerçekleştiren öğelere R'da fonksiyon denilmektedir. Fonksiyonlar diğer dillerdeki alt programlara (prosedürlere, ya da fonksiyonlara) benzemektedir. Bir fonksiyonun oluşturulması (buna bildirilmesi de diyebiliriz) ve çağırılması biçiminde iki eylem vardır. Fonksiyonların nasıl oluşturulduğu ileride ele alınacaktır. Biz burada şimdilik zaten oluşturulmuş olan fonksiyonların çağırılması üzerinde duracağız. R fonksiyonel (functional) bir programlama modeline sahip olduğu için R'da fonksiyon kavramı diğer prosedürel ve nesne yönelimli dillerden farklıdır. R'da fonksiyonlar birinci sınıf vatandaşdır (first class citizen). Bu eseri fonksiyonların R'da normal nesneler gibi işleme sokulabileceğini belirtir. Diğer fonksiyonel dillerde olduğu gibi aslında R'da operatörler de birer fonksiyon statüsündedir.

Bir fonksiyon birtakım argümanları alarak birtakım işlemleri gerçekleştirir ve duruma göre onu çağıran kişiye bir değer verir. Buna fonksiyonun geri dönüş değeri (return value) denilmektedir. Fonksiyon (...) operatörü ile çağırılır. Fonksiyonun argümanları aralarına ',' atomu konularak parantez içerisinde bildirilir. Örneğin:

```
sum(v)
foo(10, 20, 30)
```

Fonksiyon çağırmanın genel biçimi şöyledir:

```
<Fonksiyon ismi>([argüman listesi])
```

Bazı Temel Fonksiyonlar

Yukarıda da belirttiğimiz gibi R'in standart kütüphanesinde pek çok temel matematiksel ve istatistiksel fonksiyon bulunmaktadır. Bu fonksiyonların pek çoğu bizden argüman olarak bir yada birden fazla vektör alır, bunları işleme sokar ve işlem sonucunda bize yeni bir vektör verir.

- length fonksiyonu vektörün eleman sayısına geri döner. Örneğin:

```
> a <- c(1, 2, 3, 4, 5)
> b <- length(a)
> b
[1] 5
```

- sum fonksiyonu argüman olarak aldığı vektördeki değerlerin toplamına geri dönmektedir. Örneğin:

```
> a <- 1:10
> b <- sum(a)
> b
[1] 55
```

Aynı işlem şöyle de yapılabilirdi:

```
> sum(1:10)
[1] 55
```

- mean fonksiyonu aritmetik ortalama bulmak için kullanılır. Örneğin:

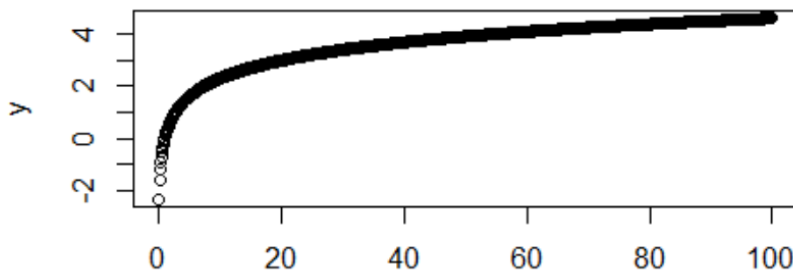
```
> a <- c(10, 20, 30, 20, 20)
> mean(a)
[1] 20
```

Aynı işlem şöyle yapılabilirdi:

```
> mean(c(10, 20, 30, 20, 20))
[1] 20
```

- log, log2, log10 fonksiyonları sırasıyla doğal logaritma, iki tabanına göre logaritma ve 10 tabanına göre logaritma hesabı yapar. Vektörün tüm elemanları bu logaritma işlemine sokulur ve yeni vektör elde edilir. Örneğin:

```
> a <- log10(0.1)
> x <- seq(0, 100, 0.1)
> y <- log(x)
> plot(x, y)
```



Örneğin:

```
> a <- log2(1024)
> a
[1] 10
```

- exp fonksiyonu üs almak için sqrt karekök almak için kullanılır. Örneğin:

```
> sqrt(10)
[1] 3.162278
```

- sort fonksiyonu vektörün elemanlarını sıraya dizerek bize sıraya dizilmiş yeni vektör verir. Örneğin:

```
> x <- sample(1:100, 10)
> x
[1] 43 36 58 13 41 49 39 63 100 64
> sort(x)
[1] 13 36 39 41 43 49 58 63 64 100
```

sort fonksiyonun ikinci parametresi isteğe bağlıdır ve default durumu F (False) biçimdedir. Bu parametre girilmezse (default durum) sıraya dizme küçükten büyüğe yapılır. Büyükten küçüğe sıraya dizme için bu ikinci parametrenin T (True) olarak girilmesi gerekir. Örneğin:

```
> sort(x)
[1] 13 36 39 41 43 49 58 63 64 100
> sort(x, F)
[1] 13 36 39 41 43 49 58 63 64 100
> sort(x, T)
[1] 100 64 63 58 49 43 41 39 36 13
```

- rev fonksiyonu vektörün ters dizilimini bize verir. Örneğin:

```
> a <- 1:10
> b <- rev(a)
>
> b
[1] 10 9 8 7 6 5 4 3 2 1
```

- print fonksiyonu bir vektörü ekrana yazdırmak için kullanılır. Örneğin:

```
> a <- 1:10
> print(a)
[1] 1 2 3 4 5 6 7 8 9 10
```

- cat fonksiyonu istenildiği kadar argüman alabilir. Fonksiyon bu argümanları birleştirip dosyaya ya da ekrana yazdırmaktadır. Örneğin:

```
> a <- 1:10
> cat(a, 100, 200)
1 2 3 4 5 6 7 8 9 10 100 200
```

- rep fonksiyonu belli bir vektörü çoklayarak yeni bir vektör verir. Fonksiyonun iki parametresi vardır. Birinci parametre çoklanacak vektörü ikinci parametre çoklama değerini belirtir. Örneğin:

```
> a <- c(10, 20)
> b <- rep(a, 5)
> b
[1] 10 20 10 20 10 20 10 20 10 20
```

Fonksiyonun ikinci parametresi de birinciyle aynı uzunlukta bir vektör olabilir. Bu durumda birinci vektörün elemanlarının hangi sayıda çoklanacağı belirtilir. Örneğin:

```
> a <- c(10, 20)
> b <- c(2, 4)
```

```
> c <- rep(a, b)
> c
[1] 10 10 20 20 20 20
```

- paste fonksiyonu istenildiği kadar çok argüman alabilir. Tüm bu argümanları yazıya dönüştürür ve uc uca ekler. Sonuç olarak bize bir yazı (string) verir. Örneğin:

```
> a <- 10
> b <- 20
> c <- paste(a, b)
> c
[1] "10 20"
```

paste fonksiyonun default olarak ayıraç olarak araya bir tane boşluk karakteri yerleştirdiğine dikkat ediniz. Eğer istenirse sep parametresiyle biz yıraç karakterini değiştirebiliriz. Örneğin:

```
> a <- "ali"
> b <- "veli"
> c <- "selami"
> d <- paste(a, b, c)
> d
[1] "ali veli selami"
> d <- paste(a, b, c, sep = "")
> d
[1] "aliveliselami"
```

- order fonksiyonu da tıpkı sort fonksiyonu gibi aslında bir sıralama yapmaktadır. Ancak sıralanan değerler vektördeki değerler değil onların indisleridir. Yani order fonksiyonundan biz bir indis vektör elde ederiz. Örneğin:

```
> a <- c(10, 30, 50, 25, 45)
> b <- order(a)
> b
[1] 1 4 2 5 3
```

- sequence fonksiyonu bizden bir vektör alır. Geri dönüş değeri olarak 1'den o vektördeki değerlere kadar olan değerlerin birleştirilmesiyle oluşan bir vektör verir. Örneğin:

```
> a <- c(1, 3, 5)
> b <- sequence(a)
> b
[1] 1 1 2 3 1 2 3 4 5
```

Örneğin:

```
> a <- c(10, 20)
> b <- sequence(a)
> b
[1] 1 2 3 4 5 6 7 8 9 10 1 2 3 4 5 6 7 8 9 10 11
[22] 12 13 14 15 16 17 18 19 20
```

- unique fonksiyonu bir vektördeki tekrar eden (mükerrer) değerleri atarak bize tekrar etmeyen değerlerden oluşan bir vektör verir. Örneğin:

```
> a <- c(2, 5, 4, 2, 7, 5, 9, 3)
> b <- unique(a)
> b
[1] 2 5 4 7 9 3
```

- union ve intersect iki vektörün kesişiminden ve birleşiminde oluşan vektörü bize verir. Örneğin:


```
> a <- c(3, 5, 8)
> b <- c(5, 8, 9)
> c <- union(a, b)
> c
[1] 3 5 8 9
```

Örneğin:

```
> a <- c(3, 5, 8)
> b <- c(5, 8, 9)
> c <- intersect(a, b)
> c
[1] 5 8
```

- sprintf fonksiyonu C'deki sprintf fonksiyonuna benzetilmiştir. Formatlı bir yazı elde etmek için kullanılır. Örneğin:

```
> a <- 10
> b <- 20.2
> str <- sprintf("a = %d, b = %f", a, b)
> str
[1] "a = 10, b = 20.200000"
```

- var fonksiyonu varyans hesaplamakta kullanılır. (Varyans standart sapmanın karesidir. Ve istatistikte değişkenliğin bir ölçüsü olarak kullanılmaktadır.) Örneğin:

```
> x <- c(4, 7, 4, 3, 4, 7, 8, 4, 3, 5)
> length(x)
[1] 10
> var(x)
[1] 3.211111
```

Anahtar Notlar: R Studio'da konsol ekranını silmek için Ctrl+L tuş kombinasyonu kullanılmaktadır. Yardım almak için ise ?<sözcük> sentaksı kullanılır.

varyans işlemini bir ifade olarak aşağıdaki gibi de yapabiliriz:

```
> sum((x - mean(x))^2) / (length(x) - 1)
[1] 3.211111
```

Burada var varyans için n değil n - 1 bölümünün uygulandığına dikkat ediniz.

- Standart sapma varyansın kareköküdür. sd fonksiyonu standart sapma için kullanılır. Örneğin:

```
> sqrt(var(x))
[1] 1.791957
> sd(x)
[1] 1.791957
```

- min fonksiyonu vektörün en küçük elemanını, max fonksiyonu en büyük elemanını bulmak için kullanılır. range fonksiyonu ise vektörün en küçük ve en büyük elemanını bize iki elemanlı bir vektör olarak verir. Örneğin:

```
> x <- c(1, 4, 6, 3, 2, 10, -4)
> min(x)
[1] -4
> max(x)
[1] 10
> range(x)
[1] -4 10
```

- median fonksiyonu istatisteki medyan değeri bulur. Anımsanacağı gibi vektör tek sayıda elemandan oluşuyorsa median ortadaki değerdir. Çift sayıda elemandan oluşuyorsa median ortadaki iki değer aritmetik ortalamasıdır. Örneğin:

```
> a <- c(3, 5, 2, 7, 9, 10)
> median(a)
[1] 6
> sort(a)
[1] 2 3 5 7 9 10
```

Örneğin:

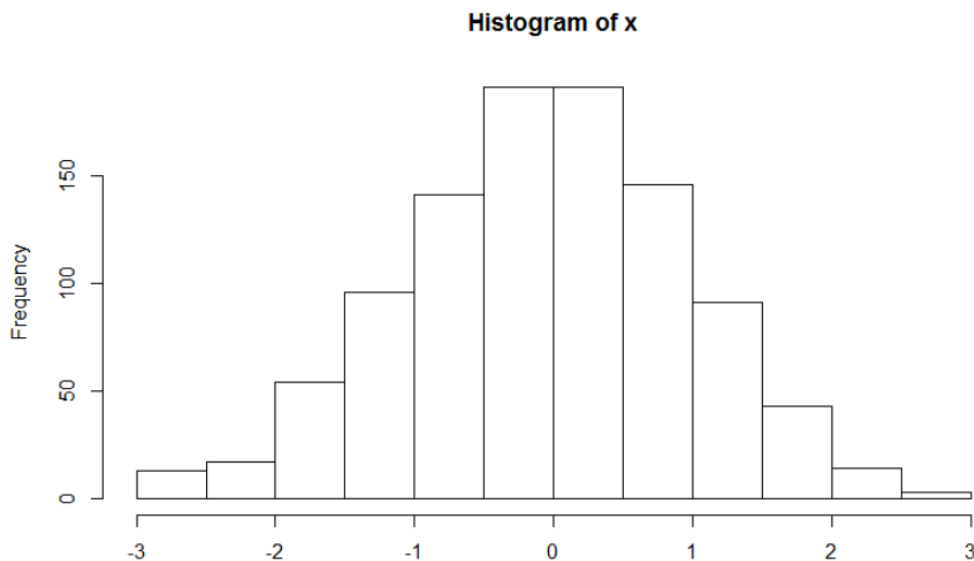
```
> a <- c(2, 5, 7, 8, 9)
> median(a)
[1] 7
> sort(a)
[1] 2 5 7 8 9
```

- which fonksiyonu argüman olarak bir bool vektör alır. Vektörün TRUE olan elemanlarına ilişkin vektör indislerini bize vektör olarak verir. Örneğin:

```
> a <- seq(10, 100, 10)
> a
[1] 10 20 30 40 50 60 70 80 90 100
> b <- c(T, F, T, T, F, F, T, T, T, F)
> a[b]
[1] 10 30 40 70 80 90
> which(b)
[1] 1 3 4 7 8 9
> which(a > 50)
[1] 6 7 8 9 10
```

- rnorm fonksiyonu ("random normal" sözcüklerinden geliyor. rastgele normal dağılmış (Gauss dağılmış) n tane değer üretmek için kullanılır. Default durumda bu değerler ortalaması 0, standart sapması 1 olan bir sistemden elde edilmektedir (standart normal dağılım). Örneğin:

```
> x <- rnorm(1000)
> hist(x)
```

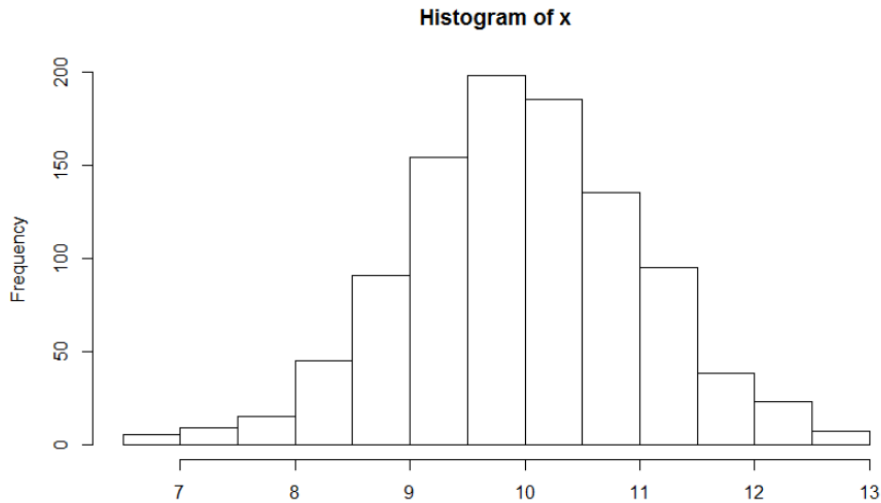


Fonksiyonun parametrik yapısı şöyledir:

```
rnorm(n, mean = 0, sd = 1)
```

Biz istersek ortalama ve standart sapması tam istediğimiz gibi olan bir sistemden rastgele normal dağılmış değerler de elde edebiliriz. Örneğin:

```
> x <- rnorm(1000, 10, 1)
> hist(x)
```



Bilindiği gibi istatistikte sürekli rassal değişkenlerin olasılık yoğunluk fonksiyonlarının (probability density functions) eğri altında kalan alanları 1'dir. İki nokta arasında eğri altında kalan alan ilgili rassal değişkenin o iki nokta arasına düşme olasılığını verir. Standart normal dağılımı (ortalaması 0, standart sapması 1 olan normal dağılım) kümülatif olasılıklara Z değeri denilmektedir. Biz pnorm fonksiyonu ile eksi sonsuzdan belli bir değere kadarki kümülatif olasılığı elde edebiliriz. Örneğin:

```
> pnorm(2)
[1] 0.9772499
```

Buradan eksi sonsuzdan 2'ye kadar standart normal dağılım eğrisi altında kalan alanın 0.9772999 olduğu görülmektedir. Bu aynı zamanda $P\{X < 2\}$ durumunu belirtmektedir. Başka bir deyişle pnorm(n) ile $P\{X < n\}$ aynı anlamdadır.

pnorm fonksiyonu da default olarak ortalaması 0, standart sapması 1 olan standart normal dağılımı dikkate almaktadır. Tabii biz iki argüman daha girerek bunu değiştirebiliriz:

```
> pnorm(10, 10, 1)
[1] 0.5
```

qnorm fonksiyonu pnorm fonksiyonun tersini yapmaktadır. Yani bu fonksiyon bizden eğri altında kalan alanı (kümülatif olasılığı) alır ona karşı gelen X (Z değerini) değerini verir. Örneğin:

```
> p <- pnorm(2)
> p
[1] 0.9772499
> x <- qnorm(p)
> x
[1] 2
```

qnorm fonksiyonunun da ortalamayı ve standart sapmayı girebileceğimiz default argümanları vardır. Örneğin:

```
> p <- pnorm(11, 10, 1)
> qnorm(p, 10, 1)
```

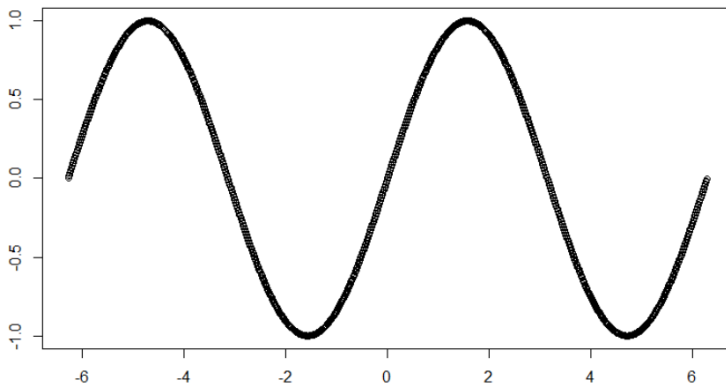
[1] 11

Ayrıca bir de `dnorm` isimli fonksiyon vardır. Bu fonksiyon bizden X değerini alır ve Gauss dağılım fonksiyonuna sokarak bize Y değerini verir. Örneğin:

```
> dnorm(1)
[1] 0.2419707
> dnorm(11, 10, 1)
[1] 0.2419707
```

- `sin`, `cos`, `tan`, `asin`, `acos`, `atan` fonksiyonları temel trigonometrik işlemleri yapmaktadır. Bu fonksiyonlarda açılar radyan cinsinden belirtilirler. Örneğin:

```
> x <- seq(-3.14*2, +3.14*2, 0.01)
> y <- sin(x)
> plot(x, y)
```



Vektör Elemanlarına Erişim

R'da bir vektörün elemanlarına erişmek için `[]` ve `[[]]` operatörü kullanılmaktadır. Biz önce `[]` operatörünü sonra da `[[]]` operatörünü ele alacağız.

Pek çok dilin aksine R'da vektör indeksleri 0'dan değil 1'den başlamaktadır. Örneğin:

```
> a <- c(10, 20, 30, 40, 50)
> a[1]
[1] 10
> a[5]
[1] 50
```

Tabii vektörün belli bir elemanına yeni bir değer de atayabiliriz:

```
> a <- c(10, 20, 30, 40, 50)
> a[1] <- 100
> a
[1] 100 20 30 40 50
```

Köşeli parantez içerisine genel olarak bir indeks vektörü yerleştirilebilir. Bu durumda o indekslerdeki elemanlardan oluşan bir vektör elde edilecektir. Örneğin:

```
> a <- 1:10
> b <- a[1:5]
> b
[1] 1 2 3 4 5
```

Ancak köşeli parantez içerisinde tek bir değer bulunmak zorundadır. O değer genel olarak bir vektör olabilir.

Örneğin:

```
> a[1, 3]
Error in a[1, 3] : incorrect number of dimensions
```

Index vektörü aynı elemanları içerebilir. Örneğin:

```
> a <- c(1, 2, 3, 4, 5)
> a[c(1, 3, 3, 5)]
[1] 1 3 3 5
```

Köşeli parantezler içerisindeki vektör (tek eleman da bir vektördür) negatif değerler içerebilir. Negatif bir değer "bu indeksin dışındaki tüm değerler" anlamına gelir. Örneğin:

```
> a <- c(10, 20, 30, 40, 50)
> a[-2]
[1] 10 30 40 50
```

Tabii indeks vektör birden fazla negatif değer de içerebilir. Örneğin:

```
> a <- c(10, 20, 30, 40, 50)
> a[c(-2, -2, -4)]
[1] 10 30 50
```

Ancak köşeli parantez içerisindeki indeks vektöründe pozitif ve negatif elemanlar bir arada bulunamazlar:

```
> a[c(-2, 4)]
Error in a[c(-2, 4)] : only 0's may be mixed with negative subscripts
```

Köşeli parantez içerisinde Bool türden (yani T (True) ve F (False) değerlerden oluşan) bir vektör yerleştirilebilir. Bu durumda T (True) olan elemanlara karşı gelen vektör değerleri elde edilir. Örneğin:

```
> a <- c(10, 20, 30, 40, 50)
> b <- c(T, F, F, T, T)
> a[b]
[1] 10 40 50
```

Bool türden vektör daha az elemana sahip olabilir. Bu durumda döngüye sokma işlemi (recycling) gerçekleştirilecektir. Örneğin:

```
> a <- c(10, 20, 30, 40, 50, 60)
> a[c(T, F)]
[1] 10 30 50
```

Bool vektör ile biz bir vektörde belli koşulu sağlayan elemanları kolayca elde edebiliriz. Örneğin:

```
> a <- c(10, 20, 30, 40, 50, 60)
> a[a > 35]
[1] 40 50 60
```

Burada önce a > 35 işleminden Bool bir vektör elde edilmektedir. Sonra o vektör indeks vektörü olarak kullanılarak T (True) olan vektör elemanları seçilmiştir. Örneğin bir vektördeki ortalamanın altında olan elemanlar şöyle elde edilebilir:

```
> a <- c(34, 56, 34, 23, 78, 45, 90)
```

```
> a[a < mean(a)]  
[1] 34 34 23 45
```

Köşeli parantez içerisindeki indeks vektöründe büyük indeksler olabilir. Bu durumda o indeks değerleri için NA (Not Available) özel değeri elde edilir Örneğin:

```
> a <- c(10, 20, 30, 40, 50, 60)  
> a[c(1, 3, 100, 5, 200)]  
[1] 10 30 NA 50 NA  
> a[c(-2, -3, -100)]  
[1] 10 40 50 60
```

. Ancak sıfır değeri ya da negatif büyük değerler varsa onlar tamamen görmemezlikten gelinmektedir. Başka bir deyişle negatif büyük indis değeri "o değerin dışındakilerin hepsi" anlamına geldiğine göre etkisi yoktur. Örneğin:

```
> a <- c(34, 56, 34, 23, 78, 45, 90)  
> a[-100]  
[1] 34 56 34 23 78 45 90
```

Köşeli parantez içerisine sıfır indeksi yerleştirilirse bu durumda R bize sıfır uzunlukta bir vektör verir. Örneğin:

```
> a <- c(10, 20, 30, 40, 50, 60)  
> b <- a[0]  
> length(b)  
[1] 0
```

Vektörün birden fazla elemanına tek hamlede atama yapabiliriz. Örneğin:

```
> a <- c(10, 20, 30, 40, 50, 60)  
> a[c(1, 3, 5)] <- 1000  
> a  
[1] 1000 20 1000 40 1000 60
```

Benzer biçimde atama işleminde indeks vektörü bool türden ise döngüye sokma işlemi yapılmaktadır. Örneğin:

```
> a <- c(10, 20, 30, 40, 50, 60)  
> a[c(T, F)] <- 100  
> a  
[1] 100 20 100 40 100 60
```

Bir vektörün uzunluğunun ötesindeki elemanlarına da atama yapılabilir. Bu durumda vektör büyütülmüş olur. Fakat aradaki elemanlar NA değerini alır. Başka bir deyişle olmayan elemanı elde etmek istediğimizde NA elde ederiz. Olmayan elemanlara atama yaptığımızda ise diziyi büyütmüş oluruz. Örneğin:

```
> a <- c(10, 20, 30, 40, 50, 60)  
> a[20] = 11111111  
> a  
[1] 10 20 30 40 50 60 NA  
[8] NA NA NA NA NA NA NA  
[15] NA NA NA NA NA 11111111  
> length(a)  
[1] 20
```

Örneğin:

```
> a <- c(10, 20, 30, 40, 50, 60)  
> a[10:15] <- 100  
> a
```

```
[1] 10 20 30 40 50 60 NA NA NA 100 100 100 100 100 100
```

Vektörün negatif büyük bir elemanına değer atamaya çalışırsak bundan tüm elemanlar etkilenir. Çünkü "negatif büyük indeksin dışındakilerin hepsi" zaten "vektörün elemanlarının hepsi" anlamına gelmektedir. Örneğin:

```
> a <- 1:10
> a[-20] <- 100
> a
[1] 100 100 100 100 100 100 100 100 100 100
```

Henüz hiç yaratılmamış bir vektörün elemanına bir değer yerleştiremeyiz. Örneğin x isimli bir değişkenin hiç yaratılmadığını düşünelim aşağıdaki atama işlemi error ile sonuçlanır:

```
x[1] <- 10
```

Bu işlemin yapılabilmesi için x'in daha önceden yaratılması gerekirdi. Bu işlemin aşağıdaki işlemten farklı olduğuna dikkat ediniz:

```
x <- 10
```

Burada x vektörü yaratılmış ve onun birinci elemanına 10 değeri atanmıştır.

R'da 0 uzunluklu bir vektör de yaratılabilmektedir. Bunun üç yolu vardır. Birinci yol integer, double, character gibi fonksiyonları kullanmaktır. Örneğin:

```
x <- integer()
```

İkinci yol vector isimli fonksiyondan faydalanmaktır. Örneğin:

```
v <- vector(mode = "integer")
```

vector fonksiyonuyla ilgili ayrıntılar ilerideki konularda ele alınacaktır.. Üçüncü yol ise mevcut bir vektörün sıfırıncı indeksli elemanını kullanmaktır. v isimli bir vektör değişkeni yaratılmışsa v[0] "v'nin türünden sıfır elemanlı bir vektör oluştur" anlamına gelir. R'da vektör indekslerinin 1'den başladığına dikkat ediniz.

Bir vektörün elemanlarına isim de verilebilir. Bu durumda ilgili elemana isim belirtilerek de erişilebilmektedir. İsim verme işlemi iki biçimde yapılabilir. Birincisinde c fonksiyonunda elemanları girerken "isim = değer" biçiminde girişi yapmaktır. Örneğin:

```
> a <- c(ali = 100, veli = 200, selami = 300)
> a
  ali   veli selami
100   200   300
```

Elemanlara isimler tırnak içerisinde string olarak da verilebilir. Örneğin:

```
> a <- c("ali" = 100, "veli" = 200, "selami" = 300)
> a
  ali   veli selami
100   200   300
```

Değişken isimlendirme kurallarına uymayan isimlerin mecburen bu biçimde verilmesi gerekmektedir. Örneğin:

```
> a <- c("ali" = 100, "veli" = 200, "selami ve ayşe" = 300)
> a
      ali          veli selami ve ayşe
100          200          300
```

Vektörün her elemanına isim vermek zorunda değiliz. Örneğin:

```
> a <- c(x = "Ali", y = "veli", "selami")
> a
      x      y
"Ali"  "veli" "selami"
```

Elemanlara isim vermenin diğer bir yolu da names isimli yer değiştirme fonksiyonunu aşağıdaki gibi kullanmaktır:

```
names(vektör_ismi) <- string_vektörü
```

Örneğin:

```
> a <- c(10, 20, 30, 40, 50, 60)
> names(a) <- c("Ali", "Veli", "Selami", "Ayşe", "Fatma", "Tuğrul")
> a
  Ali  Veli Selami  Ayşe  Fatma Tuğrul
  10   20   30    40   50   60
```

names fonksiyonu ile biz vektörün belli elemanlarına da isim verebiliriz. Örneğin:

```
> a <- c(10, 20, 30, 40, 50, 60)
> names(a)[2] <- "ali"
> a
<NA>  ali <NA> <NA> <NA> <NA>
  10   20   30   40   50   60
```

Vektörün elemanlarına verilmiş olan isimler onlara NULL değer atanarak silinebilir. Örneğin:

```
> a <- c(ali = 123, veli = 456)
> names(a) <- NULL
> a
[1] 123 456
```

Ancak vektörün belli bir elemanının ismini bu yolla silemeyiz. Örneğin:

```
> a <- c(ali = 123, veli = 456)
> names(a)[1] <- NULL
Error in names(a)[1] <- NULL : değiştirme sıfır uzunluğunda
```

Vektörün tüm elemanlarının isimleri unname isimli fonksiyonla da silinebilir. Örneğin:

```
> a <- c(ali = 123, veli = 456)
> unname(a)
[1] 123 456
```

Vektörün ilgili elemanına isim verilmişse artık biz o elemana köşeli parantez içerisinde o ismi string olarak geçirerek de referans edebiliriz. Örneğin:

```
> a["Ali"]
Ali
 10
```

Örneğin:

```
> a[c("Ali", "Selami", "Fatma")]
  Ali Selami  Fatma
  10    30    50
```


İsimlerde büyük harf küçük harf önemlidir (case sensitive). Dolayısıyla biz burada yanlış giriş yaparsak yine NA değeri elde ederiz. Örneğin:

```
> a[c("Ali", "selami", "Fatma")]
Ali  <NA> Fatma
10    NA    50
```

Eğer vektörde o isimli bir eleman yoksa NA değeri elde edilir. Örneğin:

```
> a[c("Ali", "Selma", "Fatma")]
Ali  <NA> Fatma
10    NA    50
```

Benzer biçimde vektörde olmayan isimli bir elemene atama yaparsak o elemanı vektörün sonuna eklemiş oluruz. Örneğin:

```
> a <- c(10, 20, 30)
> a["xxx"] <- 40
> a["yyy"] <- 50
> a
      xxx yyy
10  20  30  40  50
```

Örneğin:

```
> a <- c(10, 20, 30)
>
> a[c("ali", "veli", "selami")] <- c(100, 200, 300)
> a
      ali  veli selami
10    20    30   100   200   300
```

Vektör elemanlarına [[]] operatörü ile de erişilebilmektedir. Ancak [[]] operatörünün içerisindeki vektörün tek elemanlı olması zorunludur. Örneğin:

```
> a <- c(x = 10, y = 20, z = 30)
> a[[2]]
[1] 20
> a[["y"]]
[1] 20
> a[[c(1, 2)]]
Error in a[[c(1, 2)]] :
  attempt to select more than one element in vectorIndex
> a[[c("x", "y")]]
Error in a[[c("x", "y")]] :
  attempt to select more than one element in vectorIndex
```

Bir vektörün elemanlarına names fonksiyonuyla daha sonra da isim verilebilir ya da verilmiş olan isimler değiştirilebilir. Örneğin:

```
> a <- 1:4
> names(a) <- c("x", "y", "z", "k")
> a
x y z k
1 2 3 4
```

Burada fonksiyon çağrısının ok operatörünün solunda bulunduğuna dikkat ediniz. R'da böyle kullanılan fonksiyonlara "yer değiştirme fonksiyonları (replacement functions)" denilmektedir. name fonksiyonuna atanan string vektör daha

az elemana sahip olabilir. Bu durumda kalan elemanlara NULL isim verilmiş olur. Ancak fazla sayıda elemana isim vermeye çalışmak error ile sonuçlanmaktadır.

[] operatörü ile [[]] operatörü arasındaki tek fark [] operatörünün birden fazla elemanlı indis vektörü alabilmesi ancak [[]] operatörünün yalnızca tek elemanlı indis vektörü alabilmesidir. Örneğin:

```
> a <- c(ali = 100, veli = 200, selami = 300)
> a[c(1, 2)]
  ali veli
 100  200
> a[[c(1, 2)]]
Error in a[[c(1, 2)]] :
  attempt to select more than one element in vectorIndex
```

[[]] operatöründe indis negatif değer olabilir. Ancak seçilen değer yine tek olması gerekir. Örneğin:

```
> a <- c(x = 10, y = 20, z = 30)
> a[[-2]]
Error in a[[-2]] :
  attempt to select more than one element in get1index <real>
```

Ancak:

```
> a <- c(x = 10, y = 20)
> a[[-2]]
[1] 10
```

Yine [[]] operatörünün indeksi olarak birden fazla negatif değer içeren bir vektör veremeyiz. Örneğin:

```
> a <- c(x = 10, y = 20, z = 30)
> a[[c(-2, -1)]]
Error in a[[c(-2, -1)]] :
  attempt to select more than one element in vectorIndex
```

Benzer biçimde [[]] operatöründe -[] operatöründe olduğu gibi- olmayan bir elemana atama yapılırsa vektör büyütülmüş olur. Olmayan bir vektör elemanına erişim yine NA değerinin elde edilmesine yol açmaktadır.

R'da Fonksiyonlara Argüman Girme İşlemi

R'ın temel kütüphanesinde başkaları tarafından yazılmış pek çok fonksiyon vardır. Bu fonksiyonlar çeşitli parametreler almaktadır. Biz de fonksiyonları çağırırken bu parametreler için argümanlar gireriz. Argüman girmenin bazı kuralları vardır. R'da bir fonksiyonun parametrik yapısı ve bunların anlamları "yardım" alınarak görülebilir. Komut satırında ?<fonksiyon ismi> ifadesiyle o fonksiyon hakkında bilgi hızlı bir biçimde elde edilebilir. Örneğin sort fonksiyonunun parametrik yapısı şöyledir:

```
sort(x, decreasing = FALSE, ...)
```

R'da fonksiyonlar default argüman alabilmektedir. Default argüman ilgili parametreye bir değer girilmezse sanki o argüman girilmiş gibi etki yapar. Örneğin yukarıdaki sort fonksiyonunda decreasing parametresine default olarak FALSE değeri verilmiştir. Biz fonksiyonu çağırırken default değer alan parametreler için argüman girmek zorunda değiliz. Bu durumda sanki çağrı sırasında bu default değerler girilmiş gibi etki oluşacaktır. Örneğin yukarıdaki sort fonksiyonunun ikinci parametresi sıraya dizmenin küçükten büyüğe mi yoksa büyükten küçüğe mi yapılacağını belirtir. Eğer ikinci parametre FALSE girilirse sıraya dizme küçükten büyüğe, TRUE girilirse büyükten küçüğe yapılmaktadır. Bu durumda örneğin:

```
sort(x)
```

çağrısı ile:

```
sort(x, F)
```

çağırısı tamamen eşdeğer etkiye sahiptir. Biz büyükten küçüğe sıraya dizme için bu ikinci parametreyi açıkça TRUE girmemiz gerekir:

```
sort(x, T)
```

Default argüman çok kullanılan durumlar için fonksiyonun az argüman girilerek daha kolay çağrılmasına yol açmaktadır.

R'da argüman girerken parametre değişkeninin ismi de istenirse belirtilebilir. Bu işlem isim = değer biçiminde yapılır. Böylece biz eğer argümanı isimlendirirsek onları farklı sıralarda girebiliriz. Örneğin:

```
> a <- sort(decreasing = T, x = a)
```

Burada decreasing parametresi için T, x parametresi için ise a girilmiştir. Eğer argümanda isimler belirtilmezse girişin parametre sırasına göre yapıldığı anlaşılır. Ancak argümanda isim belirtilirse biz girişi istediğimiz sırada yapabiliriz.

R'da fonksiyonlar çok parametrelili olma eğilimindedir. Bu nedenle tüm parametreler için sırasıyla tek tek argüman girmek yerine yalnızca bazıları için onların isimlerini belirterek argüman girmek geri kalanları için ise default değerleri kullanmak çok uygulanan bir yöntemdir. Örneğin normal dağılmış rastgele sayı üretmek için kullanılan rnorm fonksiyonunun parametrik yapısı şöyle belirtilmiştir:

```
rnorm(n, mean = 0, sd = 1)
```

Bu parametrik yapıya göre rnorm fonksiyonunu biz en az bir parametre çağırmak zorundayız. Örneğin:

```
> a <- rnorm(20)
> a
[1] -1.50279473 -0.05871795 -1.00874670  0.43146554  0.46016773  2.23006472
[7]  0.06181493  0.02350081 -0.79207504  1.86465588  0.65671625 -1.40508083
[13] -0.25499096 -0.60345576 -1.26534176  0.78253359 -0.85565088 -1.26187570
[19] -0.94883431  0.51986000
```

rnorm fonksiyonunu tek parametreyle çağırdığımızda aslında biz mean parametresi için 0, sd parametresi için 1 girmiş gibi oluruz. Yani yukarıdaki çağrıyla biz ortalaması 0, standart sapması 1 olan 20 tane normal dağılmış rastgele sayı elde ederiz. Örneğin:

```
> a <- rnorm(20, 10)
> a
[1] 11.348835 10.063667  8.407719  9.760907 10.809608  9.782837 10.083690
[8]  9.150398  9.431458 11.650230  9.378288 10.947187  9.322524  9.156081
[15]  9.426711 10.640213  8.802042 10.202101  9.087402 11.807196
```

Burada 10 değeri mean parametresi için girilmiştir. Ancak sd yine 1 alınacaktır. Örneğin:

```
> a <- rnorm(20, 10, 2)
> a
[1]  8.915381 11.850115 11.110305 13.714220  8.347249  7.517320  9.424634
[8] 11.560255  9.476678  7.752482  8.347498 11.690092  5.954294  8.779689
[15] 11.814129  9.414470 13.332150 11.348432  9.072280  7.853918
```

Burada artık standart sapma değeri 2 olarak girilmiştir. Yukarıdaki çağrılarda parametre isimlerinin belirtilmediğine dikkat ediniz. Bu durumda argümanlar sırasıyla parametrelerle eşleşmektedir. Ancak biz argümanlarda parametre isimlerini belirterek onları farklı sıralarda da girebiliriz. Örneğin:

```
> a <- rnorm(sd = 2, mean = 10, n = 20)
```

```
> a
[1] 10.686163  8.914080  9.633009 12.001096  6.794499  9.258110 12.156021
[8] 12.350864  9.008265  7.961633 13.951242 10.728708 11.143667 13.370082
[15]  9.116364 11.758134  8.802849 10.870438  8.677136 11.507828
```

Burada argümanlar farklı sıralarda girilmiştir. Pekiyi bazı arümanlar için parametre isimleri verip bazıları için vermeyebilir miyiz? Örneğin:

```
a <- rnorm(sd = 2, 10, me = 20)
```

Bu geçerli midir? Geçerliyse hangi parametreler için hangi değerler girilmiştir?

İşte R'da argüman parametre eşleştirmesi için şu kurallar sırasıyla işletilmektedir:

- 1) Önce çağırma ifadesindeki isimli argümanların isimleri ile tam eşleşen parametreler varsa bu argümanların o parametreler için girildiği kabul edilir ve bu parametreler listeden çıkartılır.
- 2) Sonra çağırma ifadesindeki isimli argümanların isimleri ile kısmi eşleşen parametreler belirlenir. Bu kısmi eşleşen argümanların da o parametreler için girildiği kabul edilir. Onlar da listeden çıkartılır. (Argümanda parametre isminin tamamı değil onun yalnızca ilk n karakteri belirtilebilir. Buna kısmi eşleşme (partial matching) denilmektedir. Örneğin "mean" isminde parametre için "m", "me" ve "mea" kısmi eşleştirme oluşturur ancak "mb" ya da "mcd" kısmi eşleştirme oluşturmaz. Tabii kısmi eşleşen birden fazla parametrenin error'e yol açacaktır.) Kısmi eşleşme yalnızca "..." parametresinden önceki parametrelere uygulanmaktadır. Sonraki parametrelere uygulanmamaktadır.
- 3) Geri kalanlar sırasıyla isim verilmemiş argümanlarla eşleştirilir. Eğer ilgili argüman "..." parametresine karşılık geliyorsa diğer argümanların hepsi bu "..." parametresi ile eşleştirilir.
- 4) Eşleşmeyen parametreler varsa bunların default değeri alması gerekir. Aksi takdirde error oluşturur.

Şimdi rnorm fonksiyonunun parametrik yapısına bir kez daha bakalım:

```
rnorm(n, mean = 0, sd = 1)
```

Şimdi de aşağıdaki çağırışı inceleyelim:

```
a <- rnorm(sd = 2, 10, me = 20)
```

Burada birinci aşamada tam uyum sağlayan sd argümanı vardır. Bu parametre listesinden çıkartılır. Sonra kısmi eşleşen sağlayan me argümanı vardır. Bu da listeden çıkartılır. Listede yalnızca n kalmıştır. İsimsiz olan 10 parametresi bununla eşleşir. Sonuç olarak bu çağrı ile biz 10 tane standart sapması 2 olan, ortalaması 20 olan normal dağılmış rastgele sayı elde etmiş oluruz.

```
> a <- rnorm(sd = 2, 10, me = 20)
> a
[1] 16.84862 24.46545 22.58462 19.21947 19.51264 18.46788 19.87926 20.00262
[9] 20.64131 18.08158
```

Örneğin:

```
> a <- rnorm(s = 2, 10, m = 20)
> a
[1] 22.48644 17.96020 19.53078 22.33452 17.10982 19.31221 20.64080 19.56527
[9] 18.73219 18.25137
```

Şimdi aşağıdaki gibi bir foo fonksiyonun blunduğunu varsayalım:

```
foo(name, count, date, min = 100)
```

Bu fonksiyonu şöyle çağırmış olalım:

```
foo(name = "Ali Serçe", dat = "11/10/2009", 200)
```

Burada name parametresi tam eşleşmeyle ve dat parametresi de kısmi eşleşmeyle listeden çıkartılır. Artık listede sırasıyla count ve min parametreleri kalmıştır. Çağırma ifadesinde tek kalan değer 100'dür. Bu yüz sıradaki listede kalan ilk parametreyle yani count ile eşleşir. min parametresi default değer aldığı için argüman belirlemesi yapılmayabilmektedir. Dolayısıyla sanki min parametresi 100 değeriyle çağrılmış gibi işlem görecektir. Örneğin çağrı aşağıdaki gibi yapılmış olsun:

```
foo(dat = "11/10/2009", min = 200, "Ali Serçe", 300)
```

Burada önce min argümanı tam eşleşmeyle sonra da dat argümanı kısmi eşleşmeyle listeden çıkartılır. Elde name ve count kalmıştır. Bunlar da sırasıyla "Ali Serçe" ve 300 argümanlarıyla eşleşir.

Default değer almayan parametreler için çağrıda mutlaka argüman girilmesi gerekmektedir. Aksi takdirde error oluşur. foo fonksiyonunun aşağıdaki gibi yazılmış olduğunu varsayalım:

```
foo(name, count = 10, date, min)
```

Bizim de fonksiyonu şöyle çağırmış olalım:

```
foo(date = "11/10/2007", nam = "Ali Serçe", 20)
```

Bu durumda 20 değeri count için verilmiş kabul edilecektir. min parametresi için bir argüman girilmediğinden error oluşacaktır.

Ayrıca R'da değişken sayıda argüman alan fonksiyonlar da yazılabilmektedir. Bunun için "..." (ellipsis) karakterleri kullanılır. "..." (ellipsis) parametresi için istenildiği kadar argüman girebilir. Örneğin fonksiyon şöyle yazılmış olsun:

```
foo(a, b, ..., c)
```

Fonksiyonunu şöyle çağırmış olalım:

```
foo(b = 20, c = 30, 40, 50, 60, 70)
```

b ve c tam eşleşmeyle listeden çıkartılır. Geri kalan isimsiz argümanlar sırasıyla 40, 50, 60 ve 70'tir. 40 a ile eşleştirilir. 50 ise "..." ile eşleşir. Artık kurala göre geri kalanların hepsi (yani 60 ve 70) "..." ile eşleşecektir. Çağrı aşağıdaki gibi yapılsaydı error oluşurdu:

```
foo(b = 20, 30, 40, 50, 60, 70)
```

Burada b tam eşleşmeyle listeden çıkartılır. 30 a ile eşleşir. Geri kalanların hepsi "..." ile eşleşecektir. Bu durumda c için bir argüman yazılmamış olur. Tabii fonksiyonda c için default değer verilmiş olsaydı bu durum error oluşturmazdı. Örneğin:

```
foo(a, b, ..., c = 10)
```

Fonksiyon şöyle çağrılmış olsun:

```
foo(b = 20, 30, 40, 50, 60, 70)
```

Artık 20 b ile, 30 a ile, 40, 50, 60, 70 ise "..." ile eşleşir. c için default değer verilmiş olduğundan dolayı error oluşmaz.

Düzgün Dağılmış Rastgele Sayıların Üretilmesi

Rassallıkta düzgün dağılımı her ögenin çıkma olasılığının aynı olması anlamına gelmektedir. R'da düzgün dağılmış rastgele sayı üretmek için iki fonksiyon çok kullanılmaktadır: runif ve sample. runif fonksiyonu şöyle bildirilmiştir:

```
runif(n, min = 0, max = 1)
```

Fonksiyonun birinci parametresi kaç tane rassal sayının üretileceğini ikinci ve üçüncü parametreleri de üretilecek aralığı belirtir. runif fonksiyonu bu aralıkta gerçek sayı (noktalı sayı) üretmektedir. Örneğin:

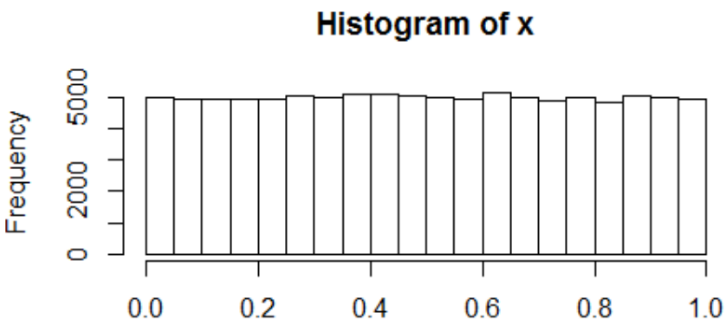
```
> x <- runif(10)
> x
[1] 0.0385764 0.4435728 0.4584551 0.2035357 0.8902581 0.8629367
[7] 0.8458173 0.4725838 0.8467611 0.9651068
```

Örneğin:

```
> x <- runif(10, 10, 20)
> x
[1] 19.60879 17.75852 12.94346 12.31143 18.45564 14.52903 18.56585
[8] 15.22747 11.17569 15.19480
```

Dağılımın düzgünlüğünü görsel olarak şöyle test edebiliriz:

```
> x <- runif(100000)
> hist(x)
```



sample fonksiyonu belli bir vektörden rastgele elemanları çekmektedir. Bildirimi şöyle yapılmıştır:

```
sample(x, size, replace = FALSE, prob = NULL)
```

Fonksiyonun birinci parametresi rastgele çekimin yapılacağı vektörü ikinci parametresi çekim sayısını belirtir. replace parametresi çekimin iadeli mi iadesiz mi yapılacağını belirtmektedir. Bu parametre FALSE olarak girilirse çekim iadesiz, TRUE olarak girilirse iadeli yapılmaktadır. Örneğin:

```
> sample(1:10, 4)
[1] 7 9 4 10
```

Örneğin:

```
> sample(c("Ali", "Veli", "Selami", "Ayşe", "Fatma"), 2)
[1] "Veli" "Fatma"
```

Örneğin:

```
> sample(c("Ali", "Veli", "Selami", "Ayşe", "Fatma"), 5, r = T)
[1] "Selami" "Veli" "Veli" "Selami" "Ali"
```

Örneğin:

```
> sample(c("Ali", "Veli", "Selami", "Ayşe", "Fatma"), 5)
[1] "Selami" "Veli"   "Ayşe"   "Ali"    "Fatma"
```

Sınıf Çalışması: Pi/4 değerini veren aşağıdaki seriden hareketle Pi sayısını 1000000 yinlemeyele hesaplayınız:

$$\frac{\pi}{4} = \sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{2k-1}$$
$$= 1 - \frac{1}{3} + \frac{1}{5} - \dots$$

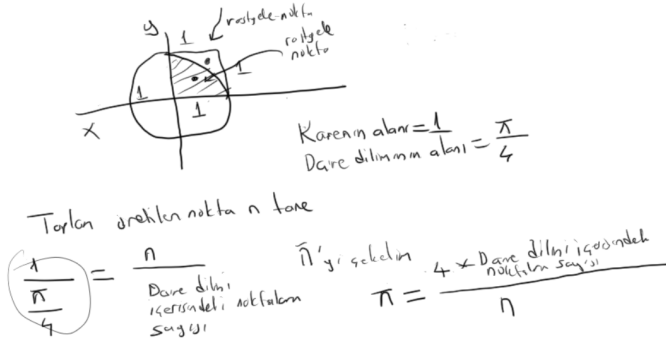
Çözüm:

```
> a <- seq(1, 1000000, 4)
> positives <- 1/a
> b <- seq(3, 1000000, 4)
> negatives <- -1/b
> sum(positives + negatives) * 4
[1] 3.141591
```

Bu işlemler tek hamlede şöyle de yapılabilir:

```
sum(1/seq(1, 1000000, 4), -1/seq(3, 1000000, 4)) * 4
```

Sınıf Çalışması: Düzgün dağılmış rastgele sayı üreterek pi sayısını burada belirtildiği gibi hesaplayınız. Önce bir birim çember alalım. Sonra bunun 4'te birlik kısmına dikkat edelim. Sonra da 0 ile 1 arasında x ve y bileşenlerinden oluşan n tane nokta alalım. Bu n tane nokta 1x1'lik karenin içerisindedir. O halde bu karenin alanının daire dilimin alanına oranı üretilen tüm sayıların daire dilimindeki noktaların sayısının oranına eşittir. Buradan da Pi değerini çekebiliriz:



Çözüm:

```
> x <- runif(n, 0, 1)
> y <- runif(n, 0, 1)
> len <- sqrt(x ^ 2 + y ^ 2)
> insidepie <- sum(len < 1)
> pi <- 4 * insidepie / n
> pi
[1] 3.140272
```

Bu işlemi çok kompakt biçimde şöyle yazabiliriz:

```
> pi <- 4 * sum(sqrt(runif(n)^2 + runif(n)^2) < 1) / n
> pi
[1] 3.142344
```

R'da Matrisler ve Çok Boyutlu Diziler

Matrisler veri analizinde çokça kullanılan veri yapılarıdır. Doğada pek çok olgu matrisel biçimde modellenebilmektedir. Örneğin sayısal bir tablo, satranç tahtası, bulmaca hemen matrisi çağrıştırmaktadır. Aslında matris çok boyutlu bir dizinin özel bir durumudur. Ancak üç boyutlu dizilerle modellemede seyrek karşılaşılmaktadır. Yüksek dizilerle ise hepten seyrek karşılaşılr.

R'da aslında matrisler (genel olarak çok boyutlu diziler) birer vektör gibi ele alınmaktadır. Bir matrisi vektörden ayıran tek şey onun kaç satır kaç sütundan oluştuğu bilgisidir. Yoksa onun verileri değildir. Örneğin aşağıdaki gibi bir vektör söz konusu olsun:

```
1 2 3 4 5 6 7 8 9
```

Biz bu değerleri aşağıdaki gibi görüntülersek matrisel bir temsil kullanmış oluruz:

```
1 2 3
4 5 6
7 8 9
```

Başka bir deyişle R'da ileride de görüleceği üzere vektör ile matris arasında yalnızca özellik (attribute) bakımından farklılık vardır.

R'da bir matris vektör kullanılarak yaratılır. Matris yaratmada kullanılan tipik fonksiyon `matrix` isimli fonksiyondur:

```
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
```

Fonksiyonun birinci parametresi matrisi oluşturacak vektörü, ikinci parametresi matrisin satır sayısını, üçüncü parametresi ise sütun sayısını belirtmektedir. Örneğin:

```
> m <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10), nrow = 5, ncol = 2)
> m
      [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10
```

Matris elemanlarına dikkat ediniz. R'da matrisler her zaman default olarak sütunsal biçimde oluşturulmaktadır. Matrisi satırsal biçimde oluşturmak için `byrow` parametresi `TRUE` girilmelidir. Örneğin:

```
> m <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10), nrow = 5, ncol = 2, byrow = T)
> m
      [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]    7    8
[5,]    9   10
```

Aslında matrisi yaratırken `nrow` ya da `ncol` parametresinden yalnızca birini verebiliriz. Diğer boyut bu değerden hareketle belirlenebilmektedir. Örneğin:

```
> m <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10), nrow = 5)
> m
      [,1] [,2]
[1,]    1    6
```



```
[2,] 2 7
[3,] 3 8
[4,] 4 9
[5,] 5 10
```

Ya da örneğin:

```
> m <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10), ncol = 2)
> m
      [,1] [,2]
[1,] 1    6
[2,] 2    7
[3,] 3    8
[4,] 4    9
[5,] 5   10
```

Eğer matris yaratılırken nrow ve ncol parametresi girilir ancak birinci vektör parametresi bu değerlerin çarpımından küçük olursa döngüye sokma işlemi uygulanır. Ancak bir uyarı mesajı da verilmektedir. Örneğin:

```
> m <- matrix(1:9, nrow = 5, ncol = 3)
Warning message:
In matrix(1:9, nrow = 5, ncol = 3) :
  data length [9] is not a sub-multiple or multiple of the number of rows [5]
> m
      [,1] [,2] [,3]
[1,] 1    6    2
[2,] 2    7    3
[3,] 3    8    4
[4,] 4    9    5
[5,] 5    1    6
```

Eğer nrow ya da ncol değeri tek başına belirtilmişse ve diğer boyut vektör uzunluğuna göre uygun değilse matris döngüye sokma işlemiyle en büyük kata dönüştürülür ve bir uyarı mesajı da verilir. Örneğin:

```
> m <- matrix(1:10, nrow = 3)
Warning message:
In matrix(1:10, nrow = 3) :
  data length [10] is not a sub-multiple or multiple of the number of rows [3]
> m
      [,1] [,2] [,3] [,4]
[1,] 1    4    7   10
[2,] 2    5    8    1
[3,] 3    6    9    2
```

Yine eğer birinci parametredeki vektör nrow ve ncol çarpımından büyükse bu vektörün nrow * ncol kadar elemanı alınarak matris oluşturulur. Yine bu durumda da bir uyarı verilir:

```
> m <- matrix(1:100, nrow = 3, ncol = 2)
Warning message:
In matrix(1:100, nrow = 3, ncol = 2) :
  data length [100] is not a sub-multiple or multiple of the number of rows [3]
> m
      [,1] [,2]
[1,] 1    4
[2,] 2    5
[3,] 3    6
```

Tabii iyi bir teknik için programcının uyarı gerektirecek bir duruma yol açmaması tavsiye edilmektedir.

Matris elemanlarına köşeli parantez içerisinde iki indis ile erişilebilir. Örneğin:

```

> m <- matrix(1:9, nrow = 3)
> m
      [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> m[1, 2]
[1] 4
> m[2, 3]
[1] 8

```

Elemana erişirken indislerin matrisin sınırları içerisinde olması gerekir. Aksi takdirde error oluşur. Örneğin:

```

> m[3, 5]
Error in m[3, 5] : altindis sınırlar dışında

```

Matrisi indekslerken indislerde birden fazla eleman içeren vektör kullanılabilir. Bu durumda o indisleri içeren alt matrisler elde edilir. Örneğin:

```

> m <- matrix(1:9, nrow = 3)
> m
      [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> m[c(1, 2), c(2, 3)]
      [,1] [,2]
[1,]    4    7
[2,]    5    8

```

Burada sonucun bir matris olarak elde edildiğine dikkat ediniz. Eğer erişimde elde edilen alt matrisin satır sayısı ya da sütun sayısı 1 ise bu durumda sonuç bir vektör olarak elde edilir. Örneğin:

```

> m[1:3, 1]
[1] 1 2 3
> m[2, 2:3]
[1] 5 8

```

Matris indislerinde satır ya da sütunlardaki negatif değerler yine "bunun dışındaki tüm indisler anlamına gelmektedir. Örneğin:

```

> m <- matrix(1:9, nrow = 3)
> m
      [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> m[1, -2]
[1] 1 7
> m[-2, -1]
      [,1] [,2]
[1,]    4    7
[2,]    6    9
> m[c(-1, -2), c(1, 2)]
[1] 3 6

```

Tabii yine satır ve sütun indesklerinde negatif ve pozitif değerler aynı anda bulundurulamazlar:

```

> m[c(-1, 2), c(1, 2)]

```

Error in m[c(-1, 2), c(1, 2)] :
negatif altindislerle sadece 0'lar karıştırılabilir

Matrislerde elemana erişimde satır ya da sütun değeri yazılmazsa tüm satır ya da tüm sütun anlaşılmaktadır.

Örneğin:

```
> m <- matrix(1:9, nrow = 3)
> m
      [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> m[,2]
[1] 4 5 6
> m[2,]
[1] 2 5 8
> m[, ]
      [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

Matrisin elemanlarını da aynı biçimde değiştirebiliriz. Örneğin:

```
> m <- matrix(1:9, nrow = 3)
> m
      [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> m[1, 1] <- 100
> m
      [,1] [,2] [,3]
[1,]  100    4    7
[2,]    2    5    8
[3,]    3    6    9
> m[2, ] <- 100
> m
      [,1] [,2] [,3]
[1,]  100    4    7
[2,]  100  100  100
[3,]    3    6    9
> m[2:3, 2:3] <- 0
> m
      [,1] [,2] [,3]
[1,]  100    4    7
[2,]  100    0    0
[3,]    3    0    0
```

Eğer atanacak vektör hedef vektörden küçükse sütun temelinde döngüye sokma işlemi uygulanmaktadır. Örneğin:

```
> m <- matrix(1:9, nrow = 3)
> m
      [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> m[1:3, 2:3] <- c(10, 20)
> m
      [,1] [,2] [,3]
[1,]    1   10   20
[2,]    2   20   10
```

```
[3,] 3 10 20
```

Tabii biz bir matrisin belli bir kısmına başka bir matrisi de atayabiliriz:

```
> m <- matrix(1:9, nrow = 3)
> m
      [,1] [,2] [,3]
[1,] 1    4    7
[2,] 2    5    8
[3,] 3    6    9
> m[, 2:3] <- matrix(c(10, 20, 30, 40, 50, 60), ncol = 2)
> m
      [,1] [,2] [,3]
[1,] 1    10   40
[2,] 2    20   50
[3,] 3    30   60
```

Matris elemanlarına erişirken köşeli parantez içerisinde tek bir vektör girersek bu durumda matris sütunsal bir vektör gibi ele alınır, sonuç da bir vektör olur. Örneğin:

```
> m <- matrix(1:9, nrow = 3)
> m
      [,1] [,2] [,3]
[1,] 1    4    7
[2,] 2    5    8
[3,] 3    6    9
> m[c(1, 2, 3)]
[1] 1 2 3
```

Örneğin

```
> m <- matrix(1:9, nrow = 3)
> m
      [,1] [,2] [,3]
[1,] 1    4    7
[2,] 2    5    8
[3,] 3    6    9
> k <- m > 5
> k
      [,1] [,2] [,3]
[1,] FALSE FALSE TRUE
[2,] FALSE FALSE TRUE
[3,] FALSE  TRUE TRUE
> m[k]
[1] 6 7 8 9
```

matrix fonksiyonunda yalnızca satır ve sütun uzunluğunu belirterek de boş bir matris oluşturabiliriz. Örneğin:

```
> m <- matrix(nrow = 2, ncol = 2)
> m
      [,1] [,2]
[1,] NA   NA
[2,] NA   NA
> m[,] <- 1:4
> m
      [,1] [,2]
[1,] 1    3
[2,] 2    4
```

Tabii matrix fonksiyonun parametrik yapısına baktığımızda onun aslında hiç argüman girilmeden de çağrılabilceğini görmekteyiz. nrow ve ncol parametrelerinin default 1 değerine sahip olduğuna dikkat ediniz:

```
> m <- matrix()
> m
      [,1]
[1,]    NA
```

Matrislerde de vektörlere benzer biçimde indis için bool vektör verilebilir. Örneğin:

```
> m <- matrix(1:9, nrow = 3)
> m
      [,1] [,2] [,3]
[1,]     1     4     7
[2,]     2     5     8
[3,]     3     6     9
> m[c(T, T, F), c(T, F, T)]
      [,1] [,2]
[1,]     1     7
[2,]     2     8
```

Matrislerde elemanlara tek bir indisle de erişebiliriz. Bu durumda matris sanki bir vektör gibi ele alınmaktadır. Dolayısıyla elde edilen ürün de vektör olacaktır. Örneğin:

```
> m <- matrix(1:9, nrow = 3)
> m[1]
[1] 1
> m[1:3]
[1] 1 2 3
```

Matrislere de karşılaştırma operatörleriyle filtrelemeler yapılabilir. Örneğin:

```
> m <- matrix(1:9, nrow = 3)
> m > 5
      [,1] [,2] [,3]
[1,] FALSE FALSE TRUE
[2,] FALSE FALSE TRUE
[3,] FALSE  TRUE TRUE
> m[m > 5]
[1] 6 7 8 9
```

Örneğin:

```
> m <- matrix(1:9, nrow = 3)
> m
      [,1] [,2] [,3]
[1,]     1     4     7
[2,]     2     5     8
[3,]     3     6     9
> a <- matrix(c(F, F, F, F, T, T, F, T, T), nrow = 3)
> m[a]
[1] 5 6 8 9
```

Tabii biz köşeli parantez içerisinde birden fazla indis girersek sonucu bir matris olarak elde edebiliriz. Örneğin:

```
> m <- matrix(1:9, nrow = 3)
> m[c(T, F, T), ]
      [,1] [,2] [,3]
[1,]     1     4     7
[2,]     3     6     9
```

Matrisin elemanlarına tıpkı vektörlerde olduğu gibi [[]] operatörüyle de erişilebilir. Ancak bu operatörde satır ve sütun indislerinde yalnızca tek elemanlı vektörler kullanılabilir. Başka bir deyişle [[]] operatörü yine matrislerde de tek elemana erişmek için kullanılmaktadır. Örneğin:

```
> m
      [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> m[[1, 2]]
[1] 4
[1] 1
> m[[1:2, 3]]
Error in m[[1:2, 3]] :
  attempt to select more than one element in get1index
> m[[1,]]
Error in m[[1, ]] : subscript out of bounds
> m[[1]]
[1] 1
```

Matrislerin satır ve sütunlarına isimler verebiliriz. Bunun için rownames ve colnames fonksiyonları kullanılmaktadır. rownames ve colnames daha önce vektörlerde gördüğümüz names fonksiyonu gibi "yer değiştirme (replacement)" fonksiyonlarıdır. Dolayısıyla atama operatörünün hedefi olarak kullanılırlar. Örneğin:

```
> m <- matrix(1:9, ncol = 3)
> m
      [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> rownames(m) <- c("x", "y", "z")
> colnames(m) <- c("a", "b", "c")
> m
   a b c
x 1 4 7
y 2 5 8
z 3 6 9
```

Tabii yer değiştirme fonksiyonlarının ok operatörünün hedefinde olması zorunlu değildir. Örneğin:

```
> rownames(m)
[1] "x" "y" "z"
> colnames(m)
[1] "a" "b" "c"
```

Bir matris matrix fonksiyonunun dışında cbind ve rbind fonksiyonlarıyla da oluşturulabilir. Fonksiyonların parametrik yapılarını inceleyiniz:

```
cbind(..., deparse.level = 1)
rbind(..., deparse.level = 1)
```

cbind vektörlerden sütunsal olarak rbind ise satıral olarak matris yapar. Örneğin:

```
> m <- cbind(c(1, 2, 3), c(4, 5, 6), c(7, 8, 9))
> m
      [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

Örneğin:

```
> m <- rbind(c(1, 2, 3), c(4, 5, 6), c(7, 8, 9))
> m
      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

Tabii cbind ve rbind fonksiyonlarında girilen vektörlerin aynı uzunlukta olması gerekmektedir. Eğer bu vektörler aynı uzunlukta değilse en uzun olanına göre matris oluşturulur. Kısa olanlar döngüye sokma işlemiyle uzuna tamamlanmaktadır. R bu durumda bir uyarı da vermektedir. Örneğin:

```
> m <- rbind(c(1, 2, 3, 10), c(4, 5, 6), c(7, 8, 9))
Warning message:
In rbind(c(1, 2, 3, 10), c(4, 5, 6), c(7, 8, 9)) :
  number of columns of result is not a multiple of vector length (arg 2)
> m
      [,1] [,2] [,3] [,4]
[1,]    1    2    3   10
[2,]    4    5    6    4
[3,]    7    8    9    7
```

Şüphesiz programcının bir uyarı mesajı gerektiren durumları oluşturmaması iyi bir tekniktir.

cbind ve rbind fonksiyonlarında matrisin satır ve sütunlarına isim de verebiliriz. Örneğin:

Örneğin:

```
> m <- cbind(a = c(1, 2, 3), b = c(4, 5, 6), c = c(7, 8, 9))
> m
      a b c
[1,] 1 4 7
[2,] 2 5 8
[3,] 3 6 9
```

Örneğin:

```
> m <- rbind(x = c(1, 2, 3), y = c(4, 5, 6), z = c(7, 8, 9))
> m
      [,1] [,2] [,3]
x       1    2    3
y       4    5    6
z       7    8    9
```

cbind ve rbind fonksiyonlarına vektör değişkenleri verilirse ilgili satır ya da sütunun isimleri bu değişkenlerden alınır. Örneğin:

```
> a <- c(1, 2, 3)
> b <- c(4, 5, 6)
> c <- c(7, 8, 9)
> m <- cbind(a, b, c)
> m
      a b c
[1,] 1 4 7
[2,] 2 5 8
[3,] 3 6 9
```

Örneğin:

```

> x <- c(1, 2, 3)
> y <- c(4, 5, 6)
> z <- c(7, 8, 9)
> m <- rbind(x, y, z)
> m
  [,1] [,2] [,3]
x    1    2    3
y    4    5    6
z    7    8    9

```

Mevcut bir matrizen cbind ve rbind fonksiyonlarıyla daha geniş matrisler elde edebiliriz. Başka bir deyişle mevcut matrisin satır sütun eklenmiş halini elde edebiliriz. Örneğin:

```

> m <- matrix(1:9, ncol = 3)
> m
  [,1] [,2] [,3]
[1,]   1   4   7
[2,]   2   5   8
[3,]   3   6   9
> d <- cbind(m, c(10, 11, 12))
> d
  [,1] [,2] [,3] [,4]
[1,]   1   4   7  10
[2,]   2   5   8  11
[3,]   3   6   9  12
> k <- cbind(c(13, 14, 15), d, c(16, 17, 18))
> k
  [,1] [,2] [,3] [,4] [,5] [,6]
[1,]  13   1   4   7  10  16
[2,]  14   2   5   8  11  17
[3,]  15   3   6   9  12  18

```

Matrislerin satır ve sütun sayıları sırasıyla nrow ve ncol fonksiyonlarıyla elde edilebilir. Örneğin:

```

> m <- matrix(1:9, ncol = 3)
> m
  [,1] [,2] [,3]
[1,]   1   4   7
[2,]   2   5   8
[3,]   3   6   9
> ncol(m)
[1] 3
> nrow(m)
[1] 3

```

Matrisler birer vektör olduğuna göre skaler çarpım normal biçimde yapılabilir. Örneğin:

```

> a <- matrix(1:9, ncol = 3)
> a
  [,1] [,2] [,3]
[1,]   1   4   7
[2,]   2   5   8
[3,]   3   6   9
> 3 * a
  [,1] [,2] [,3]
[1,]   3  12  21
[2,]   6  15  24
[3,]   9  18  27

```

Matrisel çarpım %*% operatörüyle yapılmaktadır. Örneğin:


```

> a <- cbind(c(1, 3, 5), c(3, 6, 2), c(2, 7, 3))
> b <- cbind(c(2, 5, 1), c(4, 1, 5), c(5, 1, 7))
> c <- a %*% b
> a
      [,1] [,2] [,3]
[1,]    1    3    2
[2,]    3    6    7
[3,]    5    2    3
> b
      [,1] [,2] [,3]
[1,]    2    4    5
[2,]    5    1    1
[3,]    1    5    7

```

Tabii bu işlemde soldaki matrisin satır sayısının sağdaki matrisin sütun sayısı ile aynı olması gerekir. Örneğin:

```

> a <- cbind(c(1, 3, 5), c(3, 6, 2), c(2, 7, 3))
> b <- cbind(c(1, 3), c(3, 6), c(2, 7))
> c <- a %*% b
Error in a %*% b : non-conformable arguments

```

t isimli fonksiyon matris transpozesi yapmaktadır. Örneğin:

```

> a <- cbind(c(1, 3, 5), c(3, 6, 2), c(2, 7, 3))
> a
      [,1] [,2] [,3]
[1,]    1    3    2
[2,]    3    6    7
[3,]    5    2    3
> b <- t(a)
> b
      [,1] [,2] [,3]
[1,]    1    3    5
[2,]    3    6    2
[3,]    2    7    3

```

diag fonksiyonu matrisin köşegenini bize vektör olarak verir.

Bir kare matrisin ters matrisi solve fonksiyonuyla elde edilir. Örneğin:

```

> a <- cbind(c(1, 3, 5), c(3, 6, 2), c(2, 7, 3))
> a
      [,1] [,2] [,3]
[1,]    1    3    2
[2,]    3    6    7
[3,]    5    2    3
> solve(a)
      [,1] [,2] [,3]
[1,]  0.1176471 -0.1470588  0.26470588
[2,]  0.7647059 -0.2058824 -0.02941176
[3,] -0.7058824  0.3823529 -0.08823529

```

İki parametrelili solve fonksiyonu doğrusal denklem sistemini çözmektedir. Bu fonksiyon $AX = B$ biçimindeki doğrusal denklem sisteminde A kare matrisini ve B sütun vektörünü parametre olarak alır. Geri dönüş değeri olarak denklem çözümüne ilişkin X vektörünü verir.

```

> a <- cbind(c(1, 3, 5), c(3, 6, 2), c(2, 7, 3))
> b <- c(12, 5, 7)
> solve(a, b)
[1]  2.529412  7.941176 -7.176471

```

R'da Listeler (Lists)

Biz şimdiye kadar yalnızca atomik vektör kullandık. Atomik vektörler elemanları aynı türden olan (aynı moddan olan) bir dizi gibi düşünülebilir. Örneğin bir atomik vektörün bazı elemanları tamsayı bazı elemanları string olamaz:

```
> a <- c(10, 20, "ali")
> a
[1] "10" "20" "ali"
```

Burada da görüldüğü gibi görünüşte c fonksiyonu ile biz vektörü farklı türlerden elemanlar kullanarak oluşturduğumuzu sanabiliriz. Halbuki c fonksiyonu bu girilen elemanları ortak bir türe (örnekte string'e) dönüştürmektedir.

Vektör R'da elemanlarına bir indeks ile erişilebilen veri yapısı için kullanılan bir kavramdır. Listeler aslında elemanları farklı türlerden olabilen vektörlerdir. Şimdiye kadar görmüş olduğumuz normal vektörlere R'da "atomik vektörler" denilmektedir. Listeler atomik olmayan vektörlerdir.

Listeler tipik olarak list isimli fonksiyonla oluşturulmaktadır. Örneğin:

```
> a <- list("Ali Serçe", 123)
> a
[[1]]
[1] "Ali Serçe"

[[2]]
[1] 123
```

Tabii listelerin farklı türlerden eleman tutması aslında onların farklı türlerden vektör tutması anlamına gelmektedir. Yani bir listenin elemanları birden fazla elemana sahip vektörlerden oluşturulabilir. Örneğin:

```
> l <- list(c("Ali Serçe", "Necati Ergin"), c(123, 456))
> l
[[1]]
[1] "Ali Serçe" "Necati Ergin"

[[2]]
[1] 123 456
```

R'da atomik bir vektörün elemanları başka bir vektörden oluşamaz. Yani örneğin bir vektör başka vektörleri tutamaz. Bu işlem listelerle yapılmaktadır. Aşağıdaki çağrıda vektörleri tutan bir vektör değil tüm değerlerden oluşan tek bir vektör elde edilmektedir:

```
> a <- c(c(1, 2, 3), c(2, 3, 5), c(4, 8, 9))
> a
[1] 1 2 3 2 3 5 4 8 9
```

Bir listenin elemanlarına isimler verebiliriz. Bunun için list fonksiyonunda argümanlar isim = değer biçiminde girilmelidir. Örneğin:

```
> person <- list(name = "Ali Serçe", number = 123)
> person
$name
[1] "Ali Serçe"

$number
[1] 123
```

Örneğin:

```
> l <- list(names = c("Ali Serçe", "Necati Ergin", "Oğuz Karan"), numbers = c(123, 234, 678))
> l
$names
[1] "Ali Serçe"      "Necati Ergin"   "Oğuz Karan"

$numbers
[1] 123 234 678
```

Genellikle liste elemanlarına isim verilmektedir. Listeler bileşik nesnelerdir. Bunların da parçaları (elemanları) vardır. Listenin belli bir elemanına erişmek için \$, [] ya da [[]]operatörü kullanılır. \$ operatörünün solunda liste değişkeninin ismi sağında listenin eleman ismi bulunur. Tabii bu operatörün kullanılabilmesi için liste elemanlarına isim verilmiş olmalıdır. Örneğin:

```
> person <- list(name = "Ali Serçe", no = 123)
> person$name
[1] "Ali Serçe"
> person$no
[1] 123
```

Listenin elemanları o elemanın türündendir. Örneğin yukarıdaki listede person\$name aslında bir elemanlı bir string vektörü, person\$no da bir elemanlı numeric bir vektördür.

Listenin elemanlarına \$ operatörü ile erişilirken isim kısmi eşleşmeyle verilebilir. Örneğin:

```
> person <- list(name = "Ali Serçe", no = 123)
> person$na
[1] "Ali Serçe"
> person$n
NULL
> person$no
[1] 123
```

Görüldüğü gibi kısmi eşleşme sonucunda birden fazla eleman elde ediliyorsa sonuç NULL değer olarak verilmektedir.

Bir listenin elemanlarına tek köşeli parantez ile erişebiliriz. Tek köşeli parantezin içerisine bir indeks vektörü getirilebilir. Bu vektör tek elemanlı ya da çok elemanlı olabilir. Ancak tek köşeli parantez bize listenin bazı elemanlarını yeni bir liste olarak vermektedir. Halbuki \$ operatörü bize listenin ilgili elemanını o türden verir. Örneğin:

```
> person <- list(name = "Ali Serçe", no = 123, bdate = "10/10/1970")
> subperson <- person[c(1, 2)]
> subperson
$name
[1] "Ali Serçe"

$no
[1] 123
```

Burada görüldüğü gibi person[c(1, 2)] ifadesi ile biz person listesinin 1'inci ve 2'inci indislerindeki bilgileri alarak bunu yeni bir liste (alt liste de diyebiliriz) biçiminde elde ettik. Örneğin:

```
> l <- list(name = "Ali Serçe", number = 123, email = "serce@csystem.org")
> k <- l[1:2]
> l
$name
[1] "Ali Serçe"

$number
[1] 123
```

```
$email  
[1] "serce@csystem.org"
```

```
> k  
$name  
[1] "Ali Serçe"
```

```
$number  
[1] 123
```

```
> typeof(l)  
[1] "list"  
> typeof(k)  
[1] "list"
```

\$ operatörünün sağına bir indis getiremeyiz. Yalnızca tam eşleşen ya da kısmi eşleşen eleman ismi getirebiliriz.

Tabii listenin elemanlarına isim vermemişsek \$ operatörünü kullanamayız ancak tek köşeli parantez ile yine onun bazı elemanlarına erişebiliriz. Örneğin:

```
> person <- list("Ali Serçe", 123, "10/10/1970")  
> person[1]  
[[1]]  
[1] "Ali Serçe"  
  
> typeof(person[1])  
[1] "list"
```

Listelerin elemanlarına çift köşeli parantez operatörüyle de erişilebilir. Ancak daha önceden de görüldüğü gibi çift köşeli parantezin içerisindeki indeks vektörünün tek elemanlı olması zorunludur. Çift köşeli parantez bize listenin ilgili elemanını tıpkı \$ operatöründe olduğu gibi o elemanın türünden verir. (Halbuki tek köşeli parantez operatörünün ilgili elemanı liste olarak verdiğini anımsayınız.) Örneğin:

```
> person <- list("Ali Serçe", 123, "10/10/1970")  
> person[1]  
[[1]]  
[1] "Ali Serçe"  
  
> typeof(person[1])  
[1] "list"  
  
> person[[1]]  
[1] "Ali Serçe"  
  
> typeof(person[[1]])  
[1] "character"  
  
> person[[c(1, 2)]]  
Error in person[[c(1, 2)]] : subscript out of bounds
```

Elemanlara isim verilmişse tek köşeli parantezde de çift köşeli parantezde de elemanlara erişmek için string vektör kullanılabilir. Ancak tek köşeli parantezde de çift köşeli parantezde de kısmi eşleşme uygulanmamaktadır. Çift köşeli parantezde eleman ismi geçersizse NULL değeri elde edilir. Örneğin:

```
> person <- list(name = "Ali Serçe", no = 123, bdate = "10/10/1970")  
> person[c("name", "bdate")]  
$name  
[1] "Ali Serçe"  
  
$bdate  
[1] "10/10/1970"
```

```
> person[["no"]]
[1] 123
> person[["n"]]
NULL
```

Örneğin:

```
> l <- list(c(1, 2, 3), c(4, 5, 6), c(7, 8, 9))
> l
[[1]]
[1] 1 2 3

[[2]]
[1] 4 5 6

[[3]]
[1] 7 8 9

> l[[1]]
[1] 1 2 3
> l[[2]]
[1] 4 5 6
> l[[3]]
[1] 7 8 9
```

Tabii listenin bir atomik vektör elemanına \$ ya da çift köşeli parantez operatörüyle eriştikten sonra onun da elemanlarına yine tek köşeli parantez ya da çift köşeli parantez operatörleriyle erişebiliriz. Örneğin:

```
> l <- list(one = c(1, 2, 3), two = c(4, 5, 6), three = c(7, 8, 9))
> l[[2]][2]
[1] 5
> l$three[3]
[1] 9
> l[["one"]][1]
[1] 1
```

Tek köşeli parantezin ve çift köşeli parantezin içerisinde eleman isimlerinin de string vektör olarak getirilebildiğini anımsayınız. Ancak eğer tek köşeli parantez ya da çift köşeli parantez içerisinde eleman ismi kullanılacaksa bunun tam eşleşme oluşturmaya gerektirir. Bu durumlarda kısmi eşleşme kullanılamamaktadır.

Peki neden \$ operatörü ve [[]] operatörü bize listenin elemanlarını o türden (örneğin atomik vektör) verirken [] operatörü bir liste olarak vermektedir? İşte \$ operatöründe ve [[]] operatöründe seçilen eleman bir tanedir. Tek bir elemanın liste biçiminde verilmesine gerek yoktur. Onun o türden (örneğin atomik vektör türünden) verilmesi daha uygundur. Ancak [] operatörünün içerisine biz birden fazla elemanlı vektör yerleştirebileceğimize göre bu operatörle biz farklı türlerden birden fazla liste elemanını çekebiliriz. Bunlar da ancak bir liste ile temsil edilebilir.

O halde liste elemanlarına erişme işleminin kuralları şöyle özetlenebilir:

- 1) Elemana \$ operatörüyle erişilecekse bu operatörün sağında indeks bulunamaz, yalnızca isim bulunabilir. Ancak isim tam eşleşen ya da kısmi eşleşen biçimde olabilir. \$ operatöründe eleman o elemanın türünden olacak biçimde elde edilmektedir.
- 2) Elemanlara tek köşeli parantez ile erişilecekse tek köşeli parantezin içerisinde bir indeks vektörü ya da isim vektörü bulunabilir. Bu durumda vektörün eleman sayısı birden fazla olabilir. Ancak bu operatör bize listenin elemanlarını bir liste biçiminde vermektedir. Tek köşeli parantez içerisindeki isim vektöründe kısmi eşleşme uygulanamaz.

3) Elemana çift köşeli parantez ile erişilecekse çift köşeli parantezin içerisinde indeks vektörü ya da isim vektörü bulunabilir. Ancak bu vektörün tek elemanı olmak zorundadır. Yine isimde kısmi eşleşme uygulanmaz. Çift köşeli parantez ile erişimde eleman bize o elemanın türünden verilir.

Bir listenin elemanlarına liste yaratıldıktan sonra da names isimli yer değiştirme fonksiyonuyla da isim verilebilir. Örneğin:

```
> l <- list("Ali Serçe", 123, "serce@csystem.org")
> l
[[1]]
[1] "Ali Serçe"

[[2]]
[1] 123

[[3]]
[1] "serce@csystem.org"

> names(l) <- c("name", "number", "email")
> l
$name
[1] "Ali Serçe"

$number
[1] 123

$email
[1] "serce@csystem.org"
```

Atomik vektörlerde olduğu gibi listelerde de listenin olmayan elemanlarına \$, [] ya da [[]] operatörleriyle atama yapılırsa bu durum listeye eleman ekleme anlamına gelmektedir. Örneğin:

```
> l <- list(name = "Ali Serçe")
> l$number <- 123
> l
$name
[1] "Ali Serçe"

$number
[1] 123
```

Örneğin:

```
> l <- list(name = "Ali Serçe")
> l[2:3] <- c(123, "serce@csystem.org")
> l
$name
[1] "Ali Serçe"

[[2]]
[1] "123"

[[3]]
[1] "serce@csystem.org"
```

Örneğin:

```
> l <- list(name = "Ali Serçe")
> l[c("no", "bdate")] <- c(123, 1968)
> l
$name
```

```
[1] "Ali Serçe"
```

```
$no
```

```
[1] 123
```

```
$bdate
```

```
[1] 1968
```

Listenin yüksek indisli bir elemanlarına değer atanırsa düşük indisli yeni elemanlara NULL değeri yerleştirilir. Örneğin:

```
> l <- list(name = "Ali Serçe")
```

```
> l[[5]] <- 123
```

```
> l
```

```
$name
```

```
[1] "Ali Serçe"
```

```
[[2]]
```

```
NULL
```

```
[[3]]
```

```
NULL
```

```
[[4]]
```

```
NULL
```

```
[[5]]
```

```
[1] 123
```

Listeden belli elemanları silmek için o elemanlara NULL değeri atamak yeterlidir. Örneğin:

```
> l <- list(name = "Ali Serçe", number = 123, email = "serce@csystem.org")
```

```
> l["number"] <- NULL
```

```
> l
```

```
$name
```

```
[1] "Ali Serçe"
```

```
$email
```

```
[1] "serce@csystem.org"
```

Örneğin:

```
> l <- list(name = "Ali Serçe", number = 123, email = "serce@csystem.org")
```

```
> l[c("number", "email")] <- NULL
```

```
> l
```

```
$name
```

```
[1] "Ali Serçe"
```

Örneğin:

```
> l <- list(name = "Ali Serçe", number = 123, email = "serce@csystem.org")
```

```
> l[2:3] <- NULL
```

```
> l
```

```
$name
```

```
[1] "Ali Serçe"
```

Listeler de bir çeşit vektördür. Dolayısıyla birden fazla listeyi c fonksiyonuyla birleştirebiliriz. Örneğin:

```
> a <- list(name = "Ali Serçe", number = 123)
```

```
> b <- list(email = "serce@csystem.org")
```

```
> c <- c(a, b)
```

```
> c
```

```

$name
[1] "Ali Serçe"

$number
[1] 123

$email
[1] "serce@csystem.org"

```

Bir listenin içerisindeki elemanlar unlist fonksiyonuyla vektör olarak elde edilebilir. Ancak listenin farklı türden elemanları işlem sırasında aynı türe (örneğin string türüne) dönüştürülmektedir. Örneğin:

```

> l <- list(name = "Kaan Aslan", no = 123)
> v <- unlist(l)
> v
      name      no
"Kaan Aslan"  "123"

```

Örneğin:

```

> l <- list(c(1, 2, 3), c(4, 5, 6))
> a <- unlist(l)
> a
[1] 1 2 3 4 5 6

```

Örneğin:

```

> l <- list(a = c(1, 2, 3), b = c(4, 5, 6))
> a <- unlist(l)
> a
a1 a2 a3 b1 b2 b3
 1  2  3  4  5  6

```

Örneğin:

```

> l <- list(a = c(x = 1, y = 2, z = 3), b = c(k = 4, m = 5, t = 6))
> l
$a
x y z
1 2 3

$b
k m t
4 5 6

> a <- unlist(l)
> a
a.x a.y a.z b.k b.m b.t
 1  2  3  4  5  6

```

Bir listenin tüm elemanlarına lapply fonksiyonu ile belli bir fonksiyonu uygulayabiliriz. lapply fonksiyonu parametresi ile aldığı listenin elemanlarını verdiğimiz fonksiyona sokar, elde ettiği değerleri yine aynı elemanlara sahip bir liste biçiminde bize geri dönüş değeri olarak verir. Örneğin:

```

> l <- list(a = 1:10, b = 2:20)
> x <- lapply(l, sum)
> lapply(l, sum)
$a
[1] 55

$b
[1] 209

```


Burada lapply fonksiyonunun geri dönüş değeri olarak yine bir liste verdiği dikkat ediniz. Örneğin:

```
> l <- list(age = c(34, 56, 38, 41, 23), salary = c(7000, 3200, 4800, 5500, 3250))
> result <- lapply(l, mean)
> result
$age
[1] 38.4

$salary
[1] 4750
```

Tabii listenin tüm elemanlarının lapply ile uygulanan fonksiyonun parametrik yapısına uygun olması gerekir. Örneğin mean fonksiyonu numerik verilerle çalışır. Biz buna string vektör verirsek o bize NA değerini verecektir. Örneğin:

```
> l <- list(names = c("Ali Serçe", "Veli Akkuş", "Secaattin Tanyerli"), age = c(34, 56, 38, 41, 23), salary = c(7000, 3200, 4800, 5500, 3250))
> result <- lapply(l, mean)
Warning message:
In mean.default(X[[i]], ...) :
  argument is not numeric or logical: returning NA
> result
$names
[1] NA

$age
[1] 38.4

$salary
[1] 4750
```

sapply fonksiyonu lappy fonksiyonu gibi kullanılmaktadır. Ancak sapply sonucu bize bir liste olarak değil atomik vektör olarak verir. Örneğin:

```
> l <- list(age = c(34, 56, 38, 41, 23), salary = c(7000, 3200, 4800, 5500, 3250))
> sapply(l, sum)
  age salary
192 23750
> result <- sapply(l, sum)
> typeof(result)
[1] "double"
> class(result)
[1] "numeric"
```

Bir listenin elemanı atomik bir vektör olmak zorunda değildir. Başka bir liste de olabilir. Örneğin:

```
> date <- list(day = 17, month = 12, year = 1993)
> person <- list(name = "Sacit Aydın", number = 1234, bdate = date)
> person
$name
[1] "Sacit Aydın"

$number
[1] 1234

$bdate
$bdate$day
[1] 17

$bdate$month
[1] 12
```

```
$bdate$year  
[1] 1993
```

Tabii aynı işlemi daha kompakt biçimde şöyle yapabiliydik:

```
> person <- list(name = "Sacit Aydın", number = 1234, bdate = list(day = 17, month = 12, year = 1993))  
> person  
$name  
[1] "Sacit Aydın"  
  
$number  
[1] 1234  
  
$bdate  
$bdate$day  
[1] 17  
  
$bdate$month  
[1] 12  
  
$bdate$year  
[1] 1993
```

R Paketlerinin Kurulması ve Kullanıma Hazır Hale Getirilmesi

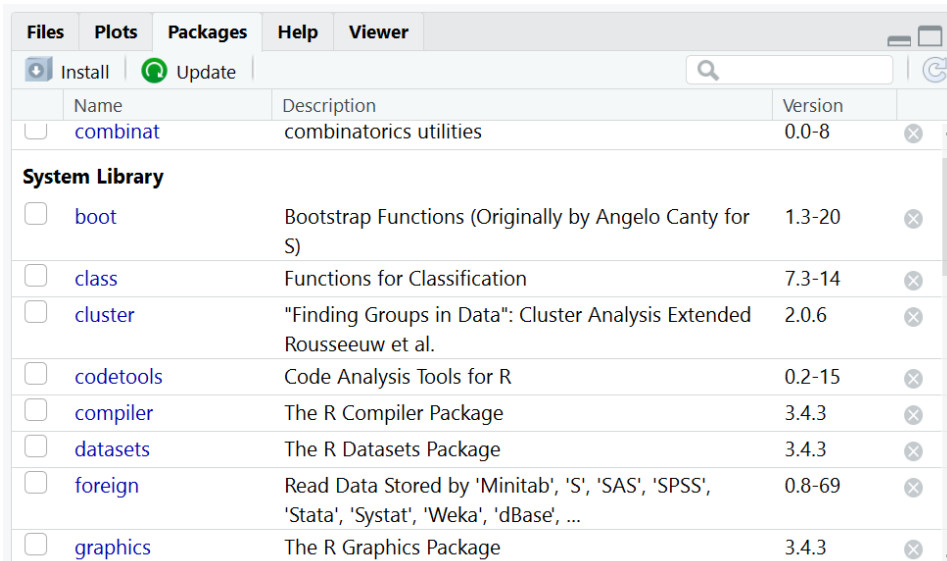
R'daki fonksiyonlar ve sınıflar paketler biçiminde organize edilmiştir. Buradaki paket kavramı işlevsel olarak Python'daki, Java'daki paketlere ya da .NET'teki DLL'lere benzetilebilir. R kurulduğunda temel sistem paketleri de kurulmuş durumdadır. Ancak programcı yüzlerce CRAN paketinden isteklerini istediği zaman da kurabilmektedir. Bir paketi indirip kurmak için `install.packages` fonksiyonu kullanılmaktadır. (Fakat bu fonksiyonu çağırmak yerine aynı işlem RStudio'da Tools/Install Packages menüsüyle de yapılabilir. Tabii bu durumda aslında RStudio da bu fonksiyonu çağırılmaktadır.) Her paketin bir ismi vardır. `install.packages` fonksiyonunu paketin ismini vererek çağırabiliriz:

```
install.packages("combinat")
```

Kurulan bir paket `remove.packages` fonksiyonuyla silinebilir:

```
remove.packages("combinat")
```

Tabii bu işlem RStudio'da Packages sekmesiyle de görsel biçimde yapılabilir:



O anda sistemde yüklü olan paketler `installed.packages` fonksiyonuyla görüntülenebilir. `install.packages` argüman girilmeden çağrılabilir. Bu durumda fonksiyonun geri dönüş değeri olarak kurulmuş paketleri bize bir matris biçiminde verir. Örneğin:

```
> installed.packages()
      Package
combinat  "combinat"
base      "base"
boot      "boot"
class     "class"
cluster   "cluster"
codetools "codetools"
compiler  "compiler"
datasets  "datasets"
foreign   "foreign"
graphics  "graphics"
grDevices "grDevices"
grid       "grid"
KernSmooth "KernSmooth"
lattice    "lattice"
MASS       "MASS"
Matrix     "Matrix"
methods    "methods"
mgcv       "mgcv"
nlme       "nlme"
nnet       "nnet"
...        ...
```

Mevcut bir paketin yeni versiyonu güncellenebilir. Bu işlem de `update.packages` fonksiyonuyla yapılmaktadır.

Bir paket kurulduktan sonra onun kullanıma hazır hale getirilmesi için `library` fonksiyonun çağrılması gerekmektedir. `library` fonksiyonunu Python ve Java'daki `import`, C#'taki `using` ve C++'taki `include` direktiflerine benzetebiliriz. `library` fonksiyonun birinci parametresi kullanıma hazır hale getirilecek paketin ismini ikinci parametresi ise bunun yerini almaktadır. İkinci parametre girilmezse fonksiyon paketi R sistemi tarafından belirlenen default dizinde arar. Örneğin biz "combinat" isimli paketi şöyle kullanıma hazır hale getirebiliriz:

```
library("combinat")
```

Bu işlem RStudio'da ilgili paketin çarpılanmasıyla da görsel biçimde yapılabilir.

Files Plots Packages Help Viewer

Install Update

Name

Description

Version

User Library

☒

combinat

combinatorics utilities

0.0-8

System Library

☐

boot

Bootstrap Functions (Originally by Angelo Canty for S)

1.3-20

☐

class

Functions for Classification

7.3-14

☐

cluster

"Finding Groups in Data": Cluster Analysis Extended Rousseeuw et al.

2.0.6

☐

codetools

Code Analysis Tools for R

0.2-15

☐

compiler

The R Compiler Package

3.4.3

☐

datasets

The R Datasets Package

3.4.3

☐

foreign

Read Data Stored by 'Minitab', 'S',

0.8-69

Yeni bir paket kullanıma hazır hale getirildiğinde (yüklendiğinde) eğer bu pakette daha önce yüklenmiş olan paketlerdeki fonksiyonlarla aynı isimli fonksiyonlar varsa yeni yüklenen paketteki fonksiyon eski fonksiyonları gizler. Örneğin default olarak yüklenmiş olan `utils` isimli sistem paketinde `combn` isimli bir fonksiyon vardır. `combinat` paketinde de aynı isimli bir fonksiyon bulunmaktadır. Eğer biz `library` fonksiyonuyla `combinat` paketini yüklersek artık `combn` ismini yalnız olarak kullandığımızda bu `combinat` paketindeki `combn` anlamına gelecektir. İşte böylesi çakışma durumlarında ismi niteliklendirerek belli bir paketteki fonksiyonu belirtebilir. R'daki paketleri C# ve C++'taki isim alanlarına (name space) benzetilebiliriz. R'da niteliklendirme için C++'taki gibi `::` operatörü kullanılmaktadır. Örneğin:

```
utils::combn(...)
combinat::combn(...)
```

Bu durumda `combinat` paketini yükledikten sonra artık yalnızca `combn` ismini kullandığımızda bu `combinat` paketindeki `combn` olacaktır. Biz `utils` paketindeki `combn` fonksiyonunu `utils::combn` ifadesiyle kullanabiliriz. Tabii bir çakışma olmadıktan sonra niteliklendirme yapmak da gereksizdir.

Permütasyon Kombinasyon İşlemleri

Kombinasyon işlemleri aslında R'ın temel paketlerinden biri olan "`utils`" paketi içerisinde bulunan `combn` fonksiyonuyla yapılabilmektedir. Ancak maalesef temel paketler içerisinde permütasyon işlemlerini yapan bir fonksiyon yoktur. Bunun için "`combinat`" ve "`gtools`" paketleri yaygın olarak kullanılmaktadır.

"`combinat`" paketi kurulup yüklendikten sonra onun fonksiyonlarını kullanabiliriz. Kombinasyon işlemleri için `combn` fonksiyonu kullanılmaktadır. Fonksiyonun parametrik yapısı şöyledir:

```
combn(x, m, fun = NULL, simplify = TRUE, ...)
```

Fonksiyonun birinci parametresi kombinasyon işlemine sokulacak vektörü alır. İkinci parametre kaçlı kombinasyonların elde edileceğini belirtir. Örneğin:

```
> combinat::combn(c(1, 2, 3), 2)
      [,1] [,2] [,3]
[1,]    1    1    2
[2,]    2    3    3
```

`combn` verilen kümenin kombinasyonlarını sütun temelinde bir matris olarak bize vermektedir. Örneğin:

```

> result <- combinat::combn(c("ali", "veli", "selami", "ayşe", "Fatma"), 4)
> result
      [,1]      [,2]      [,3]      [,4]      [,5]
[1,] "ali"     "ali"     "ali"     "ali"     "veli"
[2,] "veli"    "veli"    "veli"    "selami"  "selami"
[3,] "selami"  "selami"  "ayşe"   "ayşe"   "ayşe"
[4,] "ayşe"    "Fatma"   "Fatma"  "Fatma"  "Fatma"
> t(result)
      [,1] [,2]      [,3]      [,4]
[1,] "ali" "veli"    "selami" "ayşe"
[2,] "ali" "veli"    "selami" "Fatma"
[3,] "ali" "veli"    "ayşe"   "Fatma"
[4,] "ali" "selami"  "ayşe"   "Fatma"
[5,] "veli" "selami" "ayşe"   "Fatma"

```

combn fonksiyonunun üçüncü parametresi tüm bu kombinasyonların belli bir fonksiyona sokulmasını sağlar. Böylece her kombinasyon belli bir fonksiyona sokularak bunların sonuçları bir vektör olarak verilir. Örneğin:

```

> combinat::combn(c(1, 2, 3), 2)
      [,1] [,2] [,3]
[1,]    1    1    2
[2,]    2    3    3
> v <- combinat::combn(c(1, 2, 3), 2, sum)
> v
[1] 3 4 5

```

combn fonksiyonu ile Merkezi Limit Teoremini (Central Limit Theorem) sınavabiliriz. Merkezi Limit Teoremine göre ana kütle nasıl dağılmış olursa olsun ana kütleden çekilen n’li örnekler normal dağılmıştır. n sayısı arttıkça bu normallik daha belirgin olmaya başlar. $n > 30$ için bu normallik çok iyi düzeye gelir. Eğer ana kütle zaten normal dağılmışsa $n < 30$ olsa bile her zaman n’li örneklerin ortalaması normal dağılır. n’li örneklerin ortalamasının normal dağılımı aşağıdaki parametrelere sahiptir:

Örneklem ortalaması = Ana kütle Ortalaması

Örneklem Varyansı = Ana kütle standart sapması / \sqrt{n}

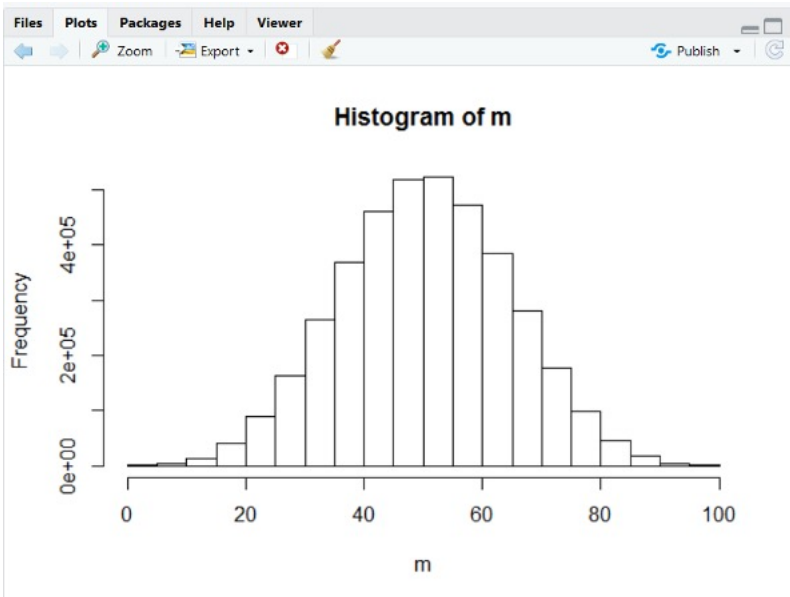
Bu durumda bizim örneklem dağılımına bakarak ana kütle hakkında güven aralıkları oluşturabilmemiz için ana kütle varyansını da bilmemiz gerekir. Ancak eğer örneklem büyüklüğü olan n değeri 30’dan büyükse örneklem varyansı anakütle varyansına çok yaklaşmaktadır. Bu durumda bizim $n \geq 30$ koşulunu sağladıktan sonra ana kütle varyansını bilmemize gerek kalmaz.

Merkezi limit teoremini R’da şöyle sınavabiliriz:

```

> m <- combn(1:100, 4, mean)
> hist(m)
> mean(m)
[1] 50.5

```



combn paketi içerisindeki diğer bir faydalı fonksiyon da permn fonksiyonudur. Bu fonksiyonun kullanımı combn ile çok benzerdir. Ancak bu fonksiyon permütasyon hesaplar. Örneğin:

```
> permn(1:3)
[[1]]
[1] 1 2 3

[[2]]
[1] 1 3 2

[[3]]
[1] 3 1 2

[[4]]
[1] 3 2 1

[[5]]
[1] 2 3 1

[[6]]
[1] 2 1 3
```

permn fonksiyonu bizden bir vektör alır ve o vektörün tüm permütasyonlarını bir listeye yerleştirir. Kombinasyon ve permütasyon işlemleri için gtools paketinden de faydalanılabilmektedir. Paketteki combinations fonksiyonu default olarak satır temelinde kombinasyonlardan oluşan matris verir. Örneğin:

```
> a <- combinations(5, 3, 1:5)
> a
      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    1    2    4
[3,]    1    2    5
[4,]    1    3    4
[5,]    1    3    5
[6,]    1    4    5
[7,]    2    3    4
[8,]    2    3    5
[9,]    2    4    5
[10,]   3    4    5
```

Fonksiyonun birinci parametresi üçüncü parametresinde belirtilen vektörün ilk kaç elemanından kombinasyon hesaplanacağını belirtir. İkinci parametresi kaçlı kombinasyonların oluşturulacağını belirtir. Üçüncü parametre ise kombinasyonun uygulanacağı vektörü almaktadır. permutations fonksiyonu da benzer biçimde kullanılmaktadır. Örneğin:

```
> a <- permutations(3, 2, 1:3)
> a
      [,1] [,2]
[1,]     1     2
[2,]     1     3
[3,]     2     1
[4,]     2     3
[5,]     3     1
[6,]     3     2
```

R'da Çalışma Dizininin (Current Working Directory) Alınması ve Değiştirilmesi

Bilindiği gibi prosesin çalışma dizini (current working directory) görelî yol ifadelerinin çözülmesinde kullanılmaktadır. Çalışma dizini bir dosya ya da dizini belirtirken kullanılan yol ifadesi mutlak değilse bir orijin görevini görmektedir. Örneğin "test.txt" gibi bir yol ifadesi işletim sistemi tarafından ilgili prosesin (örneğin R'ın) çalışma dizinin de aranmaktadır.

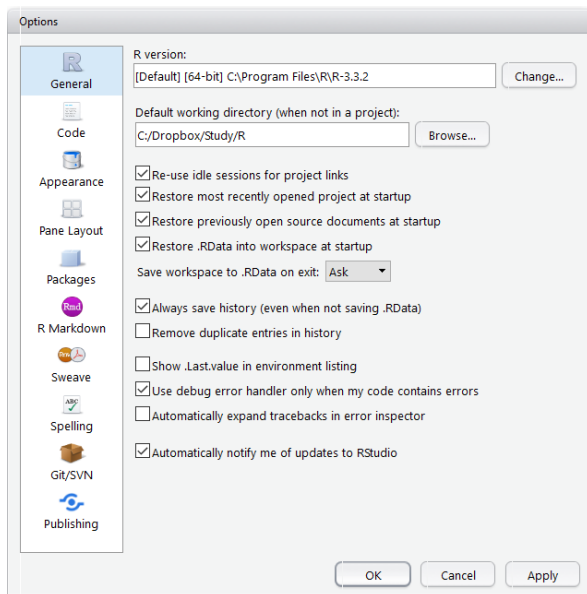
R'da default çalışma dizini getwd fonksiyonuyla elde edilebilir. Örneğin:

```
> getwd()
[1] "C:/Dropbox/Study/R"
```

setwd fonksiyonuyla bu dizin değiştirilebilir. Örneğin:

```
> setwd("C:\\Dropbox\\Shared\\Kurslar\\Arcelik-BigData\\Example Files")
> getwd()
[1] "C:/Dropbox/Shared/Kurslar/Arcelik-BigData/Example Files"
> a <- scan("data.txt")
Read 7 items
> a
[1] 10.0 12.4 45.6 23.0 34.0 45.0 49.0
```

RStudio açıldığında default çalışma dizininin istediğimiz bir dizin olmasını sağlamak için Tools/Global Options/Default working directory menüsü kullanılır.



Dosyadan ve Klavyeden Okuma Yapmak İçin Kullanılan scan Fonksiyonu

R'da çeşitli dosya formatlarından bilgi okumak için kullanılan pek çok hazır fonksiyon vardır. Böylece bu sayede biz bir text dosyadan, bir excel dosyasından, bir veritabanı dosyasından, bir SPSS dosyasından verileri alıp R ortamına aktararak işleyebiliriz.

Text bir dosyadan veri okumak için kullanılan en genel fonksiyonlardan biri scan fonksiyonudur. scan fonksiyonun parametrik yapısı şöyledir:

```
scan(file = "", what = double(), nmax = -1, n = -1, sep = "",
      quote = if(identical(sep, "\n")) "" else "'\"", dec = ".",
      skip = 0, nlines = 0, na.strings = "NA",
      flush = FALSE, fill = FALSE, strip.white = FALSE,
      quiet = FALSE, blank.lines.skip = TRUE, multi.line = TRUE,
      comment.char = "", allowEscapes = FALSE,
      fileEncoding = "", encoding = "unknown", text, skipNul = FALSE)
```

scan fonksiyonu dosyadan bilgileri okur bize bir vektör olarak verir. Fonksiyonun ilk parametresi olan "file" okunacak dosyanın yol ifadesini belirtmektedir. Bu parametre için boş string girilirse bu durum klavyeden (stdin dosyasından) okuma yapılacağı anlamına gelir. Örneğin:

```
> a <- scan("")
1: 123
2: 234
3: 567
4: 23.4
5: 45.6
6:
Read 5 items
> a
[1] 123.0 234.0 567.0 23.4 45.6
```

scan fonksiyonu normal olarak boşluk karakterlerini dikkate alarak dosya içerisindeki karakter öbeklerini ayırıştırır ve onları birer sayı olarak double türden bir vektöre yerleştirir geri dönüş değeri olarak bize onu verir. scan fonksiyonuyla yol ifadesi vererek bir text dosyadan okuma yapabiliriz. Örneğin:

```
> a <- scan("C:\\Dropbox\\Shared\\Kurslar\\Arcelik-BigData\\Example Files\\data.txt")
Read 7 items
> a
[1] 10.0 12.4 45.6 23.0 34.0 45.0 49.0
```

scan fonksiyonunun what parametresi okunacak bilginin türünü (dolayısıyla da geri dönüş değeri olarak verilecek atomik vektörün türünü) belirtmektedir. what parametresi için double() değerinin default argüman olarak verildiğini görüyorsunuz. what parametresi olarak şu argümanlar girilebilir

```
integer()
logical()
double()
character()
complex()
raw()
list()
```

Örneğin:

```
test.txt

1 2 3.1 4 5
6 7.1 8 9 10
```



```
> v <- scan("test.txt")
Read 7 items
> v
[1] 1.0 2.0 3.0 5.0 4.3 5.0 6.0
> typeof(v)
[1] "double"
```

İçerisinde noktalı sayıların bulunduğu bir dosyayı biz integer() olarak okuyamayız. Örneğin:

```
> v <- scan("test.txt", what = integer())
Error in scan("test.txt", what = integer()) :
  'an integer' scan() beklendi , 4.3 alındı
```

Ya da örneğin içerisinde yazı olan bir dosyayı nümerik türlerle okuyamayız.

Yazı okumaları için what parametresi character() olarak girilmelidir. Yine default durumda yazılar boşluk karakterleriyle birbirinden ayrılmaktadır. Örneğin:

test.txt

```
bugün hava
      çok
güzel
```

```
> v <- scan("test.txt", what = character())
Read 4 items
> v
[1] "bugün" "hava"  "çok"   "güzel"
> typeof(v)
[1] "character"
```

Ancak yazı çift tırnak içerisinde alınırsa boşluk karakterleri ayıraç olmaktan çıkartılır. Örneğin:

test.txt

```
bugün "hava
      çok"
güzel
```

```
> v <- scan("test.txt", what = character())
Read 3 items
> v
[1] "bugün"          "hava \n      çok" "güzel"
```

scan fonksiyonunda default olarak boşluk karakterleri ayıraç olarak kullanılmaktadır. sep parametresi ile ayıraç karakteri değiştirilebilir. Ancak her zaman "\n" karakteri (eğer string okumalarında iki tırnak içerisinde alınmadıysa) ayıraç olarak kullanılmaktadır. Örneğin:

test.txt

```
10,20,30,40,
50,60,70
80,90,100
```

```
> v <- scan("test.txt", sep = ",")
Read 11 items
> v
[1] 10 20 30 40 NA 50 60 70 80 90 100
```

Burada sep parametresi için "," karakteri girilmiştir. 40 değerinden sonra NA geldiğine dikkat ediniz. Satır sonları (yani "\n" karakterleri) de ayıraç olarak kullanıldığından ve 40'dan sonra satır sonuna kadar bir değer olmadığı için buradaki NA olmayan değeri belirtmektedir. Ayrıca sayısal okumalarda sep parametresi boşluk içermese bile "\n" karakterinin dışındaki boşluk karakterleri tamamem ihmal edilmektedir. Örneğin:

```
test.txt
```

```
1 0, 20 ,30 ,40, 50, 6 0,70  
80, 90, 100
```

```
> v <- scan("test.txt", sep = ",")  
Read 10 items  
> v  
[1] 10 20 30 40 50 60 70 80 90 100
```

sep parametresine girilecek argüman olan stringin tek bir karakter içermesi zorunludur. Eğer sep parametresine birden fazla argüman girilirse scan bunların ilkinin dikkate almaktadır.

sep fonksiyonunun n parametresi en fazla kaç adet değerin okunacağını belirtir. Bu parametre sayesinde biz dosyadan daha az değer okuyabiliriz. Örneğin:

```
test.txt
```

```
10 20 30 40 50 60 70 80 90 100
```

```
> v <- scan("test.txt", n = 5)  
Read 5 items  
> v  
[1] 10 20 30 40 50
```

skip parametresi ile dosyanın başındaki belli sayıda satırın atlanmasını sağlayabiliriz. Örneğin:

```
test.txt
```

Değerler şunlardır:

```
10 20 30 40 50 60 70 80 90 100
```

```
> v <- scan("test.txt", skip = 1)  
Read 10 items  
> v  
[1] 10 20 30 40 50 60 70 80 90 100
```

Bir dosyayı satır satır aşağıdaki gibi okuyabiliriz. Örneğin:

Fonksiyonun diğer parametreleri dokümanlardan izlenebilir.

readline Fonksiyonu

Bu fonksiyon klavyeden bir satır okumak için kullanılmaktadır. Parametrik yapısı şöyledir:

```
readline(prompt = "")
```

Fonksiyon parametre olarak bir prompt yazısı alabilmektedir. Örneğin:

```
> readline("Bir isim giriniz:")  
Bir isim giriniz:Ali Serçe  
[1] "Ali Serçe"
```

file ve url Fonksiyonları

R'da dosya gibi işlem gören kaynaklar "connection" termiyle temsil edilmiştir. Bazı genel fonksiyonlar bizden "connection" isterler ve okumayı oradan yaparlar. İki temel connection oluşturan fonksiyon file ve url fonksiyonlarıdır. Bu fonksiyonların parametrik yapıları şöyledir:

```
file(description = "", open = "", blocking = TRUE,
      encoding = getOption("encoding"), raw = FALSE,
      method = getOption("url.method", "default"))
```

```
url(description, open = "", blocking = TRUE,
     encoding = getOption("encoding"),
     method = getOption("url.method", "default"))
```

Fonksiyonların birinci zorunlu parametreleri sırasıyla dosyanın yol ifadesi ve URL bilgisidir. Örneğin:

```
f <- file("test.txt")
```

Örneğin:

```
https://textfiles.com/art/angi.txt
```

Bir connection nesnesini alarak faydalı işlemler yapan çeşitli fonksiyonlar vardır. Örneğin readLines isimli fonksiyon connection'daki satırları okuyarak bize onu bir string vektör biçiminde verir. Örneğin:

```
> f <- file("test.txt")
> v <- readLines(f)
> v
[1] "bugün hava"      "çok güzel"      "evet çok güzel"
```

Bu fonksiyonların ayrıntılarını R dokümanlarından inceleyebiliriz.

R'da Data Frame'ler

Data Frame satırları eşit sayıda vektörlerden oluşan özel listelerdir. Yani data frame'ler aslında birer listedir. Bu listelerin elemanları eşit sayıda elemandan oluşan atomik vektör biçimindedir. Pek çok matematiksel ve istatistiksel tablolar R'da birer data frame olarak temsil edilirler.

Bir data frame yaratmak için data.frame fonksiyonu kullanılır. Fonksiyonun parametrik yapısı şöyledir:

```
data.frame(..., row.names = NULL, check.rows = FALSE,
           check.names = TRUE, fix.empty.names = TRUE,
           stringsAsFactors = default.stringsAsFactors())
```

Fonksiyona biz eşit sayıda elemanlardan oluşan vektörler girebiliriz. Örneğin:

```
> names <- c("Kaan Aslan", "Ali Serçe", "John Lennon", "Sacit Bulut", "Ayşe Er")
> numbers <- c(123, 513, 800, 230, 256)
> df <- data.frame(names, numbers)
> df
```

	names	numbers
1	Kaan Aslan	123
2	Ali Serçe	513
3	John Lennon	800
4	Sacit Bulut	230
5	Ayşe Er	256

Görüldüğü gibi bir data frame sütunlardan oluşmaktadır. Sütunların isimleri yukarıdaki örnekte sütunları oluşturan vektörler değişkenlerinden alınmıştır. Ancak biz açıkça sütunlara isimler de verebiliriz. Örneğin:

```
> df <- data.frame(names = c("Kaan Aslan", "Ali Serçe", "John Lennon", "Sacit Bulut", "Ayşe Er"), numbers = c(123, 513, 800, 230, 256))
> df
  names numbers
1 Kaan Aslan   123
2  Ali Serçe   513
3 John Lennon   800
4 Sacit Bulut   230
5  Ayşe Er     256
```

Data frame'ler aslında birer listedir. Dolayısıyla listeler anlatılan tüm özellikler data frame'ler için de geçerlidir. Yani örneğin data frame'lerin de elemanlarına \$ operatörü ile, [] ve [[]] operatörleriyle erişebiliriz. Örneğin:

```
> df$names
[1] Kaan Aslan  Ali Serçe  John Lennon Sacit Bulut Ayşe Er
Levels: Ali Serçe Ayşe Er John Lennon Kaan Aslan Sacit Bulut
> df[2]
  numbers
1     123
2     513
3     800
4     230
5     256
> df[[1]]
[1] Kaan Aslan  Ali Serçe  John Lennon Sacit Bulut Ayşe Er
Levels: Ali Serçe Ayşe Er John Lennon Kaan Aslan Sacit Bulut
> df[1:2]
  names numbers
1 Kaan Aslan   123
2  Ali Serçe   513
3 John Lennon   800
4 Sacit Bulut   230
5  Ayşe Er     256
```

data.frame fonksiyonu default durumda string vektörleri birer faktör kabul eder. Faktörler konusunu sonraki bölümde ele alınacaktır. Ancak data.frame fonksiyonun string vektörleri faktör olarak ele almaması için stringAsFactors parametresi FALSE girilmelidir. Örneğin:

```
> df <- data.frame(names = c("Kaan Aslan", "Ali Serçe", "John Lennon", "Sacit Bulut", "Ayşe Er"), numbers = c(123, 513, 800, 230, 256), stringsAsFactors = F)
```

O halde bir data frame yukarıda da belirtildiği gibi aslında eşit sayıda elemanlardan oluşan bir listedir.

Data frame'ler aynı zamanda matris gibi de kullanılabilir. Oysa normal listeler böyle kullanılamazlar. Biz bir data frame'de satır ve sütun indeksleri vererek tıpkı matris gibi onun herhangi bir elemanına erişebiliriz. Örneğin:

```
> df <- data.frame(names = c("Kaan Aslan", "Ali Serçe", "John Lennon", "Sacit Bulut", "Ayşe Er"), numbers = c(123, 513, 800, 230, 256), stringsAsFactors = F)
> df$names[2]
[1] "Ali Serçe"
> df[1, 2]
[1] 123
> df[3:4, 1:2]
  names numbers
3 John Lennon   800
4 Sacit Bulut   230
```

Biz bir data frame'e matrislerde yaptığımız gibi filtremeler uygulayabiliriz. Örneğin:

```
> df <- data.frame(names = c("Kaan Aslan", "Ali Serçe", "John Lennon", "Sacit Bulut", "Ayşe Er"), numbers = c(123, 513, 800, 230, 256), stringsAsFactors = F)
> df[df$numbers > 500,]
      names numbers
2  Ali Serçe    513
3 John Lennon    800
```

Burada filtreleme sonucunda elde edilen ürün de bir data frame'dir.

Peki data frame'lerin matrislerden ne farkı vardır? Matrisler tüm elemanları aynı türden neslerdir. Halbuki data frame'lerin sütunları farklı türlerden olabilmektedir.

Data frame'ler aslında birer liste olduğuna göre biz onlara yeni elemanlar (sütunlar) ekleyebiliriz. Örneğin:

```
> df$scores = c(4, 6, 2, 7, 8)
> df
      names numbers scores
1 Kaan Aslan    123     4
2  Ali Serçe    513     6
3 John Lennon    800     2
4 Sacit Bulut    230     7
5  Ayşe Er     256     8
```

Bu durumda biz bir data frame'in iki sütununu işleme sokarak üçüncü bir sütun elde edebiliriz. Örneğin:

```
> df$difference <- df$numbers-df$scores
> df
      names numbers scores difference
1 Kaan Aslan    123     4        119
2  Ali Serçe    513     6        507
3 John Lennon    800     2        798
4 Sacit Bulut    230     7        223
5  Ayşe Er     256     8        248
```

Sütunsal yapıdaki text dosyalardan bilgiler read.table isimli fonksiyonla bir data frame olarak okunabilirler. Örneğin text.txt dosyası elle aşağıdaki gibi oluşturulmuş olsun ya da excel gibi bir programla elde edilmiş olsun:

```
Adı      No
Kaan     123
Ali      234
Sacit    678
Ayşe     345
```

Burada sütunlar eşit uzunlukta fakat farklı türlerdendir. İşte read.table fonksiyonu bunu bize bir data frame olarak okur. Örneğin:

```
> df <- read.table("test.txt", header = TRUE, stringsAsFactors = F)
> df
      Adı  No
1 Kaan 123
2  Ali 234
3 Sacit 678
4 Ayşe 345
```

Genellikle bu tür tabloların başına bir başlık bulunur. İşte read.table fonksiyonunun header parametresi dosyanın başında böyle bir başlık olup olmadığını belirtmekte kullanılmaktadır.

Bir data frame'e biz cbind ya da rbind fonksiyonuyla satır ya da sütun ekleyebiliriz. Ancak satır eklenirken onun sütunları farklı türden olduğuna göre eklemenin bir liste biçiminde yapılması gerekmektedir. Örneğin:

```
> df <- read.table("test.txt", header = TRUE, stringsAsFactors = F)
> df
  Adı No
1 Kaan 123
2 Ali 234
3 Sacit 678
4 Ayşe 345
> df <- rbind(df, list("Hasan", 345))
> df
  Adı No
1 Kaan 123
2 Ali 234
3 Sacit 678
4 Ayşe 345
5 Hasan 345
```

Yukarıdaki listede biz ortalamadna büyük numaraya sahip olan satırları elde etmek isteyelim:

```
> df
  Adı No
1 Kaan 123
2 Ali 234
3 Sacit 678
4 Ayşe 345
5 Hasan 345
> df[df$No > mean(df$No),]
  Adı No
3 Sacit 678
```

Microsoft Excel Dosyasından Veri Okumak

R'da Microsoft Excel dosyasından veri okuyan değişik paketlerde pek çok fonksiyon bulunmaktadır. Bu paketlerin bazıları başka birtakım araçları kullanmaktadır. Dolayısıyla o paketler kullanılacaksa o araçların da (örneğin Perl gibi) yüklü olması gerekir. Genel olarak Excel dosyasından doğrudan okuma yapan fonksiyonlar biraz yavaş çalışma eğilimindedir. Pek çok R programcısı bu nedenle Excel dosyasından doğrudan veri okumak yerine o dosyayı metin dosyası biçiminde save edip read.table fonksiyonuyla ya da read.csv fonksiyonuyla dolaylı olarak okumayı yaparlar.

Biz burada xlsx isimli paketi kullanarak Excel dosyasından doğrudan okuma yapacağız. Bunun için önce xlsx paketini install.packages fonksiyonuyla yükleyip library fonksiyonuyla kullanıma hazır hale getirmemiz gerekir. xlsx paketi Excel dosyasından okumayı "Java" kullanarak yapmıştır. Bu nedenle sistemde Java Run time Environment'in yüklü olması gerekmektedir.

xlsx paketindeki read.xlsx fonksiyonu ilgili Excel dosyasından okumayı yapar ve onu bize data frame olarak verir. Fonksiyonda biz hangi dosyanın hangi sheet'inin hangi kolonlarının okunacağını belirtebilmekteyiz. read.xlsx fonksiyonunun parametrik yapısı şöyledir:

```
read.xlsx(file, sheetIndex, sheetName = NULL, rowIndex = NULL,
  startRow = NULL, endRow = NULL, colIndex = NULL,
  as.data.frame = TRUE, header = TRUE, colClasses = NA,
  keepFormulas = FALSE, encoding = "unknown", ...)
```

Fonksiyonun birinci parametresi Excel dosyasının yol ifadesini, ikinci parametresi okunacak sheet numarasını almaktadır. startRow, endRow, rowIndex ve colIndex parametreleri okunacak bölgeyi belirtir. Örneğin:

```
> df <- read.xlsx("Test.xlsx", 1, encoding = "UTF-8")
> df
  Names Numbers
1 Ali Serçe 123
2 Veli Ballı 345
```

3	Sacit Hiçyılmaz	7374
4	Ayşe Er	834

Burada biz tüm Excel tablosunu data frame olarak okumuş olmaktadır.

İstatistikte Ölçek Türleri

Bir niteliği ölçme iddiasında olan araçlara ölçme araçları ve elde edilen sonucun anlamlandırılması için kullanılan sisteme de ölçek (scale) denilmektedir. Ölçme sonucunda elde ettiğimiz bilgiler genellikle sayısaldır. Ancak sayısal olmayan bilgiler de söz konusudur. Örneğin cinsiyet gibi, eğitim durumu gibi, doğum yeri gibi bilgiler sayısal değildir. Bunlar kimi zaman sayılarla temsil edilse bile sayısal bir anlam içermezler. Örneğin doğum yeri ilgili şehrin plaka numarasıyla ifade edilebilir. Ancak bu plaka numarasının belirttiği sayı sayı olarak yorumlanmaz. Ölçekler tipik olarak dört bölüme ayrılmaktadır.

Nominal (Kategorik) Ölçekler: Bunlar birer kategori belirtirler. Örneğin cinsiyet, coğrafi bölge gibi.

Sıralı (Ordinal) Ölçekler: Bu ölçekler kategorik olmakla birlikte bunların aralarında bir büyüklük küçüklük ilişkisi tanımlanmıştır. Örneğin eğitim durumu (ilköğretim, ortaöğretim, yüksek öğretim gibi) sıralı bir ölçekle ifade edilmektedir. Benzer biçimde askeri rütbelere, günün aşamaları (sabah, öğle akşam gibi) bu tür ölçeklerdendir.

Aralıklı (Interval) Ölçekler: Bu tür ölçeklerde iki puan arasındaki fark aynı miktar uzaklığı yakınlığı ifade eder. Örneğin bir testte 20 puan alan 10 puan alandan belli miktarda daha iyidir. 30 puan alan da 20 puan alandan aynı miktar kadar daha iyidir. Bu tür ölçeklerde mutlak sıfır noktası yoktur. Alınan puanlar her zaman belli bir orijine göre geçerlidir. Örneğin aslında sınavlardan alınan puanlar böyle bir ölçek türündedir. Sınavdan sıfır alınabilir. Ancak bu sıfır o kişinin o konu hakkında hiçbir şey bilmediği anlamına gelmez. Yani mutlak sıfır değildir.

Oransal (Ratio) Ölçekler: Bunlar da aralıklı ölçeklerin tüm özelliklerine sahiptirler. Ancak bunlarda mutlak bir sıfır noktası vardır. Dolayısıyla puanlar arasındaki oranlar mutlak olarak anlamlıdır. Örneğin uzunluk, kütle gibi temel fiziksel özellikler oransal ölçek türlerindendir.

Hipotez Nedir?

Hipotez aslında değişkenler arasında bir ilişki cümlesidir. Bu ilişki doğru ya da yanlış olabilir. Örneğin:

- Erkekler kadınlardan daha çok şiddete eğilimlidir.
- Marmara bölgesi Doğu Anadolu Bölgesinden YGS sınavında daha başarılıdır.
- Dizel arabalar benzinli arabalardan daha ekonomiktir.

Görüldüğü gibi hipotezlerde birtakım değişkenler ve onların ilişkin olduğu ölçek türleri vardır. Örneğin "Marmara Bölgesi Doğu Anadolu Bölgesinden YGS sınavında daha başarılıdır" hipotezinde iki değişken söz konusudur. Birinci değişken bölge belirtir ve tür olarak nominal (kategoriktir). İkinci değişken YGS sınavından alınan puanlarla temsil edilebilir. Bu ölçek de aralıklı türündedir. İşte ölçeklerin türleri tespit edilip veriler toplandıktan sonra hipotezin sınanması için istatistiksel hipotez testleri yapılmaktadır. Bu hipotez testleriyle hipotez belli bir güven aralığı içerisinde doğrulanmakta ya da yanlışlanmaktadır.

R'da Faktörler

R'da faktörler istatistikteki nominal yani kategorik ölçekleri temsil etmek için kullanılmaktadır. Örneğin Kadın-Erkek gibi, eğitim durumu gibi bilgiler kategorik ölçeklere örnek olarak verilebilir. Faktörler pek çok programlama dilindeki enum sabitlerini de çağrıştırmaktadır.

R'da faktörler factor isimli bir fonksiyonla oluşturulur. factor fonksiyonunun parametrik yapısı şöyledir:

```
factor(x = character(), levels, labels = levels,  
       exclude = NA, ordered = is.ordered(x), nmax = NA)
```

Fonksiyonun birinci parametresi faktör verilerini belirten atomik bir vektördür. İkinci parametre faktörü oluşturan dereceleri (level) başka bir deyişle kategorileri belirtir. Örneğin:

```
> f <- factor(c(10, 12, 24, 24, 10, 11))
> f
[1] 10 12 24 24 10 11
Levels: 10 11 12 24
```

Faktör bilgileri sayısal gibi görünse bile aslında sayısal değil kategoriktir. Yani yukarıdaki örnekte 10, 11, 12 ve 24 değerleri aslında birer sayısal büyüklük değil kavramsal bir kategori anlatmaktadır. Örneğin:

```
> f <- factor(c("erkek", "kadın", "erkek", "erkek", "kadın"), levels = c("erkek", "kadın"))
> f
[1] erkek kadın erkek erkek kadın
Levels: erkek kadın
```

Bir faktörde olmayan bir kategoriye (yani level'e) atama yapılamaz. Örneğin:

```
> f <- factor(c("erkek", "kadın", "erkek", "erkek", "kadın"), levels = c("erkek", "kadın"))
> f[1] <- "xxxx"
Warning message:
In `[<-factor`(`*tmp*`, 1, value = "xxxx") :
  invalid factor level, NA generated
> f
[1] <NA> kadın erkek erkek kadın
Levels: erkek kadın
```

Fakat örneğin:

```
> f <- factor(c("erkek", "kadın", "erkek", "erkek", "kadın"), levels = c("erkek", "kadın"))
> f
[1] erkek kadın erkek erkek kadın
Levels: erkek kadın
> f[1] <- "erkek"
> f
[1] erkek kadın erkek erkek kadın
```

Kategoriler kategori verilerden daha fazla farklılık içerebilir. Bu durum daha sonra bu kategorilerin kullanılabileceği anlamına gelir. Örneğin:

```
> f <- factor(c(1, 2, 1, 1, 2), levels = c(1, 2, 3, 4, 5))
> f
[1] 1 2 1 1 2
Levels: 1 2 3 4 5
> f[6] <- 4
> f[7] <- 5
> f
[1] 1 2 1 1 2 4 5
Levels: 1 2 3 4 5
```

factor fonksiyonun labels parametresi nihai kategori isimlerini belirlemekte kullanılır. Örneğin biz excel'den sütunu çekerken Kadınlar için K, Erkekler için E biçiminde çekmiş olabiliriz. Ancak biz Kadınlar için "Kadın", Erkekler için "Erkek" ibaresinin kullanılmasını da isteyebiliriz.

```
> df <- read.xlsx("test.xlsx", 1, header = T)
> df
  Cinsiyet X.Boy
1         E   172
2         K   167
```



```

3      K    140
4      E    185
> df$Cinsiyet <- factor(df$Cinsiyet, levels = c("E", "K"), labels = c("Erkek", "Kadın"))
> df$Cinsiyet
[1] Erkek Kadın Kadın Erkek
Levels: Erkek Kadın

```

factor fonksiyonunda bizim levels için değer vermemiz gerekmez. Bu durumda kategoriler otomatik olarak verilen değerlerden oluşturulur. Örneğin:

```

> df <- factor(c(1, 2, 3, 3, 2, 1))
> df
[1] 1 2 3 3 2 1
Levels: 1 2 3

```

split isimli fonksiyon bir data frame'i ve bir factor'ü alır. Onun faktörlerini ayrı ayrı gruplayarak bize data frame'lerden oluşan bir liste biçiminde verir. Örneğin:

```

> df <- read.xlsx("test.xlsx", 1, header = T)
> a<- split(df, df$Cinsiyet)
> a
$E
  Cinsiyet X.Boy
1         E   172
4         E   185

$K
  Cinsiyet X.Boy
2         K   167
3         K   140

```

Böylece biz tabloyu kategorilere göre gruplamış olmaktadır.

Daha önce biz lapply ve sapply fonksiyonlarını görmüştük. Faktörler söz konusu olduğunda tapply fonksiyonu da çok kullanılmaktadır. Bu fonksiyon bizden sırasıyla bir hesaplanacak vektörü ve bir faktörü alır. Sonra o faktöre sahip değerleri ayrı vektörlerde verdiğimiz fonksiyona sokar. Sonucu da bize bir vektör olarak verir. Örneğin:

```

> tapply(df$X.Boy, df$Cinsiyet, mean)
      E      K
178.5 153.5

```

String Fonksiyonları

R'da string'ler üzerinde işlemler yapan çeşitli fonksiyonlar hazır biçimde bulunmaktadır. Bu bölümde bu fonksiyonlar üzerinde duracağız. Bu fonksiyonların bazıları "base" paket içerisinde. Yani ana R yüklemesiyle bu fonksiyonlar zaten kullanıma hazır durumdadır. Ancak bunun dışında string işlemleri yapan başka çeşitli paketler de vardır. Bunların en fazla kullanılanlarında birisi stringr paketidir.

nchar isimli fonksiyon string'in karakter uzunluğunu bize verir. Fonksiyonun parametrik yapısı şöyledir:

```
nchar(x, type = "chars", allowNA = FALSE, keepNA = NA)
```

Fonksiyon bizden string vektörünü alır aynı uzunlukta bir tamsayı vektörü verir. Bu vektörün elemanları da string vektöründeki string'lerin karakter uzunluklarını belirtmektedir. Örneğin:

```

> names <- c("ali", "veli", "selami", "ayşe", "fatma")
> v <- nchar(names)
> v
[1] 3 4 6 4 5

```

```
> str <- "test"
> nchar(str)
[1] 4
```

- substr fonksiyonu bir string'in belli bir kısmını elde etmek için kullanılmaktadır. Fonksiyonun parametrik yapısı şöyledir:

```
substr(x, start, stop)
```

Fonksiyon bizden bir string vektörü başlangıç ve bitiş indeksi ister, geri dönüş değeri olarak da bize aynı uzunlukta vektörün elemanları olan string'lerin ilgili kısımlarının bulunduğu bir vektör verir. İndeksler yine normal olarak R'da 1'den başlamaktadır. Örneğin:

```
> str <- "ankara"
> v <- substr(str, 3, 5)
> v
[1] "kar"
```

Örneğin:

```
> names <- c("ali", "veli", "selami", "ayşe", "fatma")
> v <- substr(names, 1, 3)
> v
[1] "ali" "vel" "sel" "ayş" "fat"
```

substr fonksiyonu ile biz ilgili kısmı değiştirebiliriz de. Örneğin:

```
> names <- c("ali", "veli", "selami", "ayşe", "fatma")
> names
[1] "ali"      "veli"     "selami"   "ayşe"     "fatma"
> substr(names, 1, 3) <- "xxx"
> names
[1] "xxx"      "xxxi"     "xxxami"   "xxxe"     "xxxma"
```

Örneğin:

```
> names <- c("ali", "veli", "selami", "ayşe", "fatma")
> names
[1] "ali"      "veli"     "selami"   "ayşe"     "fatma"
> substr(names, 1, 3) <- c("xxx", "yyy", "zzz", "kkk", "mmm")
> names
[1] "xxx"      "yyyi"     "zzzami"   "kkke"     "mmmma"
```

- strsplit fonksiyonu bir string içerisindeki yazıyı belli karakterlerden ayrıştırarak ayrıştırılmış kısımları bize bir string vektör biçiminde verir. Fonksiyonun parametrik yapısı şöyledir:

```
strsplit(x, split, fixed = FALSE, perl = FALSE, useBytes = FALSE)
```

Fonksiyonun birinci parametresi ayrıştırılacak string vektörünü ikinci parametresi ise ayrıştırma karakterlerini alır. Fonksiyon bize ayrıştırılmış kısımları bir liste olarak vermektedir. Listenin elemanları da ayrıştırılan string vektörleridir. Örneğin:

```
> str <- "ali,veli,selami,ayşe,fatma"
> str
[1] "ali,veli,selami,ayşe,fatma"
> l <- strsplit(str, ",")
> l
[[1]]
[1] "ali"      "veli"     "selami"   "ayşe"     "fatma"
```

Tabii ayrıştırılacak string vektörü daha fazla elemana sahip olabilir. Ayrıca ayrıştırma karakterleri de birden fazla olabilmektedir. Ancak ayrıştırma için bunların peşi sıra gelmesi gerekir. Örneğin:

```
> str <- c("ali, veli", "selami, ayşe", "fatma, sacit", "alparslan, fehmi")
> l <- strsplit(x, ", ")
Error in strsplit(x, ", ") : non-character argument
> l <- strsplit(str, ", ")
> l
[[1]]
[1] "ali" "veli"

[[2]]
[1] "selami" "ayşe"

[[3]]
[1] "fatma" "sacit"

[[4]]
```

Ayrıca istersek vektör içerisindeki her yazıyı ayrı ayrıç karakterleriyle de ayrıştırabiliriz. Örneğin:

```
> str <- c("ali,veli", "selami:ayşe", "fatma!sacit", "alparslan-fehmi")
> l <- strsplit(str, c(",",":","!", "-"))
> l
[[1]]
[1] "ali" "veli"

[[2]]
[1] "selami" "ayşe"

[[3]]
[1] "fatma" "sacit"

[[4]]
[1] "alparslan" "fehmi"
```

- as.integer ve as.double fonksiyonları yazıyı sayıya dönüştürmek için as.character fonksiyonu ise sayıyı yazıya dönüştürmek için kullanılmaktadır. Örneğin:

```
> str = "123"
> i <- as.integer(str)
> s <- as.character(i)
> i
[1] 123
> s
[1] "123"
```

- startsWith fonksiyonu bir yazının başının belli bir biçimde başlayıp başlamadığına endsWith ise yazının sonunun belli bir biçimde bitip bitmediğine bakmaktadır. Bu fonksiyonlar bize bool bir vektör verirler. Fonksiyonların parametrik yapıları şöyledir:

```
startsWith(x, prefix)
endsWith(x, suffix)
```

Örneğin:

```
> str <- "abrakadabra"
> startsWith(str, "abra")
[1] TRUE
> endsWith(str, "kadavra")
```

```
[1] FALSE
```

Örneğin aşağıdaki gibi bir "Test.txt" dosyamız olsun. Bu dosyayı bir data frame olarak okuyup A harfi ile başlayan isimlere ilişkin satırları elde etmek isteyelim:

Test.txt

Names	Numbers	Gender	EntryDate
"Kaan Aslan"	123	E	12/06/2003
"Ali Serçe"	521	E	11/07/2005
"Sacit Apaydın"	762	E	06/10/2015
"Merve Çetin"	817	K	11/01/2001
"Ayşe Er"	987	K	12/03/2008

Bu işlem şöyle yapılabilir:

```
> df <- read.table("Test.txt", header = T, stringsAsFactors = F)
> df
      Names Numbers Gender EntryDate
1  Kaan Aslan   123      E 12/06/2003
2   Ali Serçe   521      E 11/07/2005
3 Sacit Apaydın 762      E 06/10/2015
4   Merve Çetin 817      K 11/01/2001
5    Ayşe Er   987      K 12/03/2008
> df[startsWith(df$Names, "A"), ]
      Names Numbers Gender EntryDate
2  Ali Serçe   521      E 11/07/2005
5   Ayşe Er   987      K 12/03/2008
```

R'da düzenli ifadelerle ilişkin birkaç fonksiyon vardır. Ancak düzenli ifadeler (regular expression) burada ele alınmayacaktır.

- trimws fonksiyonu yazının başındaki ve/veya sonundaki boşluk karakterlerini atar. Örneğin:

```
> str <- "   ankara  "
> trimws(str)
[1] "ankara"
> trimws(str, which = "right")
[1] "   ankara"
```

stringr Paketi İçerisindeki String Fonksiyonları

Bu paketi kullanabilmek için önce paketin install.packages fonksiyonuyla indirilip library fonksiyonuyla kullanıma hazır hale getirilmesi gerekir. Paketteki fonksiyonların başı str_xxx ile başlamaktadır.

- str_length fonksiyonu base paketindeki nchar fonksiyonuyla aynı şeyi yapar. Yani string'in karakter uzunluklarıyla geri döner. Örneğin:

```
> names <- c("ali", "veli", "selami")
> v <- str_length(names)
> v
[1] 3 4 6
```

- str_split fonksiyonu strSplit ile benzerdir. Örneğin:

```
> names <- c("ali, veli", "Selami, Ayşe", "Fatma, Metin")
> l <- str_split(names, ", ")
> l
[[1]]
```

```
[1] "ali" "veli"

[[2]]
[1] "Selami" "Ayşe"

[[3]]
[1] "Fatma" "Metin"
```

- `str_to_lower` küçük harfe dönüştürme için, `str_to_upper` büyük harfe dönüştürmek için `str_to_title` her sözcüğün ilk harfini büyük yapmak için kullanılır. Örneğin:

```
> cities <- c("Ağrı Dağı", "Ankara ovası", "italyan pizzası")
> str_to_upper(cities)
[1] "AĞRI DAĞI" "ANKARA OVASI" "ITALYAN PIZZASI"
> str_to_lower(cities)
[1] "ağrı dağı" "ankara ovası" "italyan pizzası"
> str_to_title(cities)
[1] "Ağrı Dağı" "Ankara Ovası" "İtalyan Pizzası"
```

Burada Türkçe karakterlerin bozulduğuna dikkat ediniz. Bunun fonksiyonların `locale` parametresi ilgili dili belirtecek biçimde girilmelidir. Örneğin:

```
> cities <- c("Ağrı Dağı", "Ankara ovası", "italyan pizzası")
> str_to_upper(cities, locale = "tr")
[1] "AĞRI DAĞI" "ANKARA OVASI" "İTALYAN PIZZASI"
> str_to_lower(cities, locale = "tr")
[1] "ağrı dağı" "ankara ovası" "italyan pizzası"
> str_to_title(cities, locale = "tr")
[1] "Ağrı Dağı" "Ankara Ovası" "İtalyan Pizzası"
```

- `str_trim` fonksiyonu yazının başındaki ve sonundaki boşluk karakterlerini atar. Örneğin:

```
> str <- c(" ankara ", " izmir ")
> str
[1] " ankara " " izmir "
> str_trim(str)
[1] "ankara" "izmir"
```

- `str_dup` fonksiyonu bir strini çoklamaktadır. Örneğin:

```
> names <- c("ali", "veli", "selami")
> str_dup(names, 3)
[1] "alialiali" "veliveliveli" "selamiselaselami"
> str_dup(names, c(2, 3, 4))
[1] "aliali" "veliveliveli"
[3] "selamiselaselamiselami"
```

- `str_replace` fonksiyonu bir yazı içerisindeki belli bir kalıbı bularak onu başka bir yazıyla değiştirmek için kullanılır. Ancak bu fonksiyon yalnızca ilk bulunduğu kalıbı değiştirir. `str_replace_all` tüm kalıpları değiştirmektedir. Örneğin:

```
> str <- "ankara ankara güzel ankara"
> str_replace(str, "ankara", "eskişehir")
[1] "eskişehir ankara güzel ankara"
> str_replace_all(str, "ankara", "eskişehir")
[1] "eskişehir eskişehir güzel eskişehir"
```

Diğer fonksiyonlara `stringr` dokümanlarından erişilebilir.

R'da Dizinsel İşlemler

R'da dosya sistemine ilişkin dosya kopyalama silme gibi çeşitli işlemler base paket içerisindeki fonksiyonlarla yapılabilmektedir. Bu fonksiyonlar genellikle file.xxx biçiminde isimlendirilmiştir.

- file.exists fonksiyonu bizden bir string vektör alır. Bu vektörün içerisindeki dosyaların olup olmadığına ilişkin bool bir vektör verir. Örneğin:

```
> file.exists("test.txt")
[1] TRUE
> file.exists("mest.txt")
[1] FALSE
> file.exists("c:\\windows\\notepad.exe")
[1] TRUE
```

Örneğin:

```
> file.exists(c("text.txt", "mest.txt", "c:\\windows\\notepad.exe"))
+ )
[1] FALSE FALSE TRUE
```

- file.remove fonksiyonu dosya silmek için file.rename fonksiyonu ise dosyanın ismini değiştirmek için kullanılır. Örneğin:

```
> file.exists("xxx.txt")
[1] TRUE
> file.remove("xxx.txt")
[1] TRUE
> file.exists("xxx.txt")
[1] FALSE
```

Örneğin:

```
> file.rename("test.txt", "mest.txt")
[1] TRUE
> file.rename("mest.txt", "test.txt")
[1] TRUE
```

Bu fonksiyonlar geri dönüş değeri olarak işlemin başarısına ilişkin bool bir vektör verirler.

- file.copy fonksiyonu dosya kopyalaması için kullanılır. Örneğin:

```
> file.copy("test.txt", "mest.txt")
[1] TRUE
```

- list.files isimli fonksiyon dizin içerisindeki dosyaları, list.dir s isimli fonksiyon ise dizin içerisindeki dizinleri bize vermektedir. Örneğin:

```
> v <- list.files("c:\\windows")
> typeof(v)
[1] "character"
> v
[1] "addins"
[2] "appcompat"
[3] "AppPatch"
[4] "AppReadiness"
[5] "assembly"
[6] "ativpsrm.bin"
[7] "bcastdvr"
[8] "bfsvc.exe"
[9] "Boot"
[10] "bootstat.dat"
....
```

list.files fonksiyonu default olarak yalnızca ilgili dizindeki dosyaları bize verir. Halbuki list.dirs fonksiyonu alt dizinleri de kapsayacak biçimde dizin içerisindeki dizinleri bize vermektedir. Fonksiyonların pek çok ayrıntısı vardır. Ayrıntıları için R dokümanlarına başvurulabilir.

Tarih ve Zamanın Elde Edilmesi

O an içinde bulunduğumuz tarih Sys.Date fonksiyonu ile zaman ise Sys.time fonksiyonu ile elde edilebilir. Bu iki fonksiyon temel paketlerdedir. Fonksiyonların parametreleri yoktur. Örneğin:

```
> d <- Sys.Date()
> typeof(d)
[1] "double"
> class(d)
[1] "Date"
> d
[1] "2018-01-07"
> t <- Sys.time()
> t
[1] "2018-01-07 17:16:44 +03"
> typeof(t)
[1] "double"
> class(t)
[1] "POSIXct" "POSIXt"
```

Burada fonksiyonlar bize bilgisayarın saatinden geçerli tarihi ve zamanı vermektedir.

R'da Programlama

R prosedürel, fonksiyonel ve nesne yönelimli çok modellenli bir programlama dilidir. Ancak baskın özellik onun fonksiyonel bir dil olmasıdır. Fonksiyonel dillerde bazı işlemler onların prosedürel karşılıklarına göre çok kompakt biçimde yapılabilmektedir. R'da diğer programlama dillerindeki pek çok deyim aslında birer fonksiyon gibi işlem görmektedir. Fonksiyonlar birinci sınıf vatandaş (first class citizen). Yani fonksiyonlar normal türlerde olduğu gibi pek çok işleme sokulabilirler. Örneğin fonksiyonlar R'da atama işlemine sokulabilirler:

```
> f <- sum
> f(1:30)
[1] 465
```

R'da fonksiyon oluşturmak için function isimli fonksiyondan faydalanılır. Fonksiyon oluşturmanın genel biçimi şöyledir:

```
function([parametre listesi]) <ifade>
```

R'da değişkenlerin operatörlerin ve sabitlerin her bir kombinasyonuna ifade (expression) denilmektedir. Aslında operatörler de R'da birer fonksiyondur. Sonuç olarak R saf fonksiyonel bir dile yakındır. Bu nedenle ifadeler birer fonksiyon çağrısı olarak düşünülebilirler. R'da aslında diğer dillerdeki deyimler (statements) de birer fonksiyon statüsündedir. Bu nedenle R'da deyim terimi kullanılmaz. Her şey bir ifadedir.

Fonksiyonlar değişkenlere atanarak kullanılırlar. Diğer dillerdeki gibi önce bildirilip sonra çağrılmazlar. Örneğin:

```
> f <- function(x, y) x + y
> f(10, 20)
[1] 30
> f(1:10, 1:10)
[1] 2 4 6 8 10 12 14 16 18 20
```

Parametre değişkenlerinin türlerinin belirtilmediğine dikkat ediniz. Onların yalnızca isimleri belirtilmektedir. R dinamik tür sistemine sahip olduğu için parametre türlerinin bir anlamı yoktur. Parametreler fonksiyon çağrıldığında argümanlardan değerlerini alırlar. Yani türleri onlar çağrıldığında belirlenmektedir.

R'da iki küme parantezi arasındaki bölgeye blok denir. R'da bir blok içerisine sıfır tane ya da daha fazla ifade yerleştirilirse bloğun kendisi de bir ifade olur. Buna bileşik ifade denilmektedir. Yani bu durumda genel biçime bakıldığında şunlar söylenebilir: Eğer fonksiyonun gövdesi birden fazla ifadeden oluşuyorsa onu küme parantezleri ile bloklamak gerekir. Örneğin:

```
> f <- function(x, y) { temp = x + y
+ avg = temp/2
+ }
> result <- f(10, 20)
> result
[1] 15
```

R'da kodun ayrıştırılması (parse edilmesi) sürecinde önemli bir kural vardır. R ayrıştırıcısı (parser) karakterleri elde ettiğinde eğer o anlamlı bir ifade oluşturuyorsa artık ifadenin bittiğini düşünür. Eğer okunan karakterler anlamlı bir ifade oluşturmuyorsa okunmaya devam edilir. Örneğin:

```
> a <-
+ 10
+
>
```

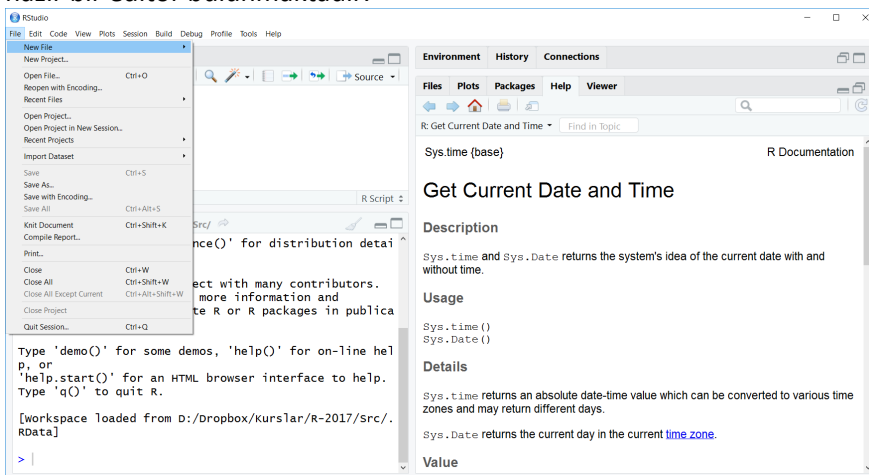
Burada programcı komut satırında ilk satırda a <- ifadesini girdiğinde bu henüz geçerli bir ifade oluşturmadığından R ayrıştırıcısı okumaya devam etmek istemiştir. Aşağı satırda çıkartılan "+" sembolü okumanın devam ettiğini göstermektedir. Daha sonra programcı 10 yazısını yazıp ENTER tuşuna bastığında artık ifade geçerli bir hale geldiği için ifade tamamlanmıştır. Örneğin:

Örneğin:

```
> a <-
+ 10 +
+ 20
>
```

Burada da süreç benzer biçimde devam etmiştir.

R'da biz komutları tek tek komut satırına yazmak zorunda değiliz. Onu bir dosyaya yazıp toptan o dosyadaki kodları çalıştırabiliriz. Buna "script tarzı çalışma" denilmektedir. Script çalışması için program kodu öncelikle bir editöre yazılır. Editör olarak Notepad gibi her hangi bir editör kullanılabilir. Ancak RStudio içerisinde zaten script yazabilmek için hazır bir editör bulunmaktadır.



RStudio'da bir script'i çalıştırmak için iki düğme vardır: Run ve source. Run düğmesi imlecin bulunduğu yerden itibaren kodu çalıştırır. Source düğmesi ise tüm kodu çalıştırır. Ayrıca kaynak dosyada bir kod parçasını seçtikten sonra da Run düğmesine basabiliriz. Bu durumda yalnızca o kod parçası çalıştırılır. Aslında Source düğmesine bastığımızda Source isimli fonksiyon çağrılmaktadır. Yani kodu çalıştırmak için Source isimli bir fonksiyon vardır.

R'da aynı satıra birden fazla geçerli ifade yazılacaksa bunların arasına ';' karakterinin konulması gerekir örneğin:

```
> f <- function(x, y) { temp = x + y; avg = temp / 2 }
> result <- f(10, 20)
> result
[1] 15
```

Başka bir deyişle R'da eğer satır sonuna kadar ifade geçerli değilse aşağı satırdan okuma devam etmektedir. Ancak satır sonuna kadar geçerli bir ifade görülürse artık ifadenin sonlandırıldığı varsayılmaktadır. Örneğin:

```
f <- function(a, b)
{
  temp <-
    a +
    b
  temp / 2
}
```

Gibi fonksiyon bildirimi geçerlidir. Bu kuralların dışında değişkenlerin, sabitlerin ve operatörlerini arasına istenildiği kadar boşluk karakterleri yerleştirilebilir.

Bildirilmi olan bir fonksiyon çağrılabilir. Çağırma işlemi sırasında daha önceden de gördüğümüz gibi parametreler için argümanlar kullanılmaktadır. Örneğin:

```
f <- function(a, b) a + b
```

Biz f fonksiyonunu şöyle çağırabiliriz:

```
f(10, 20)
```

Bir fonksiyon çağrıldığında argümanlardan parametre değişkenlerine otomatik atama yapılır. Örneğimizde 10 değeri a'ya 20 değeri de b'ye atanarak fonksiyonun gövdesi çalıştırılmaktadır.

Bir fonksiyon kabaca parametrelerden (parameters) ve bir de gövdeden (body) oluşmaktadır. Gövde fonksiyonu oluşturan ifadedir. Örneğin:

```
f <- function(a, b)
{
  temp <- a + b; temp / 2
}
```

Burada fonksiyonun parametreleri a ve b gövdesi de { temp <- a + b; temp / 2 } biçimindedir. Fonksiyon çağrıldığında çağırılma ifadesindeki argümanlar parametrelerle daha önce belirtilen kurallara göre eşleştirilir sonra da fonksiyonun gövdesi çalıştırılır. Yukarıda da belirtildiği gibi eğer fonksiyonun gövdesi tek bir ifade içeriyorsa hiç bloklama yapmaya gerek yoktur.

Bir fonksiyon çağrıldıktan sonra onun gövdesi çalıştırılınca o gövdede belli işlemleri yapan ifadeler sırasıyla çalıştırılmaktadır. Fonksiyonlar nasıl dışarıdan parametreler yoluyla değer alıyorsa yine dış dünyaya değer de iletebilmektedir. Buna fonksiyonun geri dönüş değeri (return value) denilmektedir. Örneğin:

```
f <- function(a, b)
{
  a + b
}
```

Burada a ve b fonksiyonun dışarıdan aldığı değerleri temsil eden parametreleridir. $a + b$ ifadesi de geri dönüş değerini belirtir. Yani bu fonksiyon çağrıldıktan sonra bu değer elde edilecektir. O halde buradaki fonksiyon, parametreleriyle aldığı vektörlerin toplamına geri dönmektedir.

R'da bir fonksiyonun oluşturulması ve çağırılması farklı anlamlardadır. Bir fonksiyon oluşturulduğunda bir değişkene atanır. Böylece fonksiyonun kodu çalıştırmaya hazır bir biçimde o değişkenin içerisinde bekletilmektedir. Ancak bu kod fonksiyon çağrıldığında çalıştırılır. Örneğin:

```
foo <- function()
{
  print("foo")
}
```

Bu kod çalıştırıldığında yalnızca bir atama işlemi gerçekleşir. Fonksiyonun içerisindeki kod çalıştırılmaz. Fonksiyonun içerisindeki kod fonksiyon çağrıldığında çalıştırılacaktır. Yani biz bu fonksiyonun kodunu çalıştırmak için onu aşağıdaki gibi çağırmalıyız:

```
foo()
```

R'da fonksiyonların geri dönüş değerleri fonksiyonda işletilen son ifadeyle belirlenmektedir. Örneğin:

```
avg <- function(a, b)
{
  temp <- a + b
  temp / 2
}
```

Buradaki fonksiyonda son işletilen değer $temp / 2$ olduğu için bu fonksiyonun geri dönüş değeri de $temp / 2$ 'dir. R'da etkisiz kodlar geçersiz kabul edilmemektedir. Örneğin aşağıdaki fonksiyonda $temp / 2$ ifadesi etkisiz bir kod haline gelmiştir. Fonksiyonun geri dönüş değeri son ifade 10 olduğu için 10 olacaktır:

```
avg <- function(a, b)
{
  temp <- a + b
  temp / 2
  10
}
```

Örneğin:

```
pi <- function()
{
  a <- seq(1, 1000000, 4)
  positives <- 1/a
  b <- seq(3, 1000000, 4)
  negatives <- -1/b
  sum(positives + negatives) * 4
}
```

Burada pi isimli fonksiyon herhangi bir parametre almamaktadır. Bu fonksiyonu çağırdığımızda pi sayısını elde ederiz:

```
> pi()
[1] 3.141591
```

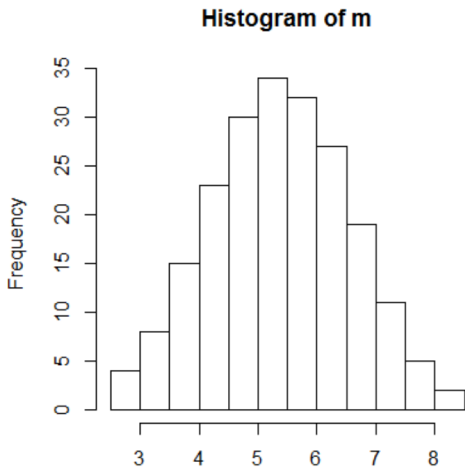
Örneğin bir vektörün belli sayıda alt kümelerinden bir bir histogram oluşturan fonksiyonu şöyle yazabiliriz:

```
g <- function(v, k)
{
  m <- combn(v, k, mean)
```

```
hist(m)
}
```

Burada fonksiyonun `v` parametresi ana kütle, `k` parametresi de kaçlı alt kümelerden histogram oluşturulacağını belirtmektedir. Fonksiyonu şöyle çağırabiliriz:

```
> g(1:10, 4)
```



Örneğin standart sapma hesaplayan kendi fonksiyonumuzu şöyle yazabiliriz:

```
mysd <- function(x) sqrt(sum((x - mean(x))^2) / (length(x) - 1))
```

Fonksiyonumuzu test edelim:

```
> sd(c(1, 3, 4, 4))
[1] 1.414214
> mysd(c(1, 3, 4, 4))
[1] 1.414214
```

R'da fonksiyonlar diğer türler gibi atama işlemine sokulabilirler. Bu özellik fonksiyonel dillerin çoğunda benzer biçimde bulunmaktadır. Ancak prosedürel ve nesne yönelimli popüler dillerin çoğu bu özelliğe sahip değildir. Örneğin:

```
f <- function(a, b)
{
  a + b
}
```

```
> f(10, 20)
[1] 30
> g <- f
> g(10, 20)
[1] 30
```

R'da fonksiyonların "birinci sınıf vatandaş" olması demekle onların diğer türler gibi atama işlemlerine sokulabilmesi anlatılmaktadır.

R'da eğer istenirse (bazı durumlarda mecbur kalınabilmektedir) fonksiyonun geri dönüş değeri `return` fonksiyonuyla da oluşturulabilmektedir. `return` fonksiyonunun kullanımı şöyledir:

```
return (ifade)
```

Örneğin:

```
f <- function(a, b)
{
  temp <- a + b
  return(temp / 2)
}
```

return fonksiyonu hem geri dönüş değerini oluşturur hem de fonksiyonu sanlandırır. Fakat R'da gerekmedikçe return kullanmamak genellikle iyi teknik olarak kabul edilmektedir.

R'da if İfadesi

R'da if diğer dillerde olduğu gibi bir deyim (statement) değil fonksiyon statüsündedir. Dolayısıyla R standartlarında if bir ifade olarak değerlendirilmektedir. if ifadesinin genel biçimi şöyledir:

```
if (ifade1) ifade2 [else ifade3]
```

if ifadesi iki bölümden oluşur: Doğruysa ve Yanlışsa bölümü. Önce parantez içerisindeki ifadenin değeri hesaplanır. Bu değer bool türünden olmak zorundadır. Eğer bu değer TRUE ise if parantezinden sonraki ifade FALSE ise else anahtar sözcüğünden sonraki ifade yapılır. Örneğin:

```
f <- function(a, b)
{
  if (a > b)
    a
  else
    b
}
> f(12, 34)
[1] 34
> f(100, 5)
[1] 100
```

Eğer if ifadesinin doğruysa ya da yanlışsa kısmında birden fazla ifade bulunacaksa bloklama yapılmalıdır. Örneğin:

```
solve <- function(a, b, c)
{
  delta <- b * b - 4 * a * c
  if (delta < 0)
    "Kök yok"
  else {
    x1 <- (-b + sqrt(delta)) / (2 * a)
    x2 <- (-b - sqrt(delta)) / (2 * a)
    paste("x1 = ", x1, ", x2 = ", x2, sep = "")
  }
}
```

Örneğin:

```
> solve(1, 0, -4)
[1] "x1 = 2, x2 = -2"
```

Örneğin:

```
> solve(2, 3, 5)
[1] "Kök yok"
```

Burada solve fonksiyonu ikinci derece denklemin katsayılarını alıp onun sonucunu bir yazı olarak geri döndürmektedir.

if ifadesi aslında bir fonksiyonun statüsündedir ve bir değer de geri döndürmektedir. if ifadesinin geri döndürdüğü değer eğer parantez içerisindeki ifade TRUE ise doğruysa kısmındaki son ifadenin değeri FALSE ise yanlışsa kısmındaki son ifadenin değeridir. Örneğin:

```
m <- if (a > b) a else b
```

Burada if ifadesinden duruma göre a ya da b değeri elde edilmiştir. Bunu bir fonksiyon olarak yazmak isteyelim:

```
max <- function(a, b)
{
  m <- if (a > b) a else b
  m
}

> v <- max(10, 20)
> v
[1] 20
```

Burada max fonksiyonu aşağıdaki gibi daha kompakt da yazılabilirdi:

```
max <- function(a, b)
{
  if (a > b) a else b
}
```

Burada max fonksiyonunun geri dönüş değeri son ifade olan if ifadesinin geri dönüş değeridir. if de duruma göre a ya da b'ye geri dönmektedir.

if ifadesinin else kısmı olmayabilir. Örneğin:

```
if (a > 0)
  print("pozitif")
print("devam")
```

Burada if ifadesinin else kısmı yoktur. Dolayısıyla print("devam") ifadesi if dışındadır.

Eğer if ifadesinin else kısmı yoksa ve koşul da sağlanmıyorsa bu durumda if ifadesinden NULL değeri elde edilir. Örneğin:

```
> a <- -10
> b <- if (a > 0) "pozitif"
> b
NULL
```

if parantezi içerisindeki bool vektör eğer birden fazla elemandan oluşuyorsa doğruluk yanlışlık kontrolüne yalnızca onun ilk elemanın değeri sokulmaktadır. Başka bir deyişle if ifadesindeki karşılaştırmadan elde edilen değerlerin yalnızca ilk elemanı anlamlıdır. Karşılaştırma vektörünün birden fazla elemandan oluşması durumunda R bize bir uyarı mesajı da verir. Örneğin:

```
foo <- function(a)
{
  if (a > 0)
    cat("pozitif")
  else
    cat("negatif")
}

> foo(c(3, -2, -1))
pozitifWarning message:
```

```
In if (a > 0) cat("pozitif") else cat("negatif") :  
  the condition has length > 1 and only the first element will be used
```

R'da tıpkı C ve C++'ta olduğu gibi bool türü ile diğer türler birbirlerine otomatik dönüştürülebilmektedir. bool türü ile diğer nümerik türleri işleme soktuğumuzda TRUE değeri 1 olarak FALSE değeri 0 olarak işleme girer. Örneğin:

```
> a <- T  
> b <- a + 2  
> b  
[1] 3
```

Benzer biçimde if ifadesinde koşul bool türden değilse sıfır dışı değerler TRUE gibi, 0 değeri FALSE gibi işlem görür. Örneğin:

```
> a <- 10  
> if (a) "Evet" else "Hayır"  
[1] "Evet"  
> a <- 0  
> if (a) "Evet" else "Hayır"  
[1] "Hayır"  
> a <- -10  
> if (a) "Evet" else "Hayır"  
[1] "Evet"
```

İç içe if ifadeleri söz konusu olabilir.

```
max3 <- function(a, b, c)  
{  
  if (a > b) {  
    if (a > c)  
      a  
    else  
      c  
  }  
  else {  
    if (b > c)  
      b  
    else  
      c  
  }  
}
```

R'da atama işlemi de bir fonksiyon olarak işlev görmektedir. Yani atama işleminin de bir geri dönüş değeri vardır. Atama işleminin geri dönüş değeri atanan değerdir. Örneğin:

```
a <- 10
```

Bu işlem den ürün olarak 10 değeri elde edilir. Yani 10 değeri hem a 'ya atanmıştır hem de ürün olarak elde edilmiştir. Böylece biz bir dizi atamayı peş peşe şöyle yapabiliriz:

```
> a <- b <- 100  
> b  
[1] 100  
> a  
[1] 100
```

Örneğin:

```
avg <- function(a, b)  
{
```

```
temp <- (a + b) / 2
temp
}
```

Bu fonksiyonun eşdeğeri aşağıdaki gibidir:

```
avg <- function(a, b)
{
  temp <- (a + b) / 2
}
```

Bunun da eşdeğeri aşağıdaki gibidir:

```
avg <- function(a, b)
{
  (a + b) / 2
}
```

R'da Döngüler

Bir prpogram parçasının yinelenmeli olarak çalıştırmasını sağlayan program yapılarında döngü (loop) denilmektedir. R'da üç temel döngü vardır: for döngüleri, while döngüleri ve repeat döngüleri.

R'da for Döngüleri

R'da yineme işlemleri için ağırlıklı olarak for döngüleri kullanılmaktadır. R'ın for döngüsü pek çok dildeki foreach döngüsü gibidir. R'da for döngüleri bir vektör alır vektör elemanı için bir ifadeyi çalıştırır. Genel biçimi şöyledir:

for (<değişken> in <vektör>) ifade

for döngüsü şöyle çalışmaktadır: Döngü her yinelenmede vektörün sıradaki elemanını değişkene atar ve döngü ifadesini çalıştırır. Vektör elemanları bittiğinde yineme de bitmiş olur. Örneğin:

```
mysum <- function(a)
{
  total <- 0
  for (x in a)
    total <- total + x
  total
}
```

Bu örneğimizde geri dönüş değerini açıkça oluşturmak için ayrı bir satıra total ifadesini ekledik. Eğer bunu yapmasaydık mysum bize NULL değeri verirdi:

```
mysum <- function(a)
{
  total <- 0
  for (x in a)
    total <- total + x
}
```

Çünkü for döngüsü de aslında R'da bir fonksiyon belirtmektedir. Bu fonksiyonun da geri dönüş değeri NULL'dır. Burada fonksiyonda son yapılan ifade total <- total + x değildir. for ifadesidir. Bu ifade for ifadesinin içerisinde. Yani for ifadesinin bir parçasını oluşturmaktadır.

```
> mysum(1:100)
[1] 5050
```

Örneğin:

```
> mysum(1:100)
[1] 5050
```

Şimdi de sum ve mean fonksiyonlarını kullanmadan ortalama hesaplayan mymean isimli fonksiyonu yazmak isteyelim:

```
mymean <- function(v)
{
  total <- 0

  for (i in v)
    total <- total + i

  total / length(v)
}

> mymean(1:3)
[1] 2
> mymean(c(1, 1, 1, 1, 2))
[1] 1.2
```

Örneğin:

```
foo <- function(names)
{
  for (str in names)
    print(str)
}

> foo(c("ali", "veli", "selami"))
[1] "ali"
[1] "veli"
[1] "selami"
```

Bir sayının asal sayı olup olmadığını veren is.prime isimli fonksiyonu yazmaya çalışalım:

```
is.prime <- function(a)
{
  for (x in 2:(a-1))
    if (a %% x == 0)
      return(FALSE)

  return(TRUE)
}
```

Burada 2'den söz konusu sayıya kadar her sayının ilgili sayıyı tam bölüp bölmediğine bakılmıştır. return fonksiyonunun tüm fonksiyonu sonlandırdığına dikkat ediniz. Akış ilgili değeri hiçbir sayı tam bölmediğinden dolayı döngüden çıkar. Bu duruma da fonksiyon TRUE ile geri döndürülmüştür. Burada for içerisindeki ifadenin if olduğuna dikkat ediniz. if toplamda tek bir ifade olduğu için bloklama yapmaya gerek yoktur. Yani kod aşağıdakiyle eşdeğerdir:

```
is.prime <- function(a)
{
  if (x == 2)
    return(TRUE);

  for (x in 2:(a-1))
  {
    if (a %% x == 0)
      return(FALSE)
  }
}
```



```

    return(TRUE)
}

```

Öklit teoremine göre asal olmayan sayıların kareköküne kadar bir asal çarpanları vardır. Bu durumda bölme denemesini sayının kareköküne kadar yapmak yeterlidir. Ayrıca çift sayıların her defasında kontrol edilmesine de gerek yoktur. O halde fonksiyon aşağıdaki gibi daha verimli biçimde düzenlenebilir:

```

is.prime <- function(a)
{
  if (a %% 2 == 0)
    return(a == 2)

  for (x in seq(3, sqrt(a), 2))
    if (a %% x == 0)
      return(FALSE)

  return(TRUE)
}

```

Bir vektörün elemanlarını tek tek aşağıdaki gibi print edebiliriz:

```

print.vector <- function(a)
{
  for (i in 1:length(a))
    print(a[i])
}

```

```

> print.vector(1:4)
[1] 1
[1] 2
[1] 3
[1] 4

```

Fonksiyonu şöyle de yazabilirdik:

```

print.vector <- function(v)
{
  for (i in v)
    print(i)
}

```

for döngüsü de R'da aslında bir fonksiyon gibidir. for döngüsü bize her zaman NULL değerini verir.

R'daki for döngülerini diğer dillerdeki for döngüleri gibi kullanabiliriz. Örneğin n defa yineleme sağlamak için for döngüsü R'da şöyle kullanılabilir:

```

for (i in 1:n) {
  ...
}

```

listeler de birer vektör belirttiğine göre biz for döngüsünü listelerle de kullanabiliriz. Bu durumda döngü her yinelenişinde listenin sıradaki elemanı döngü değişkenine çekilecektir. Örneğin:

```

l <- list(a = 10, b = 20)
for (i in l)
  print(i)

```

Burada listenin elemanları olan 10 ve 20 ekrana yazdırılacaktır. Örneğin:

```
myprint <- function(v)
{
  for (x in v)
    print(x)
}

> l <- list(a = 100, b = 200, c = 300)
> myprint(l)
[1] 100
[1] 200
[1] 300
```

Şimdi de myprint fonksiyonunu şöyle çağıralım:

```
> l <- list(a = c(10, 20, 30), b = c("ali", "veli", "selami"))
> myprint(l)
[1] 10 20 30
[1] "ali" "veli" "selami"
```

Ne Zaman Bloklama Gerekir?

if gibi for gibi ifadelerde eğer işletilecek alt ifadelerin sayısı birden fazla ise bloklama yapmak zorunludur. Örneğin:

```
if (a > 10) {
  x <- x + 1
  y <- y + 1
}
```

if ve for ifadelerinin kendileri dışarıdan bakıldığında bütünsel olarak tek bir ifadedir. Bu nedenle örneğin for döngüsünün içerisinde bir if varsa if ifadesinin kendisi tek bir ifade olduğu için bloklama yapmaya gerek yoktur. Örneğin:

```
evensum <- function(v)
{
  total <- 0

  for (i in v)
    if (i %% 2 == 0)
      total <- total + i

  total
}
```

Burada for içerisinde if vardır. if ifadesinin hepsi tek bir ifadedir. Dolayısıyla bloklamaya gerek yoktur. Tabii tek ifade için de bloklama yapmanın bir sakıncası yoktur.

Örneğin üç sayısının 2 büyüğü ile geri dönen max3 fonksiyonunda aslında hiç bloklama yapmaya gerek yoktur.

```
max3 <- function(a, b, c)
{
  if (a > b)
    if (a > c)
      a
    else
      c
  else
    if (b > c)
      b
    else
      c
}
```

```
c
}
```

Sınıf Çalışması: rev fonksiyonuu myrev ismiyle yeniden yazınız. Yani myrev fonksiyonu bizden bir vektörü parametre olarak alıp ters sırada elemanlardan oluşan yeni bir vektör verecektir.

```
myrev <- function(v)
{
    #....
}
```

Çözüm:

```
myrev <- function(v)
{
    k <- integer()

    for (i in length(v):1)
        k[length(v) - i + 1] <- v[i]
    k
}
```

Yukarıdaki örnekte integer() çağrısı bize 0 elemanı boş bir integer vektör vermektedir.

break ifadesi

Bir döngüyü belli bir aşamada break ifadesi ile sonlandırabiliriz. Programın akışı break ifadesini gördüğünde hangi döngünün içerisinde bulunuluyorsa döngü sonlandırılır. Örneğin:

```
foo <- function(a)
{
    total <- 0
    for (x in a) {
        if (x == 0)
            break
        total <- total + x
    }
    total
}
```

Burada bir vektör içerisinde 0 görülene kadarki değerler toplanmıştır. Sıfır görüldüğünde akış break ifadesine gelir ve break de döngünün sonlanmasına yol açar.

```
foo <- function()
{
    for (x in 1:10) {
        val <- scan("")
        if (val == 0)
            break
        print(val * val)
    }
}
```

Burda foo fonksiyonunda 10 kere yinelenen bir döngü vardır. Her defasında kalvyeden scan fonksiyonuyla bir değer okunmuştur ve bu değerın karesi yazdırılmıştır. Ancak 0 girildiğinde döngüden break ile çıkmıştır.

repeat Döngüsü

Bazı durumlarda döngü açık bir koşul olmayabilir. Duruma göre programcı içeride bazı koşullar sağlandığında döngüden çıkmak isteyebilir. İşte diğer program dillerinde böylesi sonsuz döngüler for döngüleri ile yapılabilmektedir. Halbuki R'da for döngüleri sonsuz döngü oluşturmak için kullanılamamaktadır. İşte R'da sonsuz döngüler için repeat isimli bir döngü düşünülmüştür. repeat döngüsü bir koşul içermez. Sürekli yineleme sağlar. Programcı da döngüden break ile çıkar. repeat döngüsünün genel biçimi şöyledir:

repeat ifade

Yine repeat içerisinde birden fazla ifade varsa bloklama yapılmalıdır. Örneğin:

```
foo <- function()
{
  x <- 0
  repeat {
    print(x)
    x <- x + 1
    if (x == 10)
      break
  }
}
```

```
> foo()
[1] 0
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
```

while Döngüsü

while döngüleri pek çok dilde benzer biçimde bulunmaktadır. Genel olarak tüm dillerde while döngüleri "bir koşul sağlandığı sürece yinelemeye" yol açmaktadır. while döngülerinin R'daki genel biçimleri şöyledir:

while (koşul) ifade

R'da while döngüleri şöyle çalışır: Her yinelemede while parantezinin içerisindeki ifadenin değeri hesaplanır. Bu değer TRUE ise ifade yapılır ve başa dönülür. FALSE ise while döngüsünden çıkarılır.

Yine birden fazla ifade döngü içerisinde bulundurulacaksa bloklama yapılmalıdır. Örneğin:

```
foo <- function()
{
  i <- 0
  while (i < 10) {
    print(i)
    i <- i + 1
  }
}
```

Örneğin:

```
foo <- function(v)
{
  i <- 1
```

```

while (v[i] != 0) {
  print(i)
  i <- i + 1
}
}

> foo(c(1:5, 0, 1:5))
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5

```

Fonksiyonların Birden Fazla Değer Geri Döndürmesi Durumu

Normal olarak bir fonksiyon tek bir değer geri döndürmektedir. Ancak bazı durumlarda fonksiyonun birden fazla değer döndürmesini isteyebiliriz. Bu durumlarda fonksiyon kendi içerisinde bir liste oluşturur ve listeye geri döner. Fonksiyonu çağıran kişi de değeri liste içerisinde alır. Örneğin:

```

split.numbers <- function(a)
{
  evens <- a[a %% 2 == 0]
  odds <- a[a %% 2 == 1]

  list(evens = evens, odds = odds)
}

> l <- split.numbers(1:100)

> l$odds
[1] 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41
[22] 43 45 47 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83
[43] 85 87 89 91 93 95 97 99
> l$evens
[1] 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32
[17] 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62 64
[33] 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94 96
[49] 98 100

```

Örneğin ikinci derece bir denklemin köklerini bir listeye yerleştirip onu bir liste olarak vermeye çalışalım:

```

getroots <- function(a, b, c)
{
  delta <- b * b - 4 * a * c
  if (delta < 0)
    return(NULL)

  x1 <- (-b + sqrt(delta)) / (2 * a)
  x2 <- (-b - sqrt(delta)) / (2 * a)

  list(x1 = x1, x2 = x2)
}

> l <- getroots(1, 0, -4)
> l
$x1
[1] 2

$x2
[1] -2

```

Sınıf Çalışması: sorted_list isimli fonksiyonu şöyle yazınız: Fonksiyon bir atomik bir vektörü parametre olarak almaktadır. Geri dönüş değeri olarak bir liste vermektedir. Listenin iki elemanı vardır. ascending elemanı verilen vektörün küçükten büyüğe sıraya dizilmiş halini, descending elemanı ise büyükten küçüğe sıraya dizilmiş halini vermektedir. Fonksiyonu yazdıktan sonra test ediniz. Sıraya dizme işlemi için taban kütüphanedeki sort fonksiyonunu kullanabilirsiniz.

Çözüm:

```
sorted_list <- function(v)
  list(ascending = sort(v), descending = sort(v, decreasing = T))

> l <- sorted_list(c(4, 7, 2, 5, 21))
> l
$ascending
[1] 2 4 5 7 21

$descending
[1] 21 7 5 4 2
```

Sınıf Çalışması: Bir vektörü parametre olarak alıp asal olanları ve asal olmayanları ayrıştırarak bir liste biçiminde geri döndüren split.prime isimli fonksiyonu is.prime fonksiyonunu kullanarak yazınız. Listenin primes elemanı asal olan vektörü nonprimes elemanı asal olmayan vektörü vermelidir.

Çözüm:

```
is.prime <- function(a)
{
  if (a == 2)
    return (TRUE);

  for (x in 2:(a-1))
  {
    if (a %% x == 0)
      return(FALSE)
  }

  return(TRUE)
}

split.prime <- function(v)
{
  a <- integer()
  b <- integer()
  acount <- 1
  bcount <- 1

  for (i in v) {
    if (is.prime(i)) {
      a[acount] <- i
      acount <- acount + 1
    }
    else {
      b[bcount] <- i
      bcount <- bcount + 1
    }
  }

  list(primes = a, nonprimes = b)
}
```

Fonksiyon Parametrelerinin Default Değer Alması Durumu

Önceki konularda da belirtildiği gibi fonksiyon parametreleri default değer alabilmektedir. Bu durumda çağrı sırasında o parametreye ilişkin argüman girilmezse o default değer girilmiş gibi işlem görnmektedir. Örneğin:

```
foo <- function(a, b = 100)
{
  cat("a = ", a, ", b = ", b)
}

> foo(200)
a = 200 , b = 100
> foo(200, 400)
a = 200 , b = 400
```

Örneğin:

```
myprint <- function(v, sep = ",")
{
  cat(v, sep = sep)
  cat("\n")
}

> myprint(1:10)
1,2,3,4,5,6,7,8,9,10
> myprint(1:10, sep = " ")
1 2 3 4 5 6 7 8 9 10
```

Bazen parametre değişkenine verilen default değerler gerçek gereksinim duyulan değerler değildir. Bu değerler çağrının default argüman ile yapıldığını belirlemek için kullanılıyor olabilir. Örneğin:

```
disp.date <- function(day = 0, month = 0, year = 0)
{
  curdate <- Sys.Date()

  if (day == 0)
    day <- as.integer(substr(curdate, 9, 10))
  if (month == 0)
    month <- as.integer(substr(curdate, 6, 7))
  if (year == 0)
    year <- as.integer(substr(curdate, 1, 4))

  str <- sprintf("%02d/%02d/%04d", day, month, year)

  cat(str)
}

> disp.date(10)
10/01/2018
> disp.date(10, 12)
10/12/2018
> disp.date()
04/01/2018
```

Okunabilirlik için default değerlerin çok kullanılan değerlerden seçilmesi uygundur.

İçe İçe Fonksiyon Bildirimleri

R'da iç içe fonksiyonlar bildirilebilir. Yani bir fonksiyon içerisinde biz başka bir fonksiyonu oluşturabiliriz. Daha sonra onu çağırabiliriz. Örneğin:

```
foo <- function()
{
  bar <- function()
    print("I am bar")

  bar()
}
> foo()
[1] "I am bar"
```

Örneğin:

```
foo <- function(a)
{
  bar <- function(b)
  {
    b * b
  }

  bar(a + 1)
}

> foo(10)
[1] 121
```

R'da Değişkenlerin Yaratılması ve Yok Edilmesi

R'da bir değişkene ilk kez değer atandığında değişken de yaratılmış olur. Eğer ilgili değişken bir fonksiyon içerisinde yaratılmışsa fonksiyonun çalışması bittikten sonra otomatik olarak yok edilir. Değişken global düzeyde (yani hiçbir fonksiyonun içerisinde olmadan en dışarda) yaratılmışsa kalmaya devam eder. Ancak programcı isterse bir değişkeni belli bir noktada yok edebilir. Bu işlem `rm` fonksiyonuyla yapılmaktadır. `rm` fonksiyonunun parametrik yapısı şöyledir:

```
rm(..., list = character(), pos = -1, envir = as.environment(pos), inherits = FALSE)
```

Örneğin:

```
> x <- 100
> x
[1] 100
> rm(x)
> x
Error: object 'x' not found
```

Normal olarak `rm` fonksiyonunda silinecek değişkenin ismi verilir. Tabii bu isim tırnak içerisinde alınmaz. Eğer isim bir string vektör biçimindeyse biz bu isimdeki değişkenleri silmek için `list` parametresini kullanırız. Örneğin:

```
> x <- 10
> rm(list = "x")
> x
Error: object 'x' not found
```

`ls` fonksiyonu tüm değişkenlerin listesini bize bir string vektör olarak verir. O halde R'da çalışma alanındaki tüm değişkenleri silmek için şu çağırışı kullanabiliriz:

```
rm(list = ls())
```

Değişkenlerin Faaliyet Alanları (Scope)

Bir değişkenin kullanılabildiği program aralığına faaliyet alanı (scope) denilmektedir. R'daki faaliyet alanı kuralı C/C++ gibi dillere benzemektedir. R'da fonksiyonun ayrı bir faaliyet alanı vardır. İç fonksiyonlar dış fonksiyonların değişkenlerini kullanabilirler. Fakat dış fonksiyonlar iç fonksiyonların değişkenlerini kullanamazlar. Ayrık fonksiyonlar da birbirlerinin değişkenlerini kullanamazlar. Örneğin:

```
foo <- function()
{
  a <- 10

  bar <- function()
  {
    b <- a + 10
    print(b)
  }

  x <- b + 1 # error
  bar()
}
```

Burada foo'nun başındaki a foo'nun içerisinde yaratılmıştır. İçteki fonksiyon dıştaki değişkenlerini kullanabilir. Dolayısıyla bar içerisinde foo'daki a'nın kullanımı tamamen geçerlidir. Ancak foo'nun içerisinde b'nin kullanımı geçerli değildir. Dış fonksiyon iç fonksiyonun değişkenlerini kullanamaz.

R'da bir fonksiyon çağrıldığı zaman fonksiyonun iç kodunun geçerliliği kontrol edilmektedir. Buna "geç değerlendirme (lazy evaluation)" denilmektedir. Geç değerlendirme konusu ileride yeniden ele alınacaktır. Geç değerlendirme fonksiyon çağrıldığında yani akış o noktaya geldiğinde yapılan değerlendirmedir. Örneğin:

```
foo <- function()
{
  bar <- function()
  {
    b <- a + 10          # akış bu noktaya geldiğinde a varsa sorun oluşmaz
    print(b)
  }

  a <- 10

  bar()
}
```

Burada bar fonksiyonunun içerisinde a kullanılmıştır. Ancak henüz a yaratılmamıştır. Eğer bar çağrıldığı sırada a yaratılmışsa bu soruna yol açmaz. Dolayısıyla yukarıdaki kodda biz foo fonksiyonunu çağırdığımızda ondan önce a yaratılmış olacağı için herhangi bir error oluşmayacaktır.

İç içe olmayan farklı iki fonksiyon birbirlerinin içerisinde yaratılmış değişkenleri kullanamaz. Örneğin:

```
foo <- function()
{
  x <- 10
}

bar <- function()
{
  y <- x + 10
}
```

Burada bar'ın içerisinde x değişkeni kullanılamaz. Önce foo sonra bar çağrılrsa bile bar fonksiyonu foo'nun içerisindeki x'i kullanamayacaktır. Örneğin:

```
> foo()
> bar()
Error in bar() : object 'x' not found
```

Hiçbir fonksiyonun içerisinde olmayan alanda yaratılan değişkenlere global değişkenler denir. Bunlar hiyerarşi olarak en tepede olduğu için her fonksiyondan kullanılabilirler. Örneğin:

```
x <- 100

foo <- function()
{
  y <- x + 10
  print(y)
}
```

Burada foo çağrıldığında x değişkeni global alanda yaratılmış olacağı için onun kullanımı soruna yol açmaz. O halde bir fonksiyon içerisinde henüz yaratılmamış bir değişken kullanılıyor olsun. Biz o fonksiyonu çağırmadan önce o değişkeni global alanda yaratırsak artık bir sorun oluşmaz.

Fonksiyonların Yan Etkiye (Side Effect) Sahip Olmaması Durumu

R'ın en önemli özelliklerinden biri saf fonksiyonel (pure functional) dillerin hemen hepsinde var olan yan etki (side effect) özelliğidir. Yan etki bir fonksiyonun kendi dışındaki değişkenler üzerinde bir değişiklik yapması durumudur. R'da fonksiyonların yan etkileri yoktur. Yani R'da biz bir fonksiyonu çağırdığımızda o fonksiyon kendisi dışında bir durum değişikliğine yol açmaz. R'da bir fonksiyon dış bir fonksiyonun değişkenini ya da global bir değişkeni kullanabilir ancak onu değiştirmek isterse aslında onu değil o değişkenin o fonksiyona özgü bir kopyasını değiştirmiş olmaktadır. Örneğin:

```
x <- 100

foo <- function()
{
  print(x)
  x <- 200
  print(x)
}

foo()
print(x)
```

Bu kodu çalıştırdığımızda foo fonksiyonu içerisindeki x global olan x'tir. Ancak foo fonksiyonu bu x'i değiştirdiğinde artık global olan x'i değiştirmez. Onun bu fonksiyona özgü bir kopyası oluşturulur ve fonksiyon onu değiştirmiş olur. Böylece yukarıdaki kod çalıştığında çıktı aşağıdaki gibi olacaktır:

```
[1] 100
[1] 200
[1] 100
```

Bu durum şöyle de düşünülebilir: Bir fonksiyon çağrıldığında o fonksiyonun kullanabileceği değişkenlerin o noktada o fonksiyona özgü kopyaları oluşturulmaktadır. Artık fonksiyon o kopyaları kullanıyor durumdadır. Her iki düşünce biçimi de aynı duruma yol açar. Özetle biz bir fonksiyonda üst bir fonksiyonun değişkenlerini ya da global değişkenleri kullanabiliriz. Ancak onu değiştirdiğimizde aslında o üst fonksiyonun değişkenlerini değiştirmiş olmayız. Onun yaratılmış olan kopyasını değiştirmiş oluruz. Böylece bir fonksiyon çağrıldığında üst fonksiyondaki ya da global alandaki hiçbir değişkenin değeri değiştirilmemiş olacaktır. Örneğin:

```
x <- 10

foo <- function()
{
```

```

print(x) # 10
x <- 20
print(x) # 20
bar <- function()
{
  print(x) # 20
  x <- 50
  print(x) # 50
}
bar()
print(x) # 20
}
foo()
print(x) # 10

```

Kod çalıştırıldığında aşağıdaki gibi bir çıktı ortaya oluşacaktır:

```

[1] 10
[1] 20
[1] 20
[1] 50
[1] 20
[1] 10

```

R'da Parametre Aktarımında Değerle Çağırma (Call By Value)

R'da fonksiyonların yan etkiye yol açmaması prensibi nedeniyle tüm argüman parametre aktarımları kopyalama yoluyla (call by value) yapılmaktadır. Adres yoluyla (call by reference) yapılmamaktadır. Dolayısıyla örneğin biz bir fonksiyona bir vektörü argüman olarak geçtiğimizde o vektörün elemanları fonksiyona tek tek kopyalanmakta ve artık fonksiyon o kopya üzerinde çalışmaktadır. Örneğin:

```

f <- function(a)
{
  for (i in 1:length(a))
    a[i] <- 0
}

v <- 1:10
print(v)
f(v)
print(v)

```

Burada ekran çıktısı şöyle olacaktır:

```

[1] 1 2 3 4 5 6 7 8 9 10
[1] 1 2 3 4 5 6 7 8 9 10

```

Sonuç olarak R'da bir fonksiyon parametre yoluyla da başka bir fonksiyonun değişkenlerini değiştiremez. Bunu ancak geri dönüş değeri yoluyla yapabilir. Bu zamana kadar görmüş olduğumuz fonksiyonların da aslında böyle çalıştığına dikkat ediniz. Örneğin sort fonksiyonu parametresiyle aldığı vektörü sort etmemektedir (edememektedir) bize sort edilmiş yeni bir vektör vermektedir. Örneğin:

```

> a <- c(3, 5, 2, 1, 9)
> b <- sort(a)
> a
[1] 3 5 2 1 9
> b
[1] 1 2 3 5 9

```

Örneğin R'da biz "parametresi ile aldığı vektöre eleman ekleyen bir fonksiyon" yazamayız:

```
f <- function(v, x)
{
  v[length(v)] <- x
}

a <- 1:10
f(a, 100)
print(a)
```

Burada f fonksiyonu eklemeyi parametre değişkeni olan a'ya yapmaktadır. v'ye yapmamaktadır. Bu işlemi şöyle yapabiliriz:

```
f <- function(v, x)
{
  v[length(v)] <- x
  v
}

a <- 1:10
a <- f(a, 100)
print(a)
```

Bizim R'da genel olarak bir fonksiyondan argüman olarak verdiğimiz değeri değıştirme yönünde beklentimizin olmaması gerekir. Çünkü bunun bir yolu yoktur.

R'da Fonksiyonların Geç Değerlendirilmesi

Pek çok prosedürel ve nesne yönelimli dilde fonksiyonlar bildirildikleri noktada değerlendirilirler. R'da ise fonksiyonlar oluşturuldukları noktada değil çağırılma sırasında değerlendirilmektedir. Örneğin:

```
foo <- function(a)
{
  a + b
}
```

Burada henüz b değişkeni oluşturulmamış olsa bile fonksiyonun bu biçimde bildirilmesi error oluşturmaz. B'nin olup olmadığına foo fonksiyonu çağırıldığında bakılacaktır. Örneğin:

```
> foo(10)
Error in foo(10) : object 'b' not found
```

Şimdi b'yi oluşturup yeniden foo fonksiyonunu çağıralım:

```
> b <- 20
> foo(10)
[1] 30
```

Benzer biçimde bir fonksiyonu bildirirken onun içerisinde başka bir fonksiyonu çağırdığımızda bu fonksiyonun bildirim aşamasında bulunuyor olması gerekmez. Örneğin:

```
foo <- function(a)
{
  bar(a)
}
```

Burada bar fonksiyonu olmasa bile foo fonksiyonunun bildiriminde hata ortaya çıkmayacaktır. Tabii eğer bar fonksiyonu olmadan foo fonksiyonunu çağırırsak aşağıdaki gibi error ile karşılaşırız:

```
> foo(10)
Error in bar(a) : could not find function "bar"
```

R'da Öznitelikler (Attributes)

Öznitelikler pek çok programlama dilinde bulunmaktadır. Pek çok dil zamanla özniteliklendirme özelliğine de sahip olmuştur. Öznitelik bir nesneye onun değerinin dışında yerleştirilen birtakım bilgilere denilmektedir. R'da bir nesneye onun değeri dışında istediğimiz gibi bilgi yerleştirebiliriz. Özniteliklerin isimleri ve değerleri vardır. Bu isim değerleri istenildiği gibi verilebilir.

Bunun için attr ve attributes fonksiyonları kullanılmaktadır. attr fonksiyonunun hem normal hem de yer değiştirmeli (replacement) biçimleri vardır.

```
attr(x, which, exact = FALSE)
attr(x, which) <- value
```

Bir değişkene öznitelik atamak için tek yapılacak şey değişkenin ve özniteliğin ismini belirterek attr fonksiyonu ile atama yapmaktır. Yine attr fonksiyonu ile nesnenin belli bir özniteliğinin değerini elde edebiliriz.

```
> a <- 100
> attr(a, "name") <- "ali"
> attr(a, "name")
[1] "ali"
```

Burada biz a değişkenine ismi "name" olan değeri de "ali" olan bir öznitelik yerleştirdik. Şimdi başka bir öznitelik daha yerleştirelim:

```
> attr(a, "number") <- 12345
> attr(a, "number")
[1] 12345
```

Burada da biz a değişkenine ismi "number" olan değeri de 12345 olan bir öznitelik iliştiirmiş olduk.

Bir değişkene iliştiirilmiş olan tüm öznitelikleri biz attributes isimli fonksiyonla bir liste olarak elde edebiliriz. Örneğin:

```
> l <- attributes(a)
> l
$name
[1] "ali"

$number
[1] 12345

> l$name
[1] "ali"
> l$number
[1] 12345
```

Aslında attributes isimli fonksiyonun yer değiştirmeli (replacement) bir biçimi de vardır. Dolayısıyla biz bir liste olarak tek hamlede bir değişkene birden fazla öznitelik iliştiirebiliriz. Örneğin:

```
> a <- 100
> attributes(a) <- list(name = "ali", number = 12345)
> attributes(a)
$name
[1] "ali"

$number
[1] 12345
```

Değişkenlere iliştilen öznitelikler çeşitli amaçlarla çeşitli fonksiyonlar tarafından kullanılabilir. İleride bunun bazı kullanımları görülecektir. Örneğin aslında matris bir vektördür. Yalnızca ona "dim" isimli bir öznitelik iliştilmiştir:

```
> a <- 100
> attributes(a) <- list(name = "ali", number = 12345)
> attributes(a)
$name
[1] "ali"

$number
[1] 12345
```

Biz bu dim özniteliğini değiştirerek matrisin boyutlarını da değiştirebiliriz. Örneğin:

```
> a <- matrix(1:12, nrow = 4)
> a
      [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
> attr(a, "dim") <- c(2, 6)
> a
      [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    3    5    7    9   11
[2,]    2    4    6    8   10   12
```

Bir öznitelige NULL değeri atanırsa o öznitelik silinmiş olur. Örneğin:

```
> a <- matrix(1:12, nrow = 4)
> attr(a, "dim") <- NULL
> a
[1]  1  2  3  4  5  6  7  8  9 10 11 12
```

Örneğin aslında vektör elemanlarına isim verdiğimiz zaman bu isimler o vektör değişkenine "names" isimli bir öznitelik olarak iliştilmektedir. Örneğin:

```
> v <- c(ali = 100, veli = 200)
> attributes(v)
$names
[1] "ali" "veli"

> v <- c(ali = 100, 200, veli = 200)
> attributes(v)
$names
[1] "ali" ""      "veli"
```

Aynı durum aslında listeler için de geçerlidir. Liste elemanlarına verilen isimler o liste değişkenine iliştilmiş öznitelik olarak saklanmaktadır. Örneğin:

```
> l <- list(name = "kaan", 100, no = 12345)
> attributes(l)
$names
[1] "name" ""      "no"
```

R'da Her Şey Fonksiyondur

Aslında R'da operatör ya da deyim kavramı yoktur. +, -, *, / gibi operatörler aslında birer fonksiyondur. Biz bunları normal fonksiyon gibi de çağırabiliriz. Örneğin:

```
> 10 + 20
[1] 30
> +(10, 20)
[1] 30
> +(10, 20)
[1] 30
```

Eğer biz normal operatörleri fonksiyon sentaksı ile kullacaksak onları ters tırnaklamamız (backquote/backtick) gerekir. Türkçe klavyelerde ters tırnak işareti Alt Gr yardımı ile çıkartılmaktadır. Bu durumda örneğin:

```
> 10 + 20 * 2
[1] 50
```

ile aşağıdaki çağrı eşdeğerdir:

```
> +(10, *(20, 2))
[1] 50
```

Tabii +, -, *, / gibi operatörler iki operandlı olduğu için iki parametrelili fonksiyon gibi davranmaktadır. Bu durumda örneğin:

```
10 + 20 + 30
```

işleminin eşdeğeri fonksiyon çağırma sentaksıyla şöyledir:

```
+(+(10, 20), 20)
```

Mademki operatörler birer fonksiyondur. Bu durumda biz onları atama işlemine sokabiliriz:

```
> add <- +
> add(10, 20)
[1] 30
```

İki parametrelili bir fonksiyonu da biz iki operandlı bir operatör gibi kullanabiliriz. Ancak bunun için fonksiyonun `%<isim>%` biçiminde isimlendirilmiş olması gerekir. Örneğin:

```
> %xxx% <- function(a, b) a + b
> 10 %xxx% 20
[1] 30
```

Burada operatör biçiminde kullanımda ismin ters tek tırnak içerisinde alınmadığına dikkat ediniz. Ancak fonksiyon biçiminde kullanmak için isim yine tek ters tırnaklanmalıdır. Örneğin:

Örneğin rep fonksiyonunu %% biçiminde iki operandlı bir operatör gibi kullanmak isteyelim:

```
> %r% <- rep
> 3 %r% 4
[1] 3 3 3 3
> "ali" %r% 2
[1] "ali" "ali"
```

Örneğin:

```
> 1 %r% 2 %r% 3
[1] 1 1 1 1 1 1
```

Örneğin:

```
> `%e%` <- function(v, k) v[k]
> c(1, 2, 3, 4, 5) %e% 3
[1] 3
```

Diğer dillerdeki if gibi , for gibi kontrol deyimleri de aslında R'da birer fonksiyondur. Örneğin biz if deyimini fonksiyon gibi de kullanabiliriz:

```
> a <- 20
> `if` (a > 10, "evet", "hayır")
[1] "evet"
```

Yani aslında if ifadesi if fonksiyonunun özel bir kullanım sentaksı gibidir. if ifadesini fonksiyon gibi kullanırken fonksiyonun üç parametrelili olduğuna ve fonksiyon isminin yine tek tırnaklandığına dikkat ediniz. Eğer fonksiyon biçiminde kullanımda yine doğruysa ya da yanlışsa kısmında birden fazla ifade varsa bloklama yapılmalıdır. Örneğin:

```
> `if` (a > 10, { print("evet"); print("doğru") }, {print("hayır"); print("değil")})
[1] "evet"
[1] "doğru"
```

Örneğin R'da for döngüsü de aslında bir fonksiyondur. Onu da bir fonksiyon gibi kullanabiliriz.:

```
> v <- 1:5
> `for` (x, v, print(x))
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

Aslında bu çağrı aşağıdakiyle eşdeğerdir:

```
> for (x in v) print(x)
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

Yine döngünün içerisinde birden fazla ifade varsa üçüncü parametre blok içersine alınmalıdır. Örneğin:

```
> `for` (x, v, { print(x); print(x*x) })
[1] 1
[1] 1
[1] 2
[1] 4
[1] 3
[1] 9
[1] 4
[1] 16
[1] 5
[1] 25
```

Benzer biçimde while ve repeat döngüleri de aslında birer fonksiyon gibi kullanılabilirler. Örneğin:

```
> i <- 1
> `while` (i < 10, {print(i); i <- i + 1})
[1] 1
```



```
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
```

Örneğin:

```
`repeat`(print("ok"))
```

Fonksiyonların Listelere Yerleştirilmesi

Biz atomik bir vektör içerisine fonksiyonları yerleştiremeyiz. Başka bir deyişle fonksiyonlardan oluşan bir atomik vektör oluşturamayız. R'da birden fazla fonksiyonu bir arada tutmak için listeler kullanılmaktadır. Anımsanacağı gibi listeler de bir çeşit vektördür. Örneğin:

```
> l <- list()
> l[[1]] <- `+`
> l[[2]] <- `-`
> l[[3]] <- `*`
> l[[4]] <- `/`
> for (f in l) print(f(10, 20))
[1] 30
[1] -10
[1] 200
[1] 0.5
```

Tabii buradaki listeyi daha sade olarak söyle de yaratabilirdik:

```
> l <- list(`+`, `-`, `*`, `/`)
> for (f in l) print(f(10, 20))
[1] 30
[1] -10
[1] 200
[1] 0.5
```

Yer Değiştirme (Replacement) Fonksiyonları

Anımsanacağı gibi R'da fonksiyonların yan etkileri yoktur. Bundan dolayı bir fonksiyon parametresiyle aldığı bir değişken üzerinde değişiklik yapamaz. Örneğin:

```
foo(x)
```

Burada foo fonksiyonu x'in değerini değiştiremez. İşte fonksiyonun parametresinin değerini değiştirebilmesi için onun "yer değiştirme fonksiyonu (replacement function)" biçiminde yazılması gerekmektedir. Biz şimdiye kadar bazı yer değiştirme fonksiyonlarını kullandık. Örneğin attr fonksiyonunu kullanarak bir değişkenin öznitelik bilgilerini değiştirmiştik:

```
> x <- 100
> attr(x, "test") <- "Test value"
```

Yine daha önce görmüş olduğumuz bir vektörün elemanlarına isim veren "names" isimli fonksiyon da bir yer değiştirme fonksiyonuydu:

```
> v <- c(10, 20)
> names(v) <- c("ali", "veli")
```

```
> v
ali veli
10 20
```

Yer değiştirme fonksiyonları fonksiyona atama yapılmıyormuş gibi bir sentaks ile kullanılmaktadır. Biz yer değiştirme fonksiyonu olmayan fonksiyonları olmayan fonksiyonları bu biçimde kullanamayız. Örneğin:

```
> sum(1:3) <- 10
Error in sum(1:3) <- 10 :
  target of assignment expands to non-language object
```

Yer değiştirme fonksiyonlarının isimleri `*<isim><-`* biçiminde olmak zorundadır. Örneğin `*foo<-`* gibi, `*bar<-`* gibi. Yer değiştirme fonksiyonlarının isimleri ters tek tırnak içerisinde alınmalıdır. Örneğin:

```
`setitem<-` <- function(x, value)
{
  x <- value
}

> a <- 10
> setitem(a) <- 20
> a
[1] 20
```

Burada biz a'yı setitem fonksiyonuna parametre olarak gönderdik. setitem atadığımız değeri a'ya yerleştirdi. Şimdi bir vektöre eleman ekleyen benzer bir yer değiştirme fonksiyonunu yazalım:

```
`additem<-` <- function(x, value)
{
  x[length(x) + 1] <- value
  x
}

> v <- 1:3
> additem(v) <- 100
> v
[1] 1 2 3 100
```

Yer değiştirme fonksiyonu diye bir fonksiyon olmasaydı bizim fonksiyonun geri dönüş değerini yeniden aynı değişkene atamamız gerekirdi. Örneğin:

```
> v <- c(3, 8, 4, 1, 6)
> v
[1] 3 8 4 1 6
> sort(v)
[1] 1 3 4 6 8
> v
[1] 3 8 4 1 6
> v <- sort(v)
> v
[1] 1 3 4 6 8
```

Şimdi bir vektörün dim özneliğini değiştirerek onu bir matris haline getiren mydim isimli bir yer değiştirme fonksiyonu yazalım:

```
`mydim<-` <- function(x, value)
{
  attr(x, "dim") <- value
  x
}

> v <- 1:9
> mydim(v) <- c(3, 3)
```

foo bir yer değiştirme fonksiyonu olmak üzere,

```
foo(x) <- y
```

işleminin eşdeğeri şöyledir:

```
x <- `foo<-`(x = x, value = y)
```

Biz bir yer değiştirme fonksiyonu çağırıp ona atama yaptığımız zaman aslında R yorumlayıcısı bunu fonksiyonu çağırıp geri dönüş değerini o değişkene atama biçiminde ele almaktadır. Örneğin aşağıdaki gibi bir yer değiştirme fonksiyonumuz olsun:

```
`foo<-` <- function(x, value)
{
  x <- value
  x
}
```

Biz bu fonksiyonu şöyle kullanmış olalım:

```
foo(a) <- 100
```

Bu işlemi aslında R yorumlayıcısı aşağıdaki gibi ele alıp uygular:

```
a <- `foo<-`(x = a, value = 100)
```

Yer değiştirme fonksiyonlarında value isimli özel bir parametre vardır. Bu value parametresi fonksiyon çağırısına atanacak değeri temsil eder. Örneğin:

```
foo(a) <- 100
```

Burada value parametresine aktarılacak değer 100 değeridir. Aslında value isimli parametrenin son parametresi gerekmemektedir. İsmi value olması yeterlidir. Örneğin:

```
`foo<-` <- function(value, x)
{
  x <- value
  x
}
> a <- 10
> foo(a) <- 20
> a
```

Yer değiştirme fonksiyonları ikiden fazla parametreye de sahip olabilir. Örneğin foo bir yer değiştirme fonksiyonu olmak üzere:

```
foo(x, y, z) <- k
```

ifadesinin eşdeğeri şöyledir:

```
x <- `foo<-`(x = x, y, z, value = k)
```

Yer değiştirme fonksiyonunun geri dönüş değeri her zaman birinci parametreyle belirtilen değişkene atanmaktadır. Örneğin:

```
`myattr<-` <- function(x, attr, value)
{
  attr(x, attr) <- value
}
```

```

x
}
> v <- 1:9
> myattr(v, "dim") <- c(3, 3)
> v
      [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

```

Görüldüğü gibi aslında yer değiştirme fonksiyonları yalnızca bir gösterim kolaylığı sağlamaktadır. Aslında yapılan iş yine fonksiyonun geri dönüş değerinin yeniden aynı değişkene atanması işlemidir.

Bazen bir fonksiyonun hem normal hem de yer değiştirmeli biçimleri birlikte bulunabilmektedir. Örneğin attr isimli fonksiyonun ya da length fonksiyonunun hem normal hem de yer değiştirmeli biçimi vardır:

```

> m <- matrix(1:9, nrow = 3)
> attr(m, "dim")
[1] 3 3
> attr(m, "dim") <- c(1, 9)
> m
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
[1,]    1    2    3    4    5    6    7    8    9

```

Ya da örneğin:

```

> v <- 1:3
> length(v)
[1] 3
> length(v) <- 2
> v
[1] 1 2

```

Fonksiyonların normal ve yer değiştirmeli biçimleri birbirlerine karışmaz çünkü bunların isimleri farklıdır. Örneğin foo için normal olan fonksiyon "foo" ismindeyken yer değiştirmeli biçimi "foo<-" ismindedir. Örneğin:

```

myattr <- function(x, attr)
{
  attr(x, attr)
}

`myattr<-` <- function(x, attr, value)
{
  attr(x, attr) <- value
  x
}

> m <- matrix(1:9, nrow = 3)
> myattr(m, "dim")
[1] 3 3
> myattr(m, "dim") <- c(1, 9)

```

Son olarak aşağıda R'in temel paketlerindeki, çok kullanılan yer değiştirme fonksiyonlarının bir listesini verelim:

```

attr<-      :function (x, which, value)
attributes<- :function (obj, value)
body<-      :function (fun, envir = environment(fun), value)
class<-     :function (x, value)
colnames<-  :function (x, value)
dim<-       :function (x, value)
dimnames<-  :function (x, value)

```

```
environment<- :function (fun, value)
formals<- :function (fun, envir = environment(fun), value)
is.na<- :function (x, value)
length<- :function (x, value)
levels<- :function (x, value)
mode<- :function (x, value)
mostattributes<- :function (obj, value)
names<- :function (x, value)
rownames<- :function (x, value)
substr<- :function (x, start, stop, value)
substring<- :function (text, first, last = 1000000L, value)
```

R'da Tablolar (Tables)

Tablo (table) sıklık değerlerini veren bir yapıdır. Bir tablo tek bir vektörden oluşturulabileceği gibi birden fazla vektörden de oluşturulabilir. Tablolar table isimli fonksiyonla yaratılmaktadır. Örneğin:

```
> v <- c(1, 2, 2, 1, 1, 3)
> t <- table(v)
> t
v
1 2 3
3 2 1
```

Burada üst satır vektördeki değerleri alt satır ise onların sıklıklarını göstermektedir. Yani burada 1'den 3 tane, 2'den 2 tane ve 3'ten 1 tane vardır. Örneğin:

```
> v <- c("ali", "ali", "selami", "veli", "veli")
> t <- table(v)
> t
v
    ali selami   veli
      2      1      2
```

Burada "ali"den 2 tane, "selami"den 1 tane ve "veli"den 2 tane vardır. Tablolar birden fazla vektörden de oluşturulabilir. Bu durumda vektörler eşit sayıda elemandan oluşmalıdır. Bunun sonucunda sıralı sıklıkların değerleri elde edilir. Örneğin:

```
> v <- c("ali", "ali", "selami", "veli", "veli")
> k <- c(1, 2, 1, 1, 1)
> t <- table(v, k)
> t
      k
v      1 2
ali      1 1
selami  1 0
veli     2 0
```

Burada "ali, 1" diziliminden 1 tane, "ali, 2" diziliminden 1 tane, "veli, 1" diziliminden 2 tane vs. vardır.

Tablonun elemanlarına tek boyutlu ya da çok boyutlu vektör gibi erişilebilmektedir. Örneğin:

```
> t[1,1]
[1] 1
```

Örneğin:

```
> t[, 1]
    ali selami   veli
```

R'da Grafik Çizimleri

Verilerin grafiklerle görselleştirilmesi betimleyici (descriptive) istatistiğin en önemli kısımlarından birini oluşturmaktadır. R'da da bu amaçla standart temel paketlerinde ve diğer bazı paketlerde pek çok fonksiyon bulundurulmaktadır.

Grafikleri çizerken grafiğe konu olan değişkenlerin ölçek türleri önemlidir. Biz ölçek türlerini daha önce nominal (nominal), sıralı (ordinal), aralıklı (interval) ve oranlı (ratio) olmak üzere dörde ayırmıştık. Şimdi burada ölçekleri başka bir açıdan "nominal (yani kategorik)" olan ve "sayısal olan (nümerik olan)" biçiminde iki sınıfa ayıracağız. Sıralı ölçekler de nominal olabilirler (örneğin eğitim durumu nominal bir ölçekle ifade edilir. Fakat aynı zamanda sıralıdır)

Bu durumda tipi grafikler dört biçimde oluşmaktadır:

- 1) Tek bir nominal değişkenin (univariate) grafiği
- 2) İki nominal değişkenin (bivariate) grafiği
- 3) Tek nümerik değişkenin grafiği
- 4) İki nümerik değişkenin grafiği

Tek nominal değişkenin grafiği genellikle sıklık grafiği biçimindedir. Yani tek nominal değişkenin olduğu durumda grafiği çizenin amacı düzeylerde (levels) kaçar tane elemanın olduğunu göstermektir. İki nominal değişkenin grafikleri karşılaştırma grafiği biçiminde karşımıza çıkar. Tek nümerik değişkenin pek çok grafiği olsa da tipik olarak histogram buna örnek verilebilir. Histogram nümerik değişkenlerin sıklık grafiğidir. İki nümerik değişkenin grafiği tipik olarak matematikteki fonksiyon (genel olarak bağıntı) grafiği gibi karşımıza çıkmaktadır.

R'da grafik işlemleri için yaygın üç paket kullanılmaktadır. Birincisi temel R paketidir. Biz kursumuzda daha çok bu paketteki fonksiyonları inceleyeceğiz. İkincisi "lattice" isimli pakettir. Üçüncüsü de "ggplot2" paketidir. Bu paketlerin hepsi birbirine benzer olmakla birlikte farklı özelliklere sahiptir.

Çizim Fonksiyonları

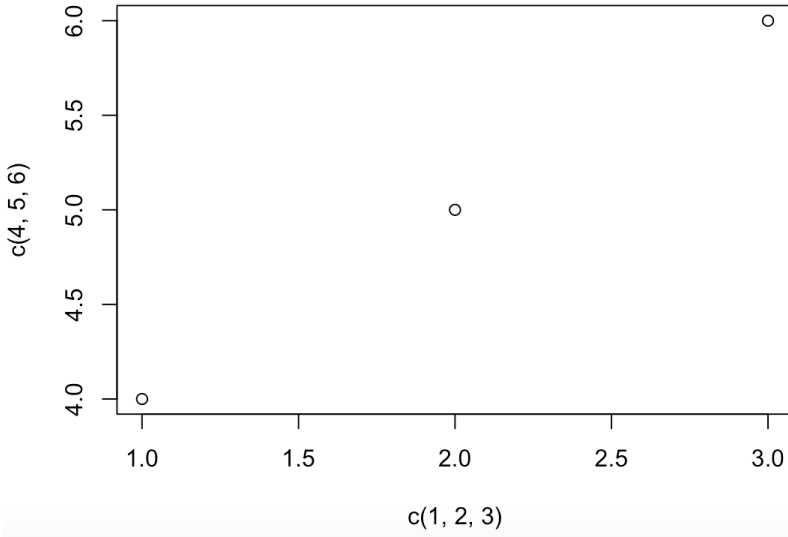
Grafik fonksiyonlarının en önemlilerinden biri plot fonksiyonudur. plot fonksiyonu hem tek değişkenin hem iki değişkenin nominal ve nümerik grafiklerini çizmek için kullanılır.

Bu fonksiyon x ve y vektörlerini alarak çizimi yapar. Fonksiyonun parametrik yapısı şöyledir:

```
plot(x, y, ...)
```

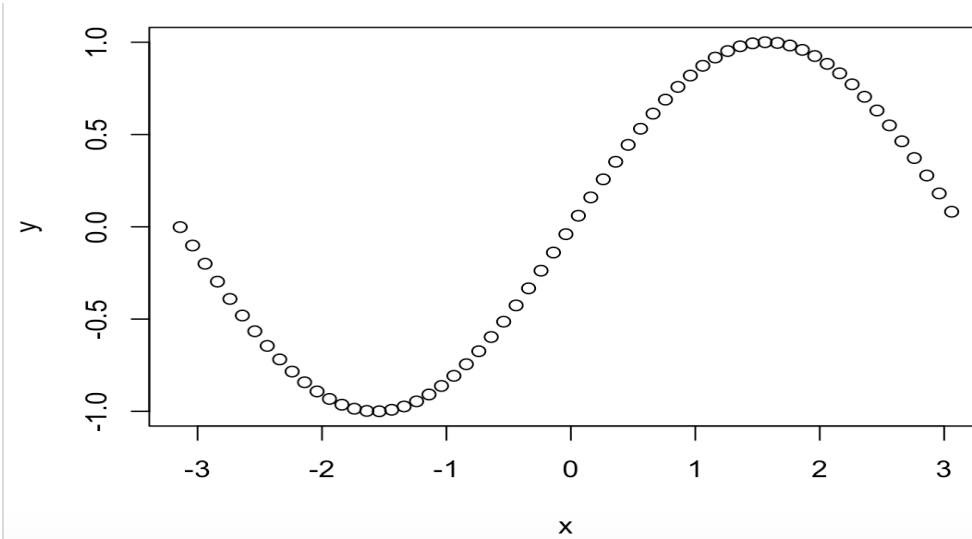
Örneğin iki nümerik vektörün noktasal grafiğini aşağıdaki gibi çizebiliriz:

```
plot(c(1, 2, 3), c(4, 5, 6))
```



Örneğin sinüs eğrisi de iki değişkenli nümerik bir grafik oluşturur (dördüncü durum):

```
> x <- seq(-3.14, +3.14, 0.1)
> y <- sin(x)
> y <- sin(x)
> plot(x, y)
```



plot fonksiyonunun pch parametresi noktaları belirtmek için hangi sembolün yerleştirileceğini belirlemekte kullanılır. pch aşağıdaki değerlerden birini alabilir:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
□	○	△	+	×	◇	▽	⊠	*	⊕	⊗	⊞	⊟	⊠	⊡	■	●	▲	◆	●	●	●	■	◆	▲	▼

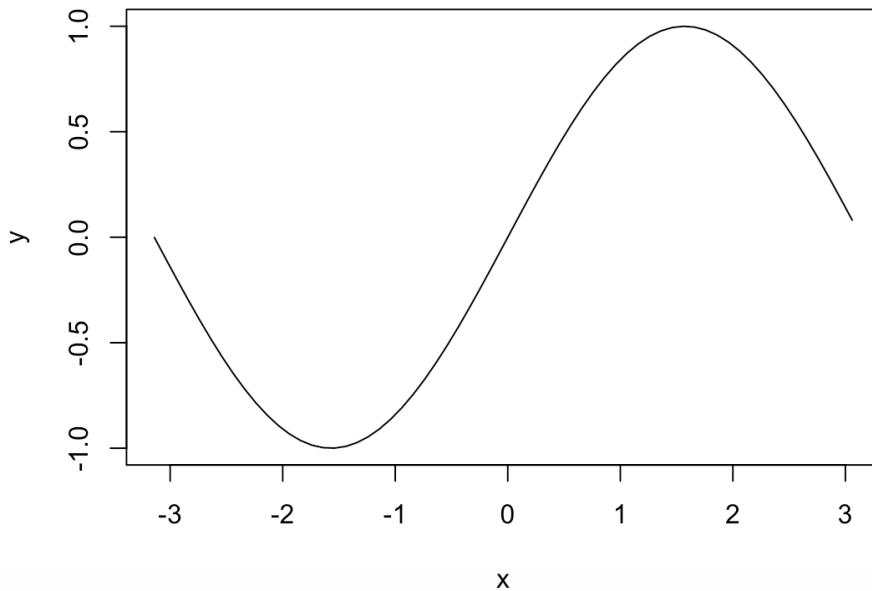
type parametresi çizimin tütünü belirtir ve şunlardan biri olabilir:

- "p" for points,
- "l" for lines,
- "b" for both,
- "c" for the lines part alone of "b",
- "o" for both 'overplotted',
- "h" for 'histogram' like (or 'high-density') vertical lines,
- "s" for stair steps,
- "S" for other steps, see 'Details' below,
- "n" for no plotting.

type parametresinin default değeri "p"dir. Tabii type parametresinde eğer nokta koyma özelliği yoksa bu durumda pch parametresinin bir anlamı kalmamaktadır. (Örneğin biz type parametresini "l" olarak girersek bu durumda pch değerinin bir önemi kalmaz.)

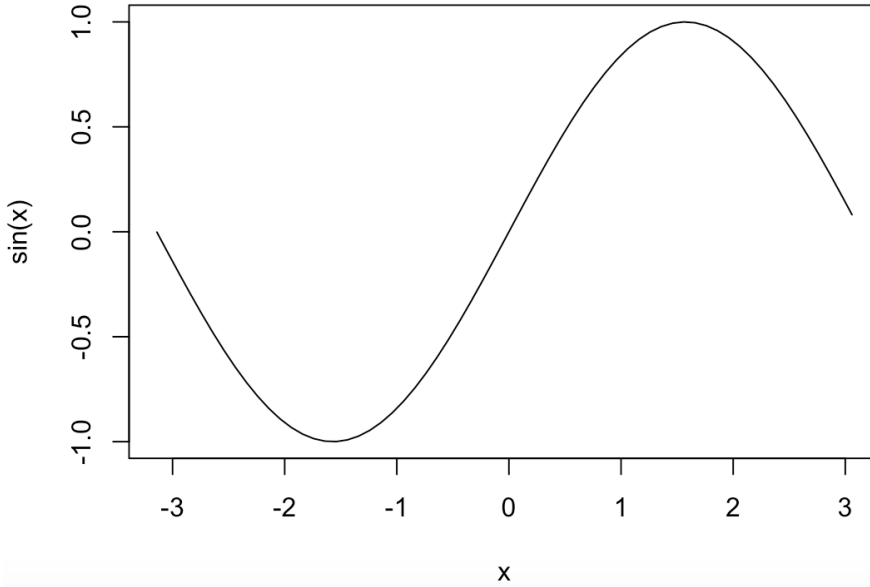
Örneğin:

```
> x <- seq(-3.14, +3.14, 0.1)
> y <- sin(x)
> plot(x, y, type = "l")
```



Fonksiyonun xlab ve ylab parametreleri eksenleri isimlendirmek için kullanılmaktadır. Örneğin:

```
> x <- seq(-3.14, +3.14, 0.1)
> y <- sin(x)
> plot(x, y, type = "l", xlab = "x", ylab = "sin(x)")
```

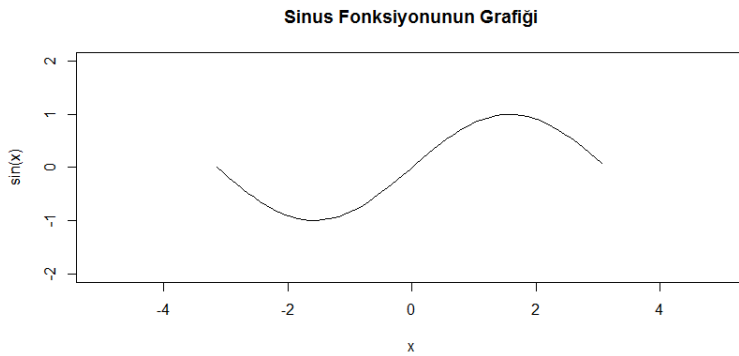
xlab ve ylab parametreleri belirtilmezse default durumda x ve y parametreleri için girilen değişken isimleri alınır.

main parametresi grafiğe başlık yerleştirmek için kullanılmaktadır. Örneğin:

```
> x <- seq(-3.14, 3.14, 0.1)
> y <- sin(x)
> plot(x, y, xlab = "x", ylab = "sin(x)", main = "Sinus Fonksiyonunun Grafiği")
```

plot fonksiyonunun xlim ve ylim parametreleri eksenlerin görüntülünecek aralık değerlerini belirlemekte kullanılır. Örneğin:

```
> x <- seq(-3.14, 3.14, 0.1)
> y <- sin(x)
> plot(x, y, xlab = "x", ylab = "sin(x)", main = "Sinus Fonksiyonunun Grafiği")
```



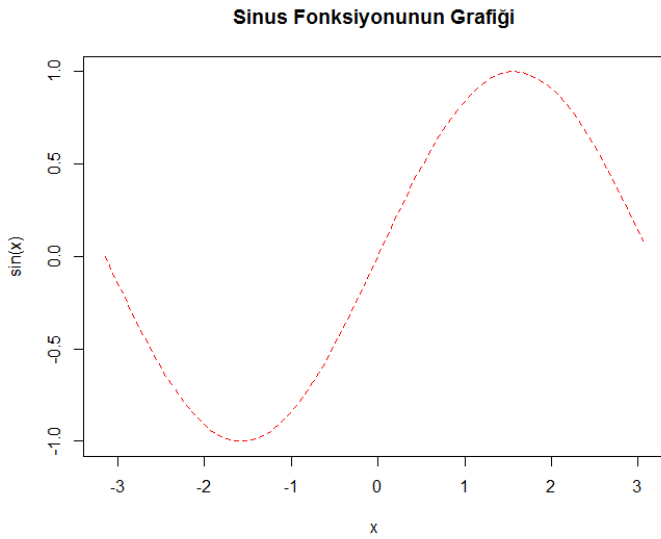
col parametresi (color sözcüğünden kısaltma) çizilen grafiği şekil rengini belirlemekte kullanılır. asp (aspect ratio) parametresi x / y oranını belirlemek için kullanılır. Bu oranın default değeri 1'dir. Örneğin:

lty parametresi çizginin tipini belirlemek için kullanılmaktadır. lty değerleeri şunlardan biri olabilir:

- 1 Düz çizgi
- 2 Kesikli çizgi
- 3 Nokta
- 4 Nokta Çizgi
- 5 Uzun çizgi
- 6 İki çizgi

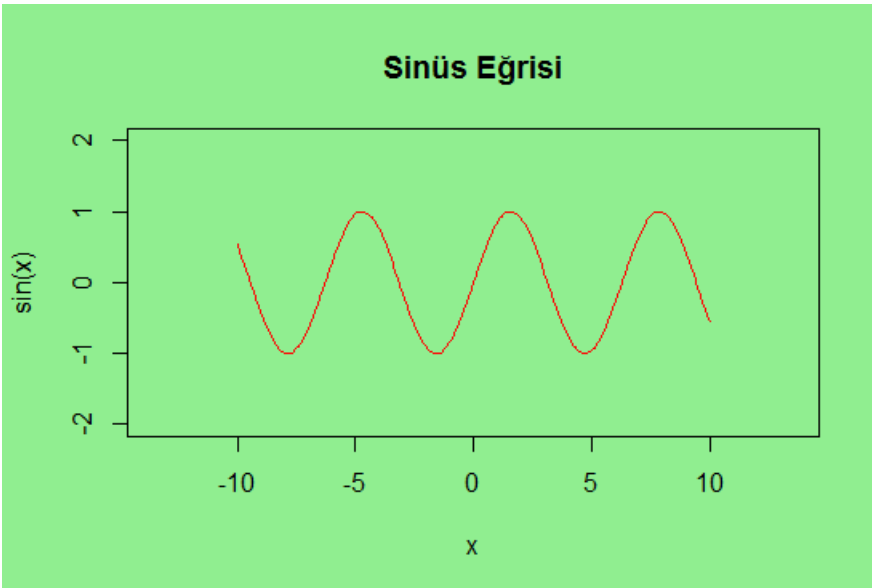
Örneğin:

```
> x <- seq(-3.14, 3.14, 0.1)
> y <- sin(x)
> plot(x, y, xlab = "x", ylab = "sin(x)", main = "Sinus Fonksiyonunun Grafiği", type = "l", col = "red", lty = 2)
```



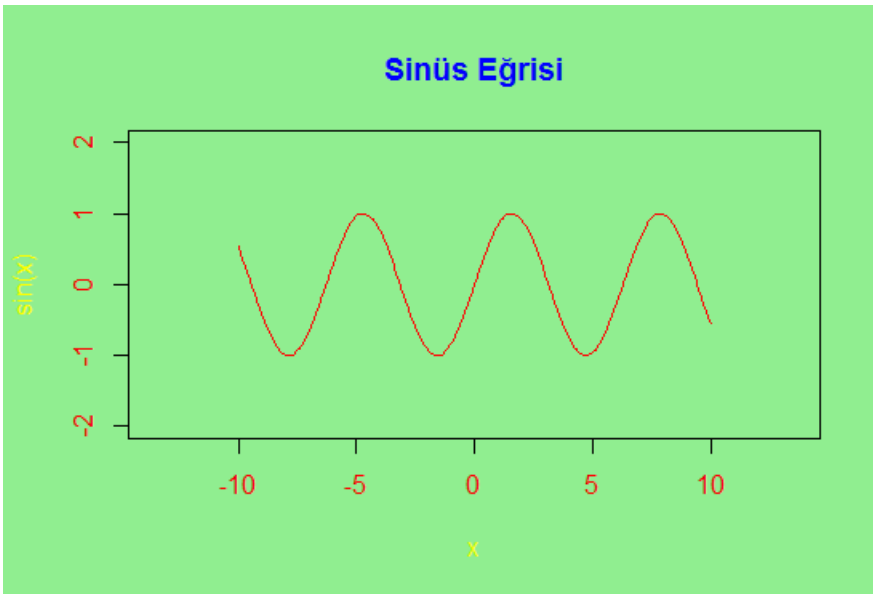
par isimli fonksiyonla grafik parametreleri global olarak değiştirilebilir. Yani par fonksiyonuyla bazı değerleri set ettiğimizde artık her grafikte o değerler o biçimde ele alınır. Örneğin par fonksiyonunda bg (background) grafik zemininin rengini belirlemekte kullanılır. Örneğin:

```
> par(bg = "lightgreen")
> plot(x, y, xlab = "x", ylab = "sin(x)", type = "l", main = "Sinüs Eğrisi", xlim = c(-10, 10),
ylim = c(-2, 2), col = "red", asp = 3)
```



par fonksiyonun col, col.axis, col.lab ve col.main parametreleri sırasıyla grafiğin default şeklini, eksen renklerini, eksen isimlerinin renklerini ve başlık yazısının renklerini belirlemek için kullanılır. Örneğin:

```
> par(bg = "lightgreen", col.axis = "red", col.lab = "yellow", col.main = "blue")
> plot(x, y, xlab = "x", ylab = "sin(x)", type = "l", main = "Sinüs Eğrisi", xlim = c(-10, 10),
ylim = c(-2, 2), col = "red", asp = 3)
```

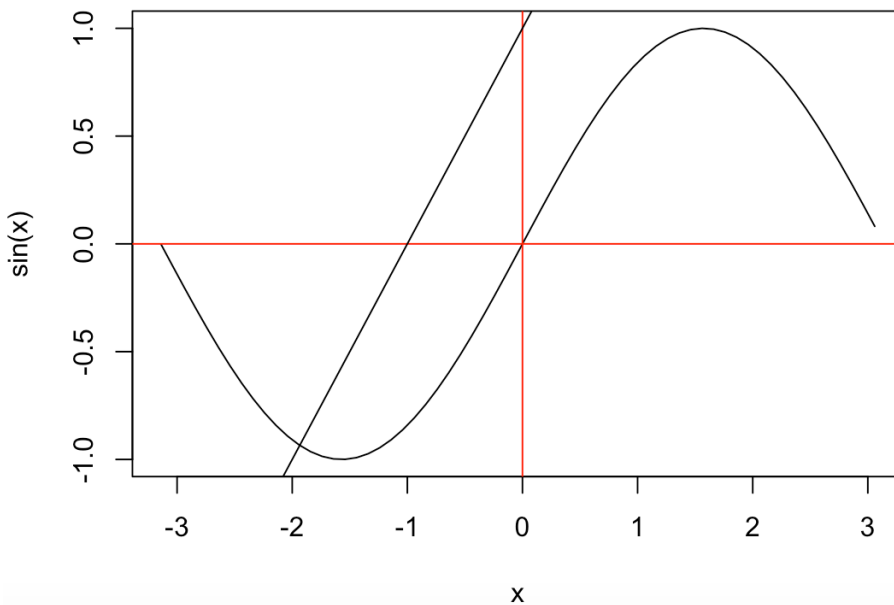


Çizilmiş bir grafiğe abline fonksiyonuyla doğrular ekleyebiliriz. Fonksiyonun parametrik yapısı şöyledir:

```
abline(a = NULL, b = NULL, h = NULL, v = NULL, reg = NULL,
       coef = NULL, untf = FALSE, ...)
```

a ve b parametreleri $y = bx + a$ doğrusundaki a ve b değerlerini belirtir. Fonksiyon her çağrıldığında bir doğruyu grafiğe eklemektedir. Fonksiyonun v ve h parametreleri sırasıyla düşey ve yatay eksenleri çizmek için kullanılmaktadır. Örneğin:

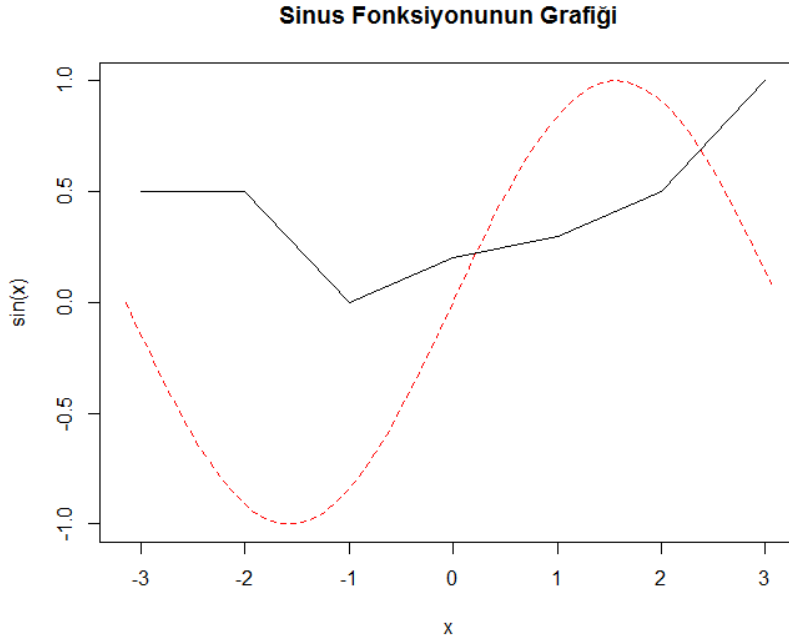
```
> x <- seq(-3.14, +3.14, 0.1)
> y <- sin(x)
> y <- sin(x)
> plot(x, y, type = "l", xlab = "x", ylab = "sin(x)")
> abline(v = 0, h = 0, col = "red")
> abline(1, 1)
```



lines fonksiyonu ile biz bir grup noktadan doğrular elde edip onu grafiğe ekleyebiliriz. Örneğin:

```
> x <- seq(-3.14, 3.14, 0.1)
```

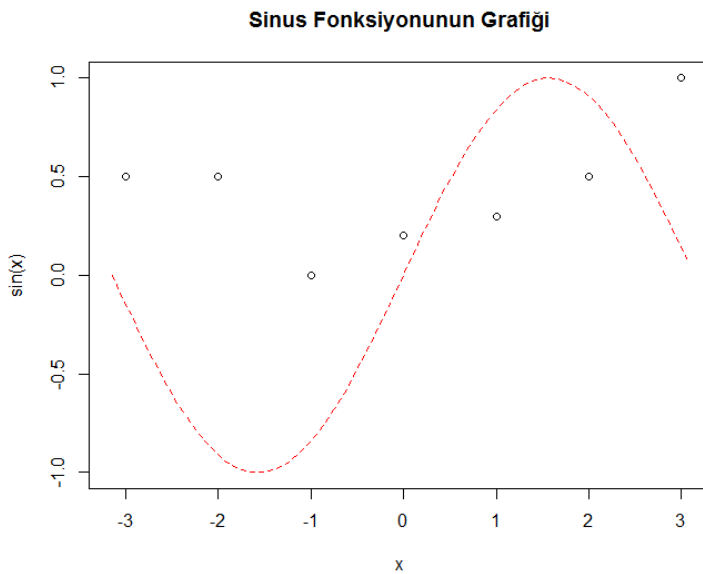
```
> y <- sin(x)
> plot(x, y, xlab = "x", ylab = "sin(x)", main = "Sinus Fonksiyonunun Grafiği", type = "l", col
= "red", lty = 2)
> lines(c(-3, -2, -1, 0, 1, 2, 3), c(0.5, 0.5, 0, 0.2, 0.3, 0.5, 1))
```



lines fonksiyonu da type gibi pch gibi col gibi parametrelere sahiptir.

points fonksiyonu grafiğe bir grup noktayı eklemek için kullanılır. Örneğin:

```
> x <- seq(-3.14, 3.14, 0.1)
> y <- sin(x)
> plot(x, y, xlab = "x", ylab = "sin(x)", main = "Sinus Fonksiyonunun Grafiği", type = "l", col
= "red", lty = 2)
> points(c(-3, -2, -1, 0, 1, 2, 3), c(0.5, 0.5, 0, 0.2, 0.3, 0.5, 1))
```



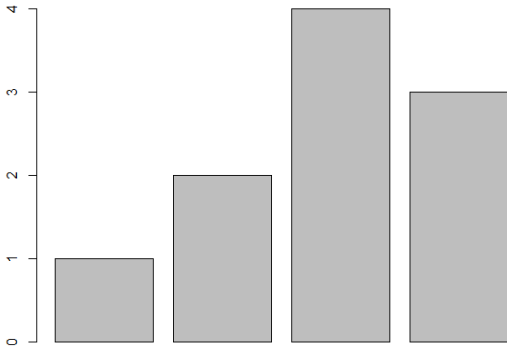
Bu fonksiyonda da pch ve col gibi parametreler bulunmaktadır.

Çizgi grafiği (bar chart) çizmek için temel paketdeki barplot fonksiyonu kullanılmaktadır. Fonksiyonun parametrik yapısı şöyledir:

```
barplot(height, width = 1, space = NULL,  
        names.arg = NULL, legend.text = NULL, beside = FALSE,  
        horiz = FALSE, density = NULL, angle = 45,  
        col = NULL, border = par("fg"),  
        main = NULL, sub = NULL, xlab = NULL, ylab = NULL,  
        xlim = NULL, ylim = NULL, xpd = TRUE, log = "",  
        axes = TRUE, axisnames = TRUE,  
        cex.axis = par("cex.axis"), cex.names = par("cex.axis"),  
        inside = TRUE, plot = TRUE, axis.lty = 0, offset = 0,  
        add = FALSE, args.legend = NULL, ...)
```

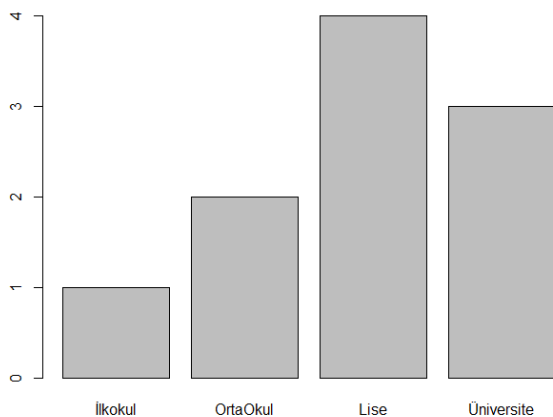
Fonksiyonun birinci parametresi olan height sıklık değerlerini belirtir. Örneğin:

```
> barplot(c(1, 2, 4, 3))
```



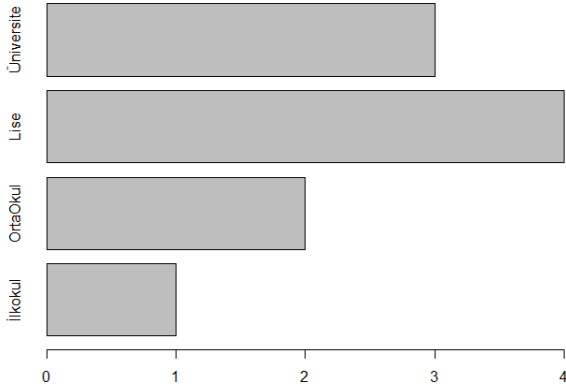
Çubukları isimlendirmek için kullanılan vektörün isimlendirilmesi gerekir. Örneğin:

```
> barplot(c(İlkokul = 1, OrtaOkul = 2, Lise = 4, Üniversite = 3))
```



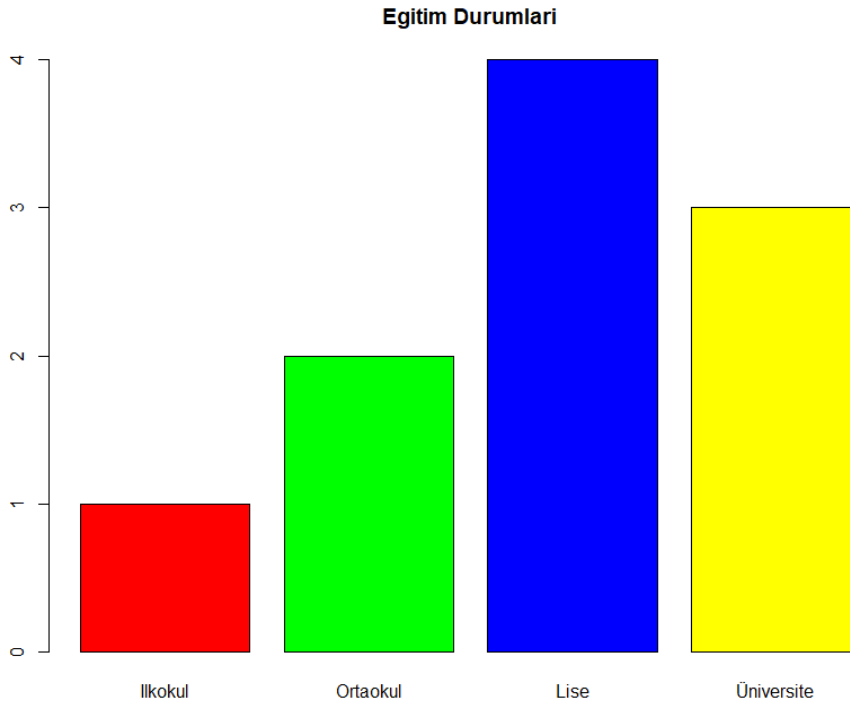
Fonksiyonun plot fonksiyonundaki pek çok parametresi ortaktır. horiz parametresi çubuk grafiğini yatay göstermek için kullanılır. Bu parametrenin default değeri FALSE biçimdedir. Örneğin:

```
> barplot(c(İlkokul = 1, OrtaOkul = 2, Lise = 4, Üniversite = 3), horiz = T)
```



Çubuk grafiğinde her çubuğa ayrı bir renk verebiliriz. Bunun için col parametresine renk vektörünün girilmesi gerekir. Örneğin:

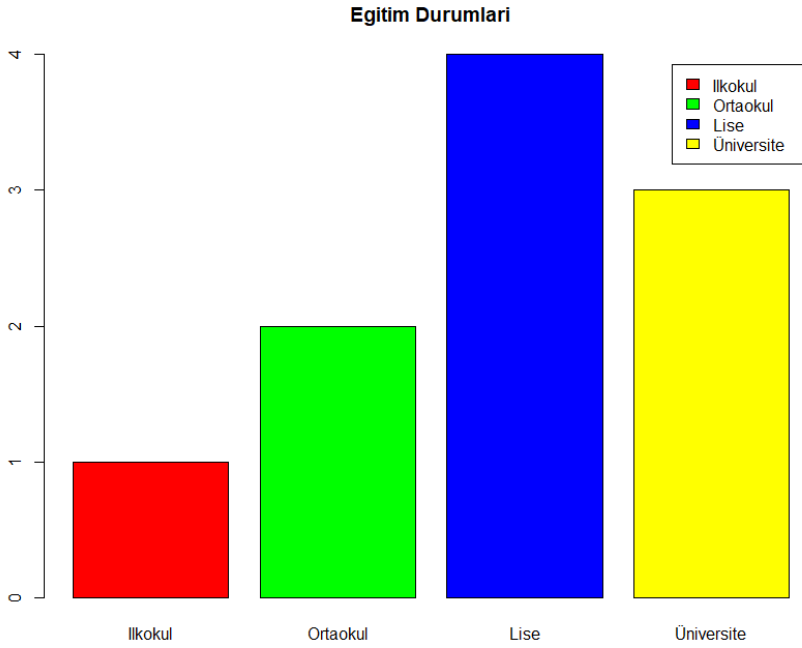
```
> barplot(c(İlkokul = 1, Ortaokul = 2, Lise = 4, Üniversite = 3), col = c("red", "green", "blue", "yellow"), main = "Eğitim Durumları")
```



Barplot'ta elemanları names.arg parametresi ile ayrıca da isimlendirebiliriz. Örneğin:

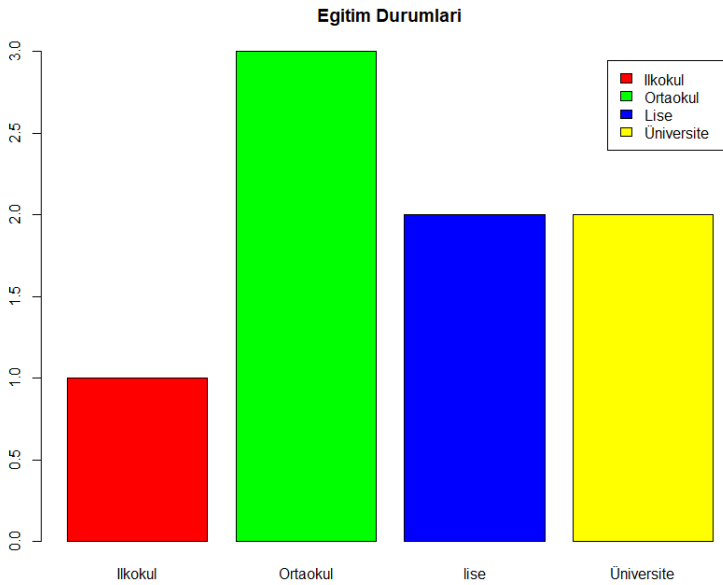
Böylece elemanlar için görüntülenecek isim boşluk karakterleri ya da özel karakterler içerebilir. barplot fonksiyonunun legend.text parametresiyle çizilen çubuklar için legend verebiliriz. Örneğin:

```
barplot(c(1, 2, 4, 3), col = c("red", "green", "blue", "yellow"), main = "Eğitim Durumları",  
names.arg = c("İlkokul", "Ortaokul", "Lise", "Üniversite"),  
legend.text = c("İlkokul", "Ortaokul", "Lise", "Üniversite"))
```



Çubuk grafiği doğrudan bir table parametresi verilerek de oluşturulabilir. Zaten anımsanacağı gibi table aslında sıklık belirten bir yapıdır. Örneğin:

```
> t <- table(c("İlkokul", "Ortaokul", "Ortaokul", "Lise", "Lise", "Üniversite", "Üniversite", "Lise"))
> barplot(t, names.arg = c("İlkokul", "Ortaokul", "lise", "Üniversite"), col = c("red", "green", "blue", "yellow"), main = "Eğitim Durumları", legend = c("İlkokul", "Ortaokul", "Lise", "Üniversite"))
```

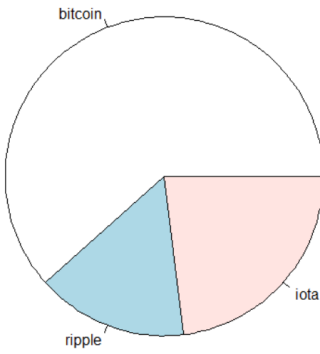


Pasta dilimi grafikleri birkaç değişken için bilginin oransal bir biçimde görüntülenmesinde çok kullanılmaktadır. Pasta dilimi grafiği de nominal değerlerin sıklığı üzerinde ya da nümerik değerler üzerinde kullanılabilir. Pasta dilimi grafiği pie fonksiyonuyla oluşturulur. Fonksiyonun parametrik yapısı şöyledir:

```
pie(x, labels = names(x), edges = 200, radius = 0.8,
    clockwise = FALSE, init.angle = if(clockwise) 90 else 0,
    density = NULL, angle = 45, col = NULL, border = NULL,
    lty = NULL, main = NULL, ...)
```

Örneğin:

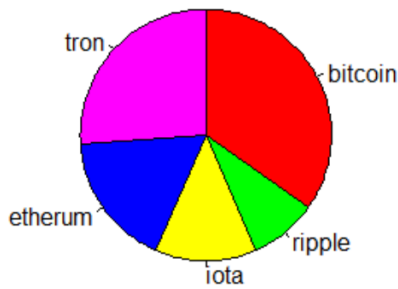
```
> pie(c(8, 2, 3), labels = c("bitcoin", "ripple", "iota"))
```



pie fonksiyonunun yine renklendirme için bir col parametresi, başlık için main parametresi vardır. clockwise parametresi elemanların saat yönünde mi yoksa ters yönde mi konumlandırılacağını belirtir. Örneğin:

```
> pie(c(8, 2, 3, 4, 6), labels = c("bitcoin", "ripple", "iota", "ethereum", "tron"), col = c("red", "green", "yellow", "blue", "magenta"), main = "Digital currencies", clockwise = T)
```

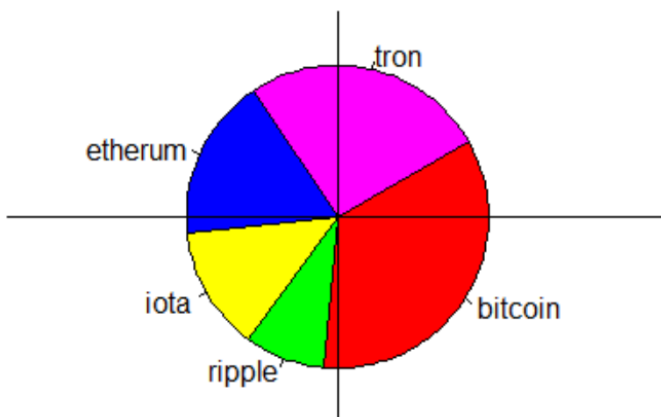
Digital currencies



init.angle parametresi ilk dilimin hangi açıdan başlatılacağını belirtir. Örneğin:

```
> pie(c(8, 2, 3, 4, 6), labels = c("bitcoin", "ripple", "iota", "ethereum", "tron"), col = c("red", "green", "yellow", "blue", "magenta"), main = "Digital currencies", clockwise = T, init.angle = 30)
> abline(v = 0, h = 0)
```

Digital currencies

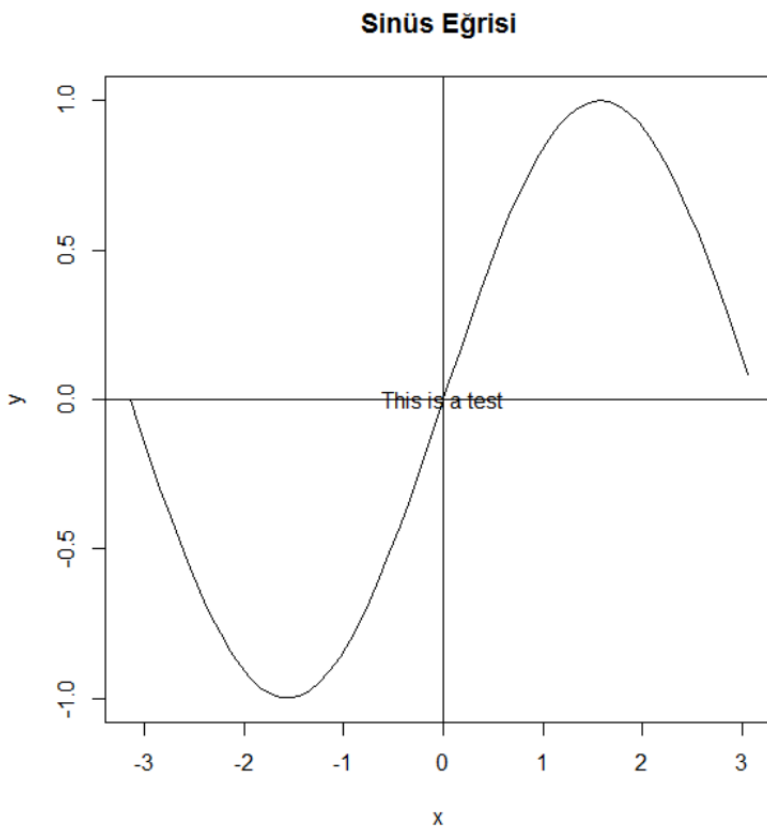


text fonksiyonu mevcut bir grafiğin içerisinde belli bir koordinatta yazı eklemek için kullanılır. Fonksiyonun parametrik yapısı şöyledir:

```
text(x, y = NULL, labels = seq_along(x$x), adj = NULL,  
     pos = NULL, offset = 0.5, vfont = NULL,  
     cex = 1, col = NULL, font = NULL, ...)
```

x ve y parametresi yazının ortasındaki koordinatı belirtir. Yani yazı bu noktaya ortalanmaktadır. labels parametresi ise yazılacak yazıyı belirtmektedir. Örneğin:

```
> x <- seq(-3.14, 3.14, 0.1)  
> y <- sin(x)  
> plot(x, y, main = "Sinüs Eğrisi", type = "l")  
> text(0, 0, "This is a test")  
> abline(v = 0, h = 0)
```



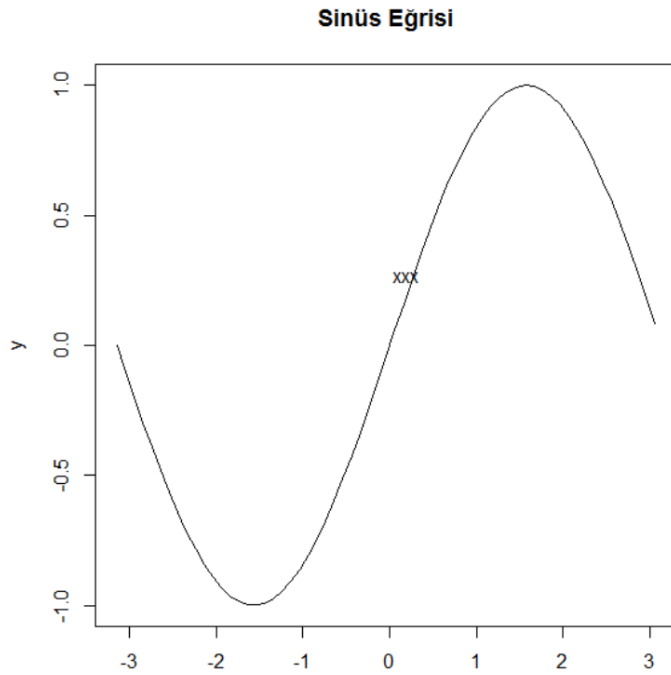
Tabii bir for döngüsü ile biz bir grup yazıyı istediğimiz yerlere basabiliriz. Örneğin:

```
> x <- seq(-3.14, 3.14, 0.1)  
> y <- sin(x)  
> plot(x, y, main = "Sinüs Eğrisi", type = "l")  
> i <- 1  
> for (k in seq(1, -1, -0.5)) {  
+   text(-3, k, i)  
+   i <- i + 1  
+ }
```

locator isimli fonksiyon bize grafikte tıklanan yerin x ve y koordinatlarını verir. Böylece biz tıklanan yere text fonksiyonu ile yazı yazabiliriz. Örneğin:

```
> x <- seq(-3.14, 3.14, 0.1)  
> y <- sin(x)
```

```
> plot(x, y, main = "Sinüs Eğrisi", type = "l")
> text(locator(1), "xxx")
```



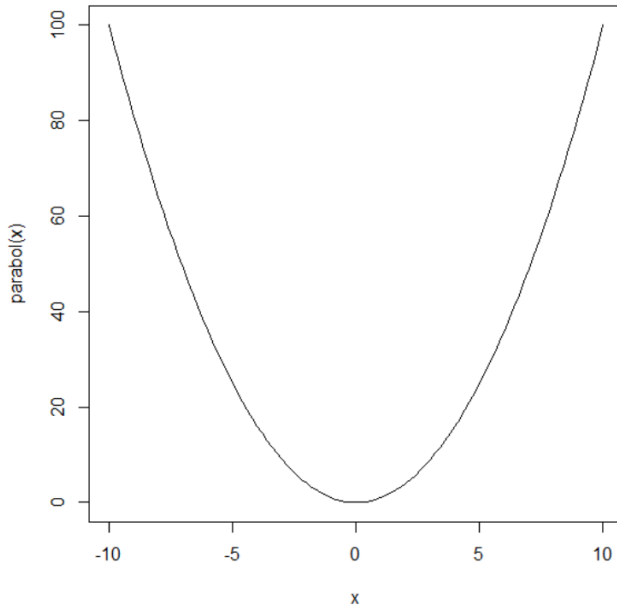
locator fonksiyonun parametresi toplam kaç kere fare ile tıklanacağını belirtir. Yani biz istersek daha fazla tıklayarak tüm tıkladığımız yerlerin koordinatlarını alabiliriz.

Bazen matematiksel fonksiyonların grafiklerini çizmek isteyebiliriz. Bu işlem plot fonksiyonuyla yapılabilirse de bunun için en pratik yöntem curve fonksiyonunu kullanmaktır. Fonksiyonun parametrik yapısı şöyledir:

```
curve(expr, from = NULL, to = NULL, n = 101, add = FALSE,
      type = "l", xname = "x", xlab = xname, ylab = NULL,
      log = NULL, xlim = NULL, ...)
```

Fonksiyonun birinci parametresi bizden bir fonksiyon alır. Sonra from'dan to'ya kadar toplan n tane değer için elde edilen değerler bu fonksiyona sokulur ve bunun karşılığında değerler elde edilir. Onun grafiği çizilir Örneğin:

```
> parabol <- function(x) x^2
> curve(parabol, -10, 10)
```



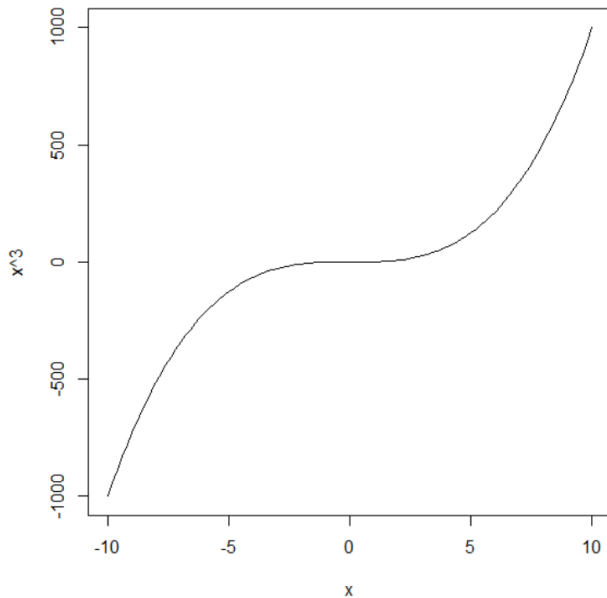
Örneğin:

```
f <- function(x) 1 / (x + 1)
> curve(f, -10, 10)
```

curve fonksiyonunun n parametresi kaç tane nokta elde edileceğini belirlemekte kullanılır. Bu parametre default olarak 101 değerini almaktadır.

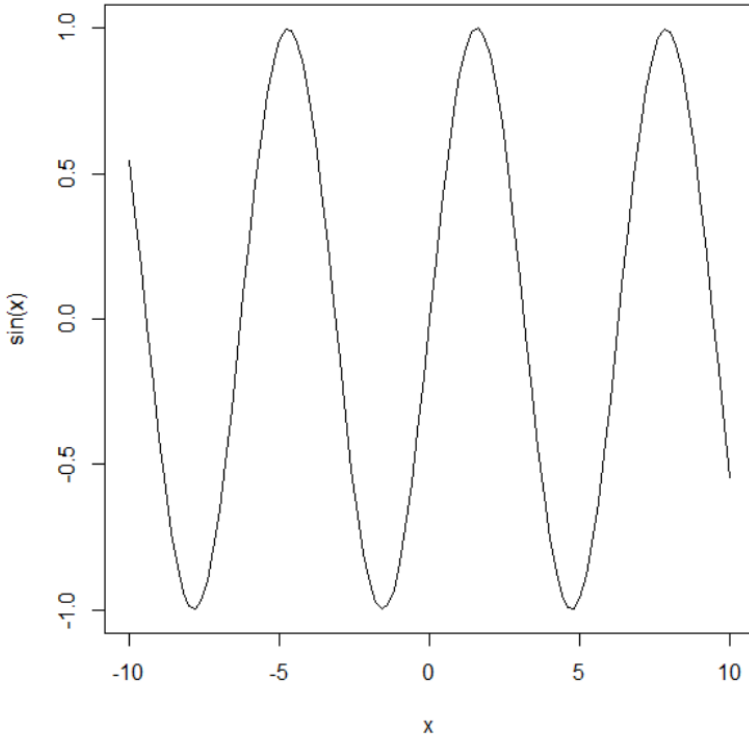
Eğer curve fonksiyonunun birinci parametresi x'e dayalı bir ifade ise bunu fonksiyon olarak vermeye gerek yoktur. Örneğin:

```
> curve(x^3, -10, 10)
```



Örneğin:

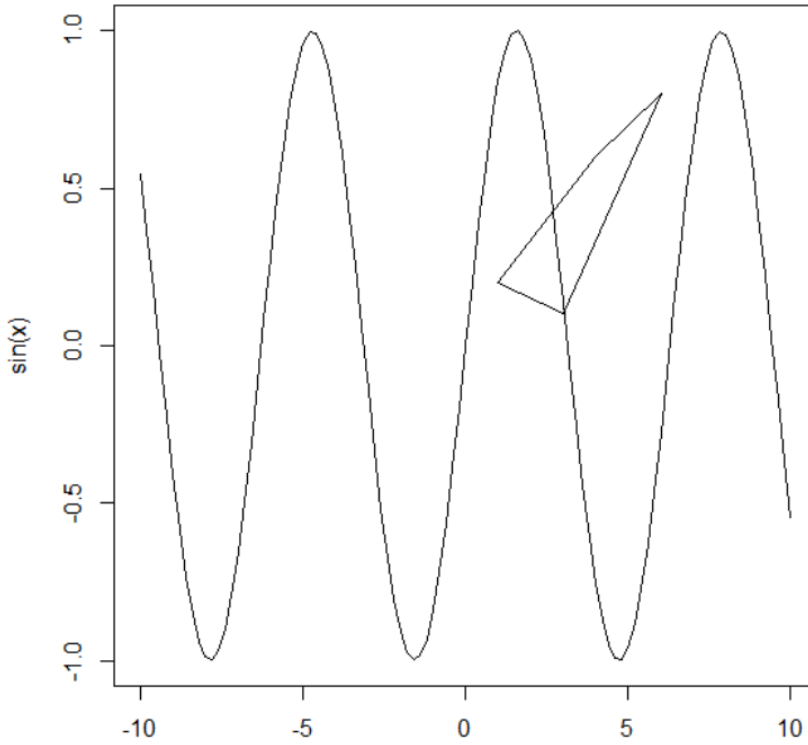
```
> curve(sin(x), -10, 10)
```



polygon fonksiyonu bizden noktalar kümesini iki ayrı fonksiyon olarak alır. Onları çizgilerle birleştirir. Son noktayı da ilkiyle birleştirerek kapalı bir şekil elde eder.

Örneğin:

```
> curve(sin(x), -10, 10)
> polygon(c(1, 3, 6, 4), c(0.2, 0.1, 0.8, 0.6))
```



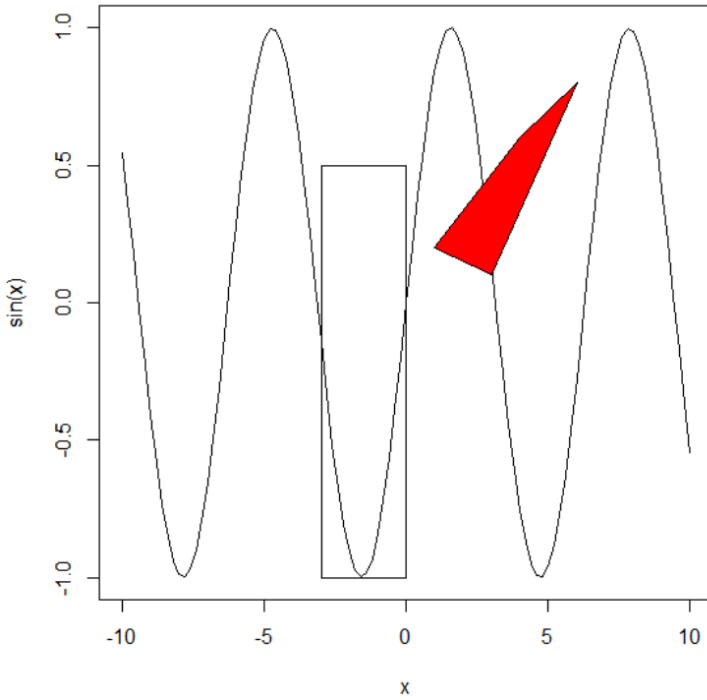
rect isimli fonksiyon sol üst köşe ve sağ alt köşe koordinatlarını alarak dikdörtgen çizer. Parametrik yapısı şöyledir:

```
rect(xleft, ybottom, xright, ytop, density = NULL, angle = 45,
```

```
col = NA, border = NULL, lty = par("lty"), lwd = par("lwd"),  
...)
```

Örneğin:

```
> curve(sin(x), -10, 10)  
> polygon(c(1, 3, 6, 4), c(0.2, 0.1, 0.8, 0.6))  
> polygon(c(1, 3, 6, 4), c(0.2, 0.1, 0.8, 0.6), col = "red")  
> rect(0, -1, -3, 0.5)
```

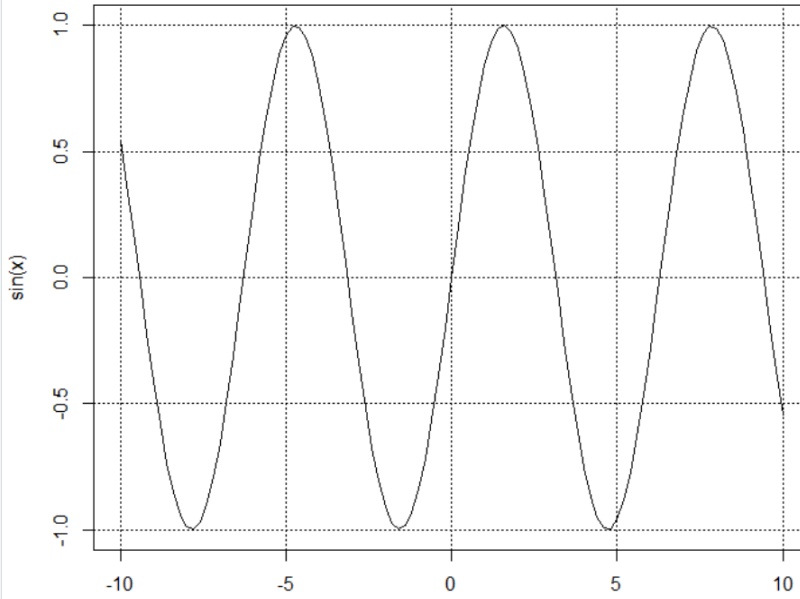


grid fonksiyonu grafikte ızgara çizgilerini çıkartmak için kullanılır. Parametrik yapısı şöyledir:

```
grid(nx = NULL, ny = nx, col = "lightgray", lty = "dotted",  
lwd = par("lwd"), equilogs = TRUE)
```

Örneğin:

```
> curve(sin(x), -10, 10)  
> grid()  
> grid(col = "black")
```



grid fonksiyonun çizgi rengini, biçimini ve çizgi sayısını belirlemekte kullanılan parametreleri vardır.

Eksenlerin özelleştirilmesi axis isimli fonksiyonla yapılmaktadır. axis fonksiyonun parametrik yapısı şöyledir:

```
axis(side, at = NULL, labels = TRUE, tick = TRUE, line = NA,  
      pos = NA, outer = FALSE, font = NA, lty = "solid",  
      lwd = 1, lwd.ticks = lwd, col = NULL, col.ticks = NULL,  
      hadj = NA, padj = NA, ...)
```

Burada side parametresi eksenin hangi olduğunu belirlemekte kullanılır. Yatay eksen için 1 değeri, düşey eksen için 2 değeri kullanılmalıdır. at parametresi eksenindeki tick değerlerini belirtir. Örneğin:

```
> curve(sin(x), -10, 10)  
> axis(1, at = -10:10)  
> axis(2, seq(-1, 1, 0.1))
```

Grafiklerin Dosyalarda Saklanması

R'da grafikler grafik aygıtlarında görüntülenir. Bir grafik aygıtı GUI ekran olabileceği gibi, bir dosya ya da yazıcı olabilir. O andaki grafik aygıtları dev.list fonksiyonuyla elde edilmektedir. Örneğin:

```
> dev.list()  
RStudioGD      png  
      2          3
```

Burada iki grafik aygıtı vardır. Bunlardan biri RStudioGD isimli RStudio'daki grafik ekranını belirtir. Diğeri de bir png dosyasıdır.

Bir grafik çizimi o anda aktif olan grafik aygıtı yapılmaktadır. Aktif grafik aygıtı elde etmek için dev.cur fonksiyonu kullanılır.

Çizimler o andaki aktif grafik aygıtına yapılmaktadır. Aktif grafik aygıtı dev_cur isimli fonksiyonla elde edilebilir. Örneğin :

```
> dev.cur()  
RStudioGD  
      2
```

Biz bir çizimi dosyada saklamak için iki yöntem izleyebiliriz. Birincisi png, jpeg ve pdf gibi fonksiyonlarla dosya temelli yeni bir grafik aygıtı oluşturmak ve çizimi onun içerisine doğrudan yapmak (tabii bu durumda biz çizimleri doğrudan görmeyiz). İkincisi de dev.copy fonksiyonuyla aktif aygıttaki çizimi istenilen aygıtı kopyalamaktır. png, jpeg ve pdf isimli fonksiyonlar hedefi png dosyası olan, jpeg dosyası olan ve pdf dosyası olan grafik aygıtlar oluştururlar. Bu fonksiyonlar bizden ilgili dosyanın yol ifadesini parametre olarak almaktadır. Yaratılan yeni grafik aygıtı artık default aygıt duruma gelir.

Örneğin:

```
> jpeg("test.jpg")
> dev.list()
RStudioGD      png      jpeg
      2          3          4
> dev.cur()
jpeg
  4
> curve(sin(x), -10, 10, main = "Sinüs Eğrisi")
> dev.off()
```

Burada jpeg fonksiyonuyla bir jpeg dosyası bir grafik aygıt biçiminde oluşturulmuştur. Artık bu grafik aygıt aktif durumdadır. Yani çizimler artık bu jpeg dosyasına yapılacaktır. dev.off() aktif aygıtı kapatır. Bu işlem sonucunda elimizde içerisinde sinüs eğrisi olan "test.jpg" dosyası bulunacaktır. Örneğin:

```
> pdf("test.pdf")
> curve(sin(x), -10, 10, main = "Sinüs Eğrisi")
> dev.off()
```

dev.off fonksiyonu aktif grafik aygıtını kapatmaktadır.

R'da Nesne Yönelimli Programlama

R fonksiyonel bir dil olmanın yanı sıra nesne yönelimli özelliklere de sahiptir. R'in nesne yönelimli özellikleri için zamanla üç biçim geliştirilmiştir. Bunlara S3, S4 ve S5 denilmektedir. En eski olan ve en geniş desteğe sahip olan S3 sınıflarıdır. Bunu S4 ve S5 izlemiştir. Bazı R programcıları yalnızca S3 kullanmaktadır. Yalnızca S4 kullananlar da vardır. S5 nispeten yenidir.

Nesne yönelimli programlamadan kast edilen şey birtakım olguları sınıflarla (class) temsil etmek ve bu sınıfları kullanarak hedefleri gerçekleştirmektir. R'da sınıf kavramı C++ gibi, Java ve C# gibi dillerdeki gibi ayrıntılı değildir. R'in sınıfsal özellikleri oldukça basit ve kısıtlıdır. S3, S4 ve S5 sistemlerinde türetme kavramı ve çokbiçimli davranış bulunmaktadır.

S3 Sınıf Sistemi

S3 sınıf sisteminde bir sınıf (class) bir vektörden ya da tipik olarak bir listeden oluşturulur. Bir vektörü ya da listeyi bir sınıfa dönüştürmek için tek yapılacak şey onun "class" isimli özniteliğine bir isim niteliğinde değer atamaktır. Bu işlem iki biçimde yapılabilir. Bunun için biz daha önce görmüş olduğumuz attr fonksiyonundan faydalanabiliriz. Örneğin:

```
> l <- list(name = "kaan", no = 123)
> attr(l, "class") <- "myclass"
```

class isimli yer değiştirme fonksiyonu zaten vektörün "class" özniteliğine atama yapmak için kullanılmaktadır. Yani yukarıdaki işlemin eşdeğeri şöyledir:

```
> l <- list(name = "kaan", no = 123)
> class(l) <- "myclass"
```

Tabii gerek attr gerekse class fonksiyonlarının normal biçimleri de vardır. Örneğin:

```
> attr(1, "class")
[1] "myclass"
> class(1)
[1] "myclass"
```

Gördüğünüz gibi bir değişkenin hangi sınıfa ilişkin olduğunu class fonksiyonuyla hemen elde edebilmekteyiz. Örneğin:

```
> v <- 1:10
> class(v)
[1] "integer"
> df <- list(name = "ali", no = 123)
> class(df)
[1] "list"
```

Farklı değişkenlerin "class" özniteliklerine aynı ismi verirse bunlar aynı sınıftan olur. Örneğin her liste aslında aynı "list" isimli sınıftandır. Örneğin:

```
> a <- list(name = "ali", no = 123)
> b <- list(city = "eskişehir", plate = "26")
> class(a) <- "myclass"
> class(b) <- "myclass"
```

Tabii nesne yönelimli programlamada sınıf türünden değişkenler (instance) aynı elemanlara sahip oldukları için isimleri aynı olan ancak değerleri farklı olan liste türünden değişkenlere aynı sınıf ismini vermek daha uygundur. Örneğin:

```
> a <- list(name = "ali", no = 123)
> b <- list(name = "salih", no = 345)
> class(a) <- "myclass"
> class(b) <- "myclass"
```

Burada a ve b name ve no elemanlarına sahip "myclass" türünden değişkenlerdir.

Biz bir fonksiyon yazıp o fonksiyonun arzu ettiğimiz bir sınıf türünden liste vermesini sağlayabiliriz. Örneğin:

```
createPlot <- function(x, y)
{
  l<- list(x = x, y = y)
  class(l) <- "plot"
  l
}
```

Burada createPlot isimli fonksiyon bize "plot" isimli bir sınıf türünden nesne vermiştir. Biz de geri dönüş değeri olarak elde ettiğimiz nesne üzerinde print fonksiyonunu uygularsak print.plot isimli fonksiyon çağrılacaktır. print.plot fonksiyonu şöyle yazılmış olsun:

```
print.plot <- function(l)
{
  plot(l$x, l$y)
}
```

Şimdi createPlot fonksiyonunu çağırıp sonucu print edelim:

```
> c <- createPlot(c(1, 2, 3), c(4, 5, 6))
```


R'daki pek çok fonksiyon da aslında yukarıdaki createPlot gibi işlemler yapmaktadır. Yani bunlar bir liste yaratıp elemanları set edip onun "class" özniteliğini değiştirerek bize başka bir sınıf türünden nesne gibi vermektedir. Örneğin:

```
> df <- data.frame(numbers = c(1, 2, 3), names = c("ali", "veli", "selami"))
> class(df)
[1] "data.frame"
> typeof(df)
[1] "list"
> df
  numbers names
1       1   ali
2       2   veli
3       3 selami
```

Aslında data.frame fonksiyonu kendi içerisinde bir liste yaratıp verdiğimiz değerleri bu listeye yerleştirip listenin de "class" özniteliğini "data.frame" yaptıktan sonra onu bize vermektedir. "data frame" aslında bir listedir.

Sınıfın Sınıf Özelliğinin Kaldırılması

Bir sınıfı sınıf yapan aslında "class" isimli özniteliğe atama yapılmış olmasıdır. Biz bu atamayı ortadan kaldırırsak ilgili nesneyi yine orijinal türden (vektör ya da liste) elde edebiliriz. Bu işlem iki biçimde yapılabilir.

- 1) "class" özniteliğine NULL değerini atamak
- 2) unclass fonksiyonunu kullanmak. Aslında unclass fonksiyonu zaten "class" özniteliğine NULL değerini atamaktadır.

Örneğin:

```
print.employee <- function(e)
{
  cat("Adı Soyadı:", e$name, ", Maaş:", e$salary)
}

> e <- list(name = "Kaan Aslan", salary = 12300)
> class(e) <- "employee"
> e
Adı Soyadı: Kaan Aslan Maaş: 12300
```

Şimdi e nesnesinin "class" özniteliğine NULL atayalım:

```
> class(e) <- NULL
> e
$name
[1] "Kaan Aslan"

$salary
[1] 12300
```

Görüldüğü gibi artık print.employee fonksiyonu devreye girmemiştir. Aynı işlemi şöyle de yapabiliriz:

```
> ue <- unclass(e)
> ue
Adı Soyadı: Kaan Aslan Maaş: 12300
> ue
$name
[1] "Kaan Aslan"

$salary
[1] 12300
```

Burada unclass fonksiyonunun parametresindeki nesneyi değiştirmedikçe "class" özniteliği kaldırılmış yeni bir nesneye geri döndüğüne dikkat ediniz.

S3 Sınıfı Nedir ve Neden Kullanılmaktadır?

Sınıf (class) kavramı Nesne Yönelimli Programlama Tekniğinin en önemli yapı taşını oluşturmaktadır. Biz kursumuzda bu tekniği ayrı bir başlık halinde değerlendirmeyeceğiz. Derneğimizdeki pek çok kursta NYPT ayrıntılarıyla ele alınmaktadır. Sınıf S3 sisteminde aslında "aynı elemanlara sahip olan değişkenleri" oluşturmak için kullanılmaktadır. Belli bir sınıfın belli elemanları vardır. O sınıfa ilişkin olan değişkenlerin de aynı elemanlara sahip olması beklenir. Ancak S3 sisteminde sınıf türünden değişkenlerin aynı elemanlara sahip olması zorunlu değildir.

Biz birden fazla elemana sahip olan olguları bir sınıf biçiminde ele alabiliriz. Örneğin tarih bilgisi gün, ay ve yıl bileşenlerinden oluşmaktadır. Bir tarih bilgisi bir sınıf biçiminde ifade edilebilir. S3 sisteminde bir sınıf oluşturmak için özel fonksiyonlar yoktur. Sınıflar vektörlerden ya da listelerden onların "class" özniteliği set edilerek oluşturulmaktadır. S3 sisteminde bir sınıf türünden nesne (değişken) yaratmak için de ayrı bir fonksiyon yoktur. R programcısı bir vektör ya da listeyi oluşturup onun class özniteliğini set eder. Tabii aynı sınıfların elemanlarının aynı olması en normal durumdur.

Şimdi bir tarih bilgisini bir sınıf olarak oluşturmak isteyelim. Pek çok nesne yönelimli dilde önce bir sınıf bildirimi yapılır sonra o sınıf türünden nesneler yaratılmaktadır. Fakat S3 sisteminde önce sınıf bildirilip sonra nesneler yaratılmaz. Zaten nesne oluşturulurken sınıf da oluşturulmuş olmaktadır. Örneğin:

```
> a <- list(day = 10, month = 12, year = 1990)
> class(a) <- "date"
> b <- list(day = 12, month = 11, year = 2007)
> class(b) <- "date"
```

Burada iki tane nesne (değişken) oluşturulmuş ve bu iki değişkene de "class" özniteliği olarak "date" atanmıştır. Bu durumda biz a ve b nesnelerinin date sınıfı türünden olduğunu söyleriz. Peki a ve b'yi benzer kılan şey nedir? Yanıt aynı elemanlara sahip olması ve aynı kavramın bir kopyası (instance) olması. Artık biz bir fonksiyonun parametresinin "date" sınıfı türünden olduğunu söylediğimizde bunun bileşik bir tarih bilgisi içeren bir kavram olduğu anlaşılır. R'in S3 sisteminde yukarıda da belirtildiği gibi ayrı bir sınıf bildirimi ve nesne yaratımı yoktur. Ancak programcı isterse belli bir sınıf türünden nesne yaratan fonksiyonlar yazabilir. Örneğin:

```
create.date <- function(d, m, y)
{
  l <- list(day = d, month = m, year = y)
  class(l) <- "date"
  l
}
```

```
> a <- create.date(10, 12, 1990)
> b <- create.date(12, 11, 2007)
```

Şimdi biz iki "date" nesnesini alıp onlar eşit mi diye karşılaştıran bir fonksiyon yazabiliriz:

```
isequal.date <- function(d1, d2)
{
  if (d1$day == d2$day && d1$month == d2$month && d1$year == d2$year)
    return(TRUE)
  return(FALSE)
}
```

```
> a <- create.date(10, 12, 1990)
> b <- create.date(12, 11, 2007)
> isequal.date(a, b)
[1] FALSE
```

Tabii fonksiyon içerisinde parametrelerin sınıf öznitelikleri kontrol edilip işlem sonlandırılabilir. Bir mesaj vererek çalışmayı sonlandırmak için stop fonksiyonu kullanılmaktadır. Örneğin:

```
isequal.date <- function(d1, d2)
{
  if (class(d1) != "date" || class(d2) != "date")
    stop("d1 and d2 parameter must be date type")

  if (d1$day == d2$day && d1$month == d2$month && d1$year == d2$year)
    return(TRUE)
  return(FALSE)
}
```

Şimdi isequal.date fonksiyonunu date sınıfından olmayan değerlerle çağırma deneyelim:

```
> isequal.date(10, 12)
Error in isequal.date(10, 12) : d1 and d2 parameter must be date type
```

Aslında R'in temel paketlerinde de pek çok fonksiyon bize çeşitli sınıflar türünden değerler vermektedir. Örneğin as.Date isimli fonksiyon bizden tarihi bir yazı olarak alır ve onu "Date" isimli bir sınıf nesnesi olarak bize verir:

```
> d <- as.Date("2009/12/11")
> d
[1] "2009-12-11"
> class(d)
[1] "Date"
```

Burada Date sınıfının print fonksiyonu (yani print.Date fonksiyonu) devreye girmiştir. O da tarihi bize yyyy-aa-ggg biçiminde bir yazı olarak görüntülemiştir. Date sınıfı aslında Double türünden tek elemanlı bir vektörden oluşmaktadır. O da belli bir tarihten geçen gün sayısını tutmaktadır:

```
> d <- as.Date("2009/12/11")
> d
[1] "2009-12-11"
> class(d)
[1] "Date"
> ud <- unclass(d)
> ud
[1] 14589
> typeof(ud)
[1] "double"
```

Şüphesiz tarihin belli bir tarihten geçen gün sayısı biçiminde ifade edilmesi bizim kullanışsızdır. Bilgisayarın saatinden o andaki tarihi alıp bize Date sınıf nesnesi olarak veren sys.Date isimli bir fonksiyon vardır. Örneğin:

```
> d <- Sys.Date()
> d
[1] "2018-02-17"
> class(d)
[1] "Date"
> unclass(d)
[1] 17579
```

Peki biz bu tarih bilgisinin gün, ay ve yıl bileşenlerini nasıl alabiliriz. İşte Date sınıf nesnesini alarak bize POSIXlt isimli bir başka sınıf veren as.POSIXlt fonksiyonu vardır. POSIXlt sınıfı bir liste türündendir. Bu listenin elemanları bize tarih bileşenlerini vermektedir.

```
> d <- Sys.Date()
```

```

> pd <- as.POSIXlt(d)
> class(pd)
[1] "POSIXlt" "POSIXt"
> upd <- unclass(pd)
> upd
$sec
[1] 0

$min
[1] 0

$hour
[1] 0

$mday
[1] 17

$mon
[1] 1

$year
[1] 118

$yday
[1] 6

$yday
[1] 47

$isdst
[1] 0

attr(,"tzone")
[1] "UTC"

```

Sınıf İle İlişkili Fonksiyonların İsimlendirmesi

R'da geleneksel olarak bir sınıf ile ilgili işlemler yapan fonksiyonların isimleri <fonksiyon ismi>.<sınıf ismi> biçiminde belirtilmektedir. Örneğin:

```

as.Date
print.Date

```

fonksiyonları Date sınıfı ile ilgilidir. Biz de fonksiyonun ismine bakarak onun hangi sınıf ile ilgili olduğunu anlayabiliriz.

S3 Sisteminde Türetme

Türetme (derivation) ya da kalıtım (inheritance) NYPT'nin önemli anahtar kavramlarından biridir. Türetme mevcut bir sınıfa eleman ekleyerek yeni bir sınıf oluşturma anlamına gelir. NYPT terminolojisinde mevcut sınıfa "taban sınıf (base class)" oluşturulan yeni sınıfa da "türemiş sınıf (derives class)" denilmektedir. R'ın S3 sisteminde ilkel de olsa bir türetme kavramı vardır.

S3 sisteminde türetme yapmak için tek yapılacak şey "class" özniteliğine tek elemanlı değil iki elemanlı bir string vektör atamaktır. (Aslında R'da bu anlamda çoklu türetme de vardır. Yani "class" özniteliğine iki elemandan daha fazla elemanlı bir string vektör de atanabilir.) Örneğin:

```

> e <- list(name = character(0), no = integer(0))
> class(e) <- "employee"
> w <- list(name = "Kaan Aslan", no = 1234, salary = 1200)
> class(w) <- c("worker", "employee")

```

```

> w
$name
[1] "Kaan Aslan"

$no
[1] 1234

$salary
[1] 1200

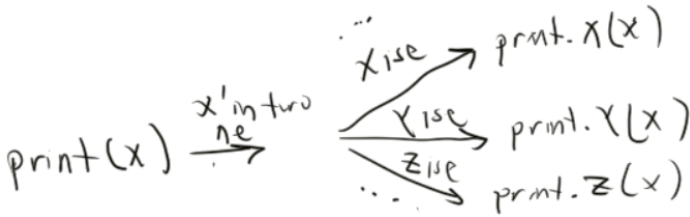
attr(,"class")
[1] "worker" "employee"

```

R'in S3 sisteminde türetme çokbiçimlilik konusunda bir etki göstermektedir.

S3 Sisteminde Çokbiçimlilik (Polymorphism)

Çokbiçimlilik (polymorphism) NYPT'nin önemli anahtar kavramlarından biridir. Çokbiçimlilik biyolojiden aktarılmış bir terimdir. Çokbiçimlilik biyolojide canlıların çeşitli doku ve organlarının temel işlevleri aynı kalmak üzere türler arasında farklılık göstermesine denilmektedir. Yazılımda ise çokbiçimlilik türden bağımsız program kodlarını oluşturmak için bir teknik olarak kullanılır. Örneğin print fonksiyonu neyi print etmektedir? (Komut satırında bir ifadeyi yazıp ENTER tuşuna bastığımızda aslında print fonksiyonu çağrılmaktadır.) İşte print fonksiyonunun parametresi hangi sınıf türündense o sınıfa özgü print fonksiyonuyla yapılmaktadır. Başka bir deyişle aslında print fonksiyonu başka bir fonksiyonu çağırarak bir sarma fonksiyon gibidir. print yazdırılmak istenen değişkenin türüne bakar o hangi sınıf türündense print.X fonksiyonunu çağırır. Burada X sınıfın ismini belirtmektedir.



Görüldüğü gibi print fonksiyonu aslında genel bir fonksiyon gibidir. R'da türe dayalı olarak başka fonksiyonları çağırarak bu tür fonksiyonlara "generic fonksiyonlar" denilmektedir. Generic fonksiyonlar sayesinde biz belli bir değişkenin türünü bilmeden genel işlemler yapabilmekteyiz. Diğer nesne yönelimli dillerin bir bölümünde bu tür fonksiyonlara "sanal fonksiyonlar (virtual functions)" denilmektedir. Çokbiçimlilik bu tür fonksiyonlarla sağlanmaktadır.

S3 sisteminde genel olarak bir sınıf ile ilişkili bir fonksiyonun isimlendirilmesi "<fonksiyon ismi>.<sınıf ismi>" biçiminde yapılmaktadır. Örneğin myclass sınıfına ilişkin print fonksiyonu print.myclass biçiminde isimlendirilir. Şimdi böyle bir fonksiyon yazalım:

```

create.myclass <- function(name, no)
{
  mc <- list(name = name, no = no)
  class(mc) <- "myclass"
  mc
}

print.myclass <- function(x)
{
  cat("Adı Soyadı:", x$name, ", No:", x$no)
}

```

Şimdi "myclass" isimli sınıf türünden bir nesne yaratalım:

```

> mc <- create.myclass("Kaan Aslan", 123)

```

Artık biz "myclass" türünden bir nesneyi print etmek istediğimizde print bunun için print.myclass isimli fonksiyonu çağıracaktır.

```
> print(mc)
Adı Soyadı: Kaan Aslan , No: 123
```

Yukarıda da belirtildiği gibi komut satırında aslında biz bir değişkenin ismini yazıp ENTER tuşuna bastığımızda komut satırı programları bu değişkeni print fonksiyonu çağırarak ekrana yazdırmaktadır. Örneğin:

```
> mc
Adı Soyadı: Kaan Aslan , No: 123
```

Bir generic fonksiyon için sınıflarda yazılmış olan fonksiyonları methods isimli fonksiyonla elde edebiliriz. Örneğin:

```
methods("print")
[1] print.acf*
[2] print.anova*
[3] print.aov*
[4] print.aovlist*
[5] print.ar*
[6] print.Arima*
[7] print.arima0*
[8] print.AsIs
[9] print.aspell*
[10] print.aspell_inspect_context*
[11] print.bibentry*
[12] print.Bibtex*
...
```

İşte R'in temel paketlerinde print gibi pek çok generic fonksiyon vardır. Pekiyi biz bir generic fonksiyonu nasıl yazabiliriz? S3 sisteminde generic fonksiyonlar UseMethod isimli fonksiyon kullanılarak yazılmaktadır. Örneğin print fonksiyonu şöyle yazılmıştır:

```
print <- function(x, ...) UseMethod("print")
```

Yani print fonksiyonunun kodu aslında UseMethod("print") çağrısından oluşmaktadır. UseMethod parametre olarak çağrılacak metodun ismini alır ve x'in türüne dayalı olarak ilgili sınıfın bu metodunun çağrılmasına yol açar. Şimdi biz aşağıdaki gibi generic bir foo fonksiyonu yazalım:

```
foo <- function(x) UseMethod("bar")
```

Burada biz foo(a) gibi bir çağrı yaptığımızda R yorumlayıcısı a'nın hangi sınıf türünden olduğuna bakar. O sınıfa ilişkin "bar" isimli fonksiyonu çağırır. Bir fonksiyonu bir sınıfa ilişkin hale getirmek için onu yukarıda da belirtildiği gibi <fonksiyon ismi>.<sınıf ismi> biçiminde isimlendirmek gerekir. Örneğin:

```
bar.myclass <- function(x)
{
  print("myclass bar function called")
}
```

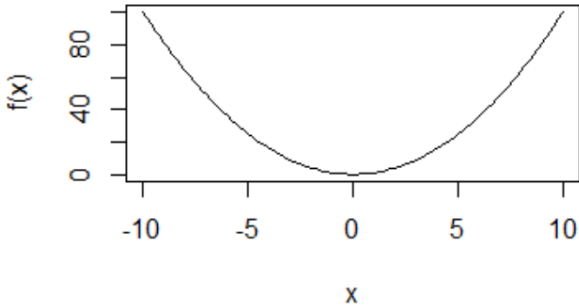
```
> foo(mc)
[1] "myclass bar function called"
```

Örneğin:

```
print.curve <- function(f)
{
  curve(f, -10, 10)
}
```

```
create.curve <- function(f)
{
  class(f) <- "curve"
  f
}

> f <- create.curve(function(x) x^2)
> f
```



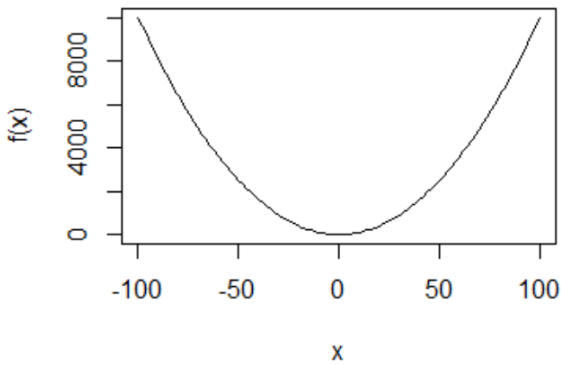
Generic print fonksiyonunun parametrik yapısı şöyledir:

```
print(x, ...)
```

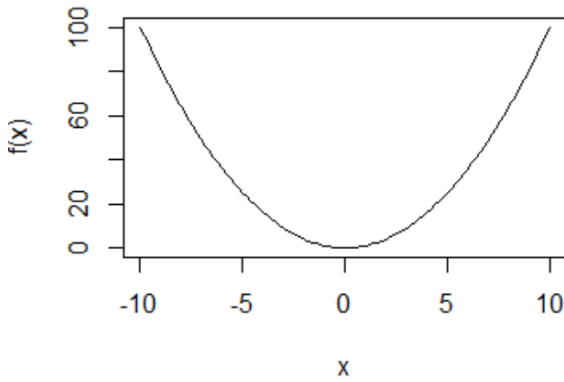
Yani biz print fonksiyonuna istersek başka argümanlar da geçebiliriz. Generic bir fonksiyon buradaki argümanları ilgili sınıfın print fonksiyonuna aynı biçimde geçmektedir. O halde biz print.curve fonksiyonunu şöyle de yazabilirdik:

```
print.curve <- function(f, a = -10, b = 10)
{
  curve(f, a, b)
}

> f <- create.curve(function(x) x^2)
> print(f, -100, 100)
```



Tabii biz komut satırında f değişkenin ismini yazıp ENTER tuşuna bastığımızda komut satırı onu print(f) biçiminde çağıracaktır. Bu durumda a ve b parametresi default olarak -10, 10 değerini alacaktır.



S4 Sınıf Sistemi

S4 R'da sistemi biraz daha güvenli ve formal bir sınıf sistemi sunmaktadır. Bazı R programcıları S3'ü bazıları ise S4'ü tercih etmektedir. Tabii mevcut R yorumlayıcıları her iki sistemi de desteklemektedir.

S4 sisteminde bir sınıf S3'teki gibi nesnenin "class" özniteliği set edilerek değil doğrudan setClass isimli bir fonksiyon ile yaratılmaktadır. setClass fonksiyonunun parametrik yapısı şöyledir:

```
setClass(Class, representation, prototype, contains = character(),
         validity, access, where, version, sealed, package,
         S3methods = FALSE, slots)
```

Fonksiyon sınıf nesnesini yaratan fonksiyona geri dönmektedir. İleride de ele alınacağı gibi aslında sınıf nesneleri new isimli fonksiyonla da yararılabilmektedir.

setClass fonksiyonunun birinci parametresi yarılacak sınıfın ismini belirtir. slots parametresi sınıfın elemanları belirlemek için kullanılmaktadır. Bu parametre isimli bir string (character) vektörü olmak zorundadır. İsim elemanın ismini string ise türünü belirtir. Elemanın türü "character", "numeric", "logical", "list" gibi olabilmektedir. Örneğin:

```
setClass("myclass", slots = c(name = "character", no = "numeric"))
```

prototype parametresi bir çeşit "constructor" görevini görmektedir. Yani nesne yaratımı sırasında ilgili elemanların alacağı default değeri belirlemekte kullanılır. Bu parametre bir liste biçiminde oluşturulmalıdır. Listenin elemanları da elemanların isimleri olmalıdır. Örneğin:

```
setClass("myclass", slots = c(name = "character", no = "numeric"),
         prototype = list(name = "Noname", no = 0))
```

S4 sisteminde bir sınıf türünden nesneler new isimli fonksiyonla yaratılmaktadır. new fonksiyonun parametrik yapısı şöyledir:

```
new(Class, ...)
```

Fonksiyonun birinci parametresi nesnesi yaratılacak sınıfın ismini alır. Diğer parametreler tek tek eleman ismi = değer biçiminde liste tarzı oluşturulur. Örneğin:

```
mc <- new("myclass", name = "Kaan Aslan", no = 123)
```

```
> mc
An object of class "myclass"
Slot "name":
[1] "Kaan Aslan"
```

```
Slot "no":
[1] 123
```


Eğer yaratım sırasında new fonksiyonunda bir elemanı belirtmeseydik protoype listesinde belirtilen değer ona atanacaktı. Örneğin:

```
mc <- new("myclass", name = "Kaan Aslan")
> mc
An object of class "myclass"
Slot "name":
[1] "Kaan Aslan"

Slot "no":
[1] 0
```

Aslında setClass fonksiyonu aynı zamanda yaratıcı fonksiyona geri dönmektedir. Yani biz S4 sınıf nesnesini new yerine setClass fonksiyonunun geri döndürdüğü fonksiyonla da yaratabiliriz. Örneğin:

```
myclass <- setClass("myclass", slots = c(name = "character", no = "numeric"),
  prototype = list(name = "Noname", no = 0))

mc <- myclass(name = "Kaan Aslan", no = 123)

> mc
An object of class "myclass"
Slot "name":
[1] "Kaan Aslan"

Slot "no":
[1] 123
```

S4 sisteminde sınıfın elemanlarına slot denilmektedir. Slotlara erişmek için ise \$ yerine @ operatörü kullanılmaktadır. Örneğin:

```
> mc@name
[1] "Kaan Aslan"
> mc@no
[1] 123
```

S4 sisteminde sınıfa bir fonksiyon eklemek için setMethod isimli fonksiyon kullanılmaktadır. (Halbuki S3 sistemimde fonksiyonun ismi <fonksiyon ismi>.<sınıf ismi> biçiminde verilerek sınıfa ilişkin fonksiyonlar yazılıyordu.) SetMethod fonksiyonunun parametrik yapısı şöyledir:

```
setMethod(f, signature = character(), definition,
  where = topenv(parent.frame()),
  valueClass = NULL, sealed = FALSE)
```

Fonksiyonun birinci parametresi sınıfa eklenecek fonksiyonun ismini, ikinci parametresi ekleme işleminin yapılacağı sınıfın ismini üçüncü parametresi ise eklenecek fonksiyonu belirtir. Örneğin:

```
myclass <- setClass("myclass", slots = c(name = "character", no = "numeric"),
  prototype = list(name = "Noname", no = 0))

mc <- myclass(name = "Kaan Aslan", no = 123)
setMethod("show", "myclass", function(object) { print("myclass show function")})
```

Burada sınıfa eklenmek istenen fonksiyonun ismi "show" biçimindedir. Eklenecek fonksiyonun nesne parametresi object biçiminde isimlendirilmiştir. Bu parametre aslında herhangi bir biçimde isimlendirilebilmektedir. Ancak bu parametreye farklı bir ismin verilmesi uyarı mesajının oluşmasına yol açar.

S4 sınıf sisteminde show generic bir fonksiyondur. Komut satırında bir değişken ismi yazılıp ENTER tuşuna basıldığında eğer değişken S4 sınıf sistemine ilişkin bir sınıf türündense print yerine ilgili sınıfın show fonksiyonu çağrılmaktadır. Örneğin:

```
> mc  
[1] "myclass show function"
```

S4 sisteminde türetme için setClass fonksiyonunun contains parametresi kullanılır. Bu parametreye taban sınıfın (ya da sınıfların) ismi girilirse söz konusu sınıf türemiş sınıf olur. Örneğin:

```
myclass <- setClass("myclass", slots = c(name = "character", no = "numeric"),  
  prototype = list(name = "Noname", no = 0))  
  
yourclass <- setClass("yourclass", slots = c(name = "character", no = "numeric", salary =  
  "numeric"), prototype = list(name = "Noname", no = 0, salary = 0), contains = "myclass")  
  
yc <- yourclass(name = "Kaan Aslan", no = 123, salary = 7000)
```

Burada "yourclass" isimli sınıf "myclass" isimli sınıftan türetilmiştir. Türetme çokbiçimlilik konusunda bazı etkilere sahiptir. İzleyen bölümde bu konu ele alınacaktır.

S4 sisteminde sınıfın fonksiyonu normal yöntemle değil çokbiçimli mekanizmayla çağrılmaktadır. S4 sisteminde generic fonksiyon oluşturmak için setGeneric fonksiyonu kullanılmaktadır. setGeneric fonksiyonunun parametrik yapısı şöyledir:

```
setGeneric(name, def = , group = list(), valueClass = character(),  
  where = , package = , signature = , useAsDefault = ,  
  genericFunction = , simpleInheritanceOnly = )
```

Fonksiyonun birinci parametresi oluşturulacak generic fonksiyonun ismini, ikinci parametresi de onun default tanımlamasını oluşturmaktadır. Bu tanımlama yapılmayabilir. Eğer bu tanımlama yapılırsa çokbiçimli çağırma sırasında ilgili sınıfın ilgili fonksiyonu bulunamazsa buradaki fonksiyon çalıştırılır. Örneğin:

```
setGeneric("foo", function(object) {print("default implementation")})
```

Tabii aslında ikinci parametreyi de belirtmeden fonksiyonu tek parametrelilik olarak da aşağıdaki gibi çağırabiliriz:

```
setGeneric("foo")
```

Bu durumda default fonksiyon yazılmamış olur.

Örneğin:

```
myclass <- setClass("myclass", slots = c(name = "character", no = "numeric"),  
  prototype = list(name = "Noname", no = 0))  
  
setGeneric("foo", function(object) {print("default implementation")})  
setMethod("foo", "myclass", function(object) { print("myclass foo")})  
  
mc <- myclass(name = "Kaan Aslan", no = 123)
```

R'da Genel Yazım Biçimleri (Google's R Style Guide)

R için pek çok programcı ve kitap yazarı birbirlerine benzeyen çeşitli yazın biçimleri kullanmaktadır. Google firmasının "R Style Guide" isimli dokümanı Google R programcıları için düzenlenmiştir. Bu yazım biçiminin anahtar noktaları şunlardır:

- R script dosyalarının uzantıları .R biçiminde olmalıdır.

- Birden fazla sözcük içeren değişken isimlerinde sözcükler arasında '.' karakteri kullanılabilir. Ya da bunlar deve notasyonu ile yazılabilirler. Örneğin:

```
numberOfStudents  
sector.number  
current.working.directory
```

Tabii değişken isimlerinin çok uzun olmaması tavsiye edilmektedir.

- 80 karakterden daha fazla satır uzatılmamalıdır.

- Bloklamada Tab yerine Space karakterleri kullanılır. Tipik olarak girinti için 2 Space karakteri önerilmektedir.

- İki operandlı operatörlerin sol ve sağ yanlarında bir Space boşluk bırakılır.

- if ifadesi ve diğer kontrol ifadelerinde küme parantezleri aynı satırda açılır ve ayrı bir satırda kapatılır. Örneğin:

```
if (condition) {  
    one or more lines  
} else {  
    one or more lines  
}
```

- Mümkün olduğu kadar aynı satıra tek bir ifade yazılmalı ve ';' ayracı kullanılmamalıdır.

- Yorumlama satırları # ve bir Space ile başlatılır.

- Fonksiyon bildiriminde önce default değer almayan parametreler sonra default değer alan parametreler yerleştirilmelidir.

R'in İstatistikte Kullanımına İlişkin Temel Bilgiler

R özellikle veri analizi ve istatistiksel analiz için tercih edilen bir programlama dilidir. Kursumuzun bu bölümünde R'a dayalı olarak temel bir istatistik bilgisi verilecektir. İstatistik ile olasılık konusu iç içedir. Bu nedenle olasılık ve istatistik konuları pek çok yerde bir arada ele alınmaktadır.

İstatistik iki temel bölüme ayrılmaktadır: Betimsel İstatistik (Descriptive Statistics) ve Çıkarımsal İstatistik (Inferential Statistics). Betimsel istatistikte toplanmış olan birtakım veriler çeşitli yöntemlerle betimlenir ve sunulur. Bu istatistiğin amacı olanı göstermek ve açıklamaktır. Halbuki çıkarımsal istatistiğin amacı geleceğe yönelik kestirimlerde bulunmaktır.

Olasılık Nedir?

Olasılığın tam bir tanımını yapmak zordur. Pek çok tanım önerilmiştir. Fakat en kabul gören tanımlardan biri "görelilik" tanımıdır. Bu tanıma göre olasılık bir limit durumudur. Bir olayın n defa tekrarlanması durumunda oluşan orandır. Örneğin bir paranın atılmasında yazı ya da tura gelme olasılığı 0.5'tir. Tura gelme sayısını atış sayısına bölersek ve bu atışları çok fazla yaparsak oran git gide 0.5'e yakınsayacaktır.

$$\lim_{n \rightarrow \infty} \frac{\text{Olus Sayısı}}{n}$$

Bu görelilik tanımını R'da sınavalım:

Örneğin parayı 100 kere atmış olalım:

```
> table(sample(c("Y", "T"), 100, replace = T))/100
```

```
      T      Y  
0.51 0.49
```

Şimdi 1000000 tane atalım:

```
> table(sample(c("Y", "T"), 1000000, replace = T))/1000000
```

```
      T      Y  
0.498882 0.501118
```

Şimdi 100000000 kere atalım:

```
table(sample(c("Y", "T"), 100000000, replace = T))/100000000
```

```
      T      Y  
0.5001363 0.4998637
```

Rastgele olaylara rassal olaylar (random event) denilmektedir. Bir rassal olaydaki mümkün tüm sonuçların kümesine evrensel küme denir. Örneğin para atımında $E = \{Y, T\}$ biçimindedir. Zar atımında ise $E = \{1, 2, 3, 4, 5, 6\}$ biçimindedir. Olasılık teorisinde evrensel kümenin her bir alt kümesine "olay (event)" denilmektedir. Evrensel kümenin tek elemanlı alt kümelerine ise "basit olay (simple event)" denilmektedir. Eğer evrensel kümedeki her öğenin ortaya çıkma şansı aynı ise belli bir olayın olasılığı da $n(O) / n(E)$ biçimindedir. Örneğin zar atışında "3 veya 5 gelme olasılığı" bu durumda $2 / 6 = 1 / 3$ 'tür.

Rassal Değişken (Random Variable) Kavramı

İstatistik için en önemli kavramlardan biri de rassal değişken (random variable) kavramıdır. Rassal değişken evrensel kümenin her bir basit olayını bir gerçek sayı ile eşleyen bir fonksiyondur. Örneğin bir parayı aynı anda atalım. Yazı için 0, tura için 1 değerini kabul edelim. Bu atıştan elde edilen toplam değer bir rassal değişkendir. Rassal değişken rastgele olaylardan elde edilen sayısal bir sonuçtur.

$YT = 1$

$YY = 0$

$TY = 1$

$TT = 2$

Rassal değişkenler olasılıkta büyük harflerle gösterilirler. Biz de bu rassal değişkene X diyelim. Pekiyi X 'in 1 olma olasılığı nedir?

$P(X = 1) = 1 / 2$

Bazı rassal değişkenlere şöyle örnekler verilebilir:

- Belli bir saatte belli bir noktadan geçen araçların sayısı. (Burada evrensel küme 0 ile toplam kapasite kadardır.)
- Rastgele çekilen birisinin boyu
- Bir günde üretilen bozuk ürünlerin sayısı

Rassal değişken olayları sayıya indirgiyerek düşünmemize yaramaktadır.

Rassal değişkenler kesikli (discrete) ve sürekli (continuous) olmak üzere ikiye ayrılmaktadır. Eğer bir rassal değişken her gerçek sayı değerini alamıyor, yalnızca belirli değerleri alabiliyorsa ona kesikli rassal değişken, eğer her gerçek değeri alabiliyorsa ona da sürekli rassal değişken denir. Örneğin:

- Atılan zarın üzerindeki sayı (kesikli)
- Bir topluluktan rastgele seçilen bir insanın boyu (sürekli)
- Fabrikada üretilen şekerlerin ağırlığı (sürekli)
- Fabrikada üretilen hatalı ürün sayısı (kesikli)
- Kavşaktan belli bir saat diliminde geçen araçların sayısı
- ...

Bir rassal değişkenin belli bir değeri alma olasılığı söz konusu olabilir. Bu olasılık X rassal değişken olmak üzere $P(X = N)$ ile gösterilir. Örneğin kavşaktan bir saat içerisinde geçen araç sayısı X isimli rassal değişkenle gösteriliyor olsun. $P(X = 56)$ ifadesi kavşaktan bir saat içerisinde 56 araç geçme olasılığını belirtmektedir.

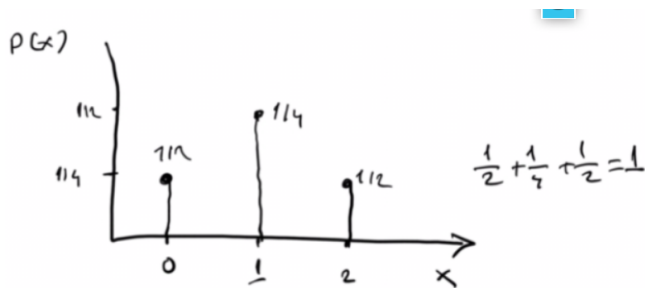
Kesikli rassal değişkenlerin nokta değer olasılıkları 0 ile 1 arasında bir değer olarak karşımıza çıkar. Örneğin iki zar aynı anda atıldığında yüzlerdeki sayılar toplamına Y rassal diyelim. $P(Y = 3)$, $P(Y = 4)$ gibi değerlerin 0 ile 1 arasında belli olasılıkları vardır. Ancak sürekli sürekli rassal değişkenlerin nokta olasılıkları her zaman sıfırdır. Örneğin bir boncuk fabrikasında günde 1000000 boncuk üretiliyor olsun. Rastgele seçilen bir boncuğun ağırlığına X rassal değişkeni diyelim. $P(X = 1.2 \text{ gram})$ olma olasılığı sıfırdır. Çünkü 1.2 değeri aslında sonsuz sayıdaki gerçek değerden biridir. Belli bir değer için sonsuz içerisindeki olasılığı sıfırdır. İşte sürekli, rassal değişkenlerin nokta olasılıkları sıfırdır fakat aralık olasılıkları 0'dan büyük olabilir. Örneğin boncuk fabrikası örneği için $P(1.2 < X < 1.3)$ olasılığı sıfır olmak zorunda değildir.

Olasılık Fonksiyonları ve Olasılık Yoğunluk Fonksiyonları

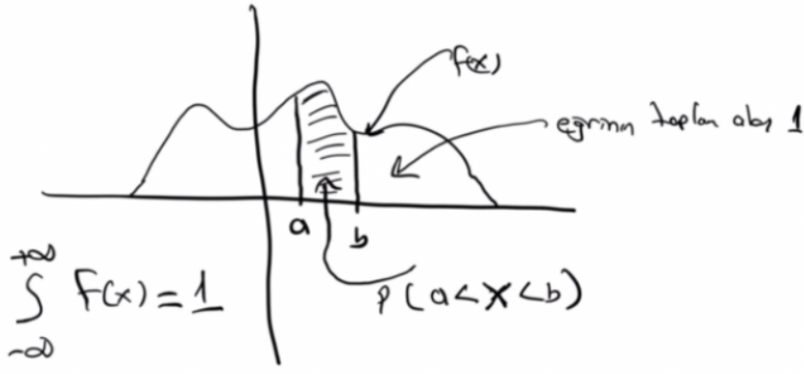
Bir kesikli rassal değişkenin belli bir değerini onun olasılığına eşleyen fonksiyona olasılık fonksiyonu (probability function) denilmektedir. Olasılık fonksiyonlarının grafikleri çizilebilir. Olasılık fonksiyonlarının toplam y değerleri (yani olasılık değerleri) 1 olmak zorundadır. Örneğin Yazı için 0 Tura için 1 değerinin karşı getirildiğini düşünelim. İki parayı aynı anda atalım. Bu olaydaki evrensel küme şöyledir:

YY
YT
TY
TT

Şimdi gelen paraların sayısal karşılıklarına ilişkin bir X rassal değişkeni olsun. Bu X rassal değişkeninin olasılık fonksiyonunu çizelim:



Sürekli rassal değişkenlerin nokta değerleri sıfır olduğuna göre onların olasılık yoğunluk fonksiyonları nasıl olabilir? İşte olasılık yoğunluk fonksiyonları öyle fonksiyonlardır ki o fonksiyonların eğri altında kalan alanı 1'dir. İki nokta arasında eğri altında kalan alan ise o rassal değişkenin iki nokta arasında olma olasılığını vermektedir. Örneğin:

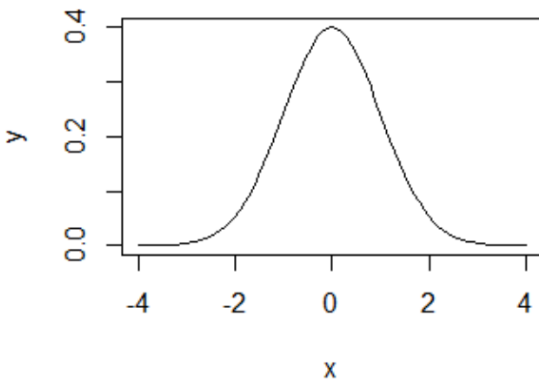


Şekildeki $f(x)$ fonksiyonunun olasılık yoğunluk fonksiyonu olabilmesi için eğri altında kalan alanın toplamının 1 olması gerekir. Bu koşullar altında artık biz X rassal değişkeninin a ile b arasında değer alma olasılığını integral hesapla bulabiliriz:

$$\int_a^b f(x)$$

Peki sürekli bir rassal değişkenin olasılık yoğunluk fonksiyonu nasıl elde edilmektedir? Olasılık yoğunluk fonksiyonun eğrisel biçimi aslında bir histogramla hemen görsel olarak anlaşılabilir. Şöyle ki: Histogram zaten belli aralıklardaki sıklıklara göre çizilen bir grafik. Sıklıklar da aslında olasılıkları belirtir. O halde kaba olsa da histogram bize olasılık yoğunluk fonksiyonun biçimi hakkında hemen bilgi verebilir. Tabii histogram yalnızca kaba bir bilgi vermektedir. İlgili rassal değişkeninin olasılık yoğunluk fonksiyonunu tam olarak veremez. O halde olasılık yoğunluk fonksiyonunu elde edebilmek için şöyle bir yöntem izlenebilir: Rassal sistemden rastgele değerler çekilip göreceli sıklıklar elde edilip limit işlemleriyle olasılık yoğunluk fonksiyonları oluşturulabilir.

Peki bir rassal sistemin olasılık fonksiyonu ya da olasılık yoğunluk fonksiyonu tamamen birbirlerinden bağımsız ve ayrı mıdır? İşte pek çok rassal sistemin olasılık fonksiyonları ya da olasılık yoğunluk fonksiyonları aslında birbirlerine benzemektedir. Yani belli modeller vardır. Sistem bu modellerden birine oldukça uyma eğilimindedir. Örneğin insanı pek çok sürekli rassal değişkenlerle ifade edilebilecek özellik normal dağılım denilen olasılık yoğunluk fonksiyonuna uymaktadır. Normal dağılım eğrisi aşağıdaki benzerdir.



Bu olasılık yoğunluk fonksiyonundan görsel olarak şunları hemen anlayabiliriz: Normal dağılımda ortalama etrafında toplanma olasılığı oldukça yüksektir. Ortalamadan uzaklaştıkça bu olasılık düşmektedir. Örneğin kilo için bu eğriyi yorumlayalım. İnsanların ortalama kilosunun 60 olduğunu varsayalım. Rastgele birisinin 50 ile 70 arasında olma olasılığı ile 100 ile 120 arasında olma olasılığını karşılaştırdığımızda 50 ile 70 arasında olma olasılığının çok daha yüksek olduğu görülecektir.

Olasılık yoğunluk fonksiyonları için iki parametre söz konusudur: Ortalama (mean) ve standart sapma. Olasılık yoğunluk fonksiyonunun ortalaması tam olarak eğri altında kalan alanı ortalamaya değerdir. Standart sapma

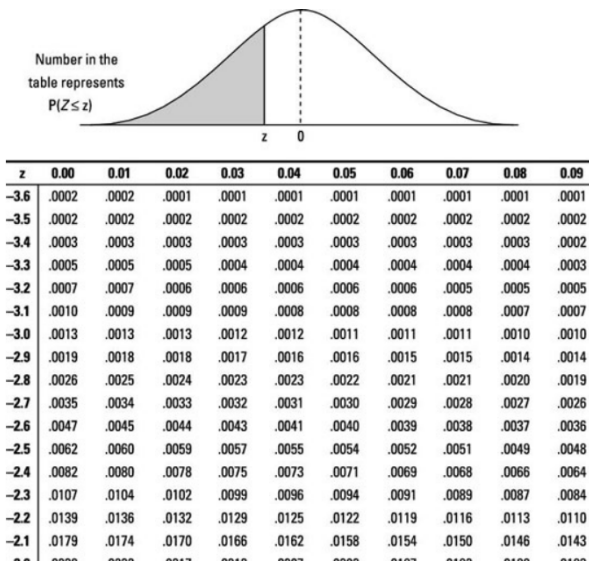
ortalamadna uzaklığın ölçüsüdür. Normal dağılımda standart sapma çan eğrisinin yayılmasıyla karakterizedir. Standart sapma azaldıkça normal eğrisi kapanır ve ortalama etrafında olma eğilimi artar.

Sürekli ve kesikli pek çok dağılım modellenmiştir. Bu dağılımlar çeşitli olguları açıklayabilmektedir. Ancak şüphesiz normal dağılım en önemli sürekli dağılımdır. Doğadaki rassal olayların çoğu bu dağılıma uymaktadır. Bir olguyu incelerken onun normal dağılıma uyup uymadığı çeşitli hipotez testleriyle test edilebilmektedir. (Örneğin Kolomogorov-Simirnov testi gibi.)

Ortalaması 0 olan standart sapması 1 olan normal dağılıma "standart normal dağılım" ya da Z dağılımı denilmektedir. Ortalaması ve standart sapması farklı olan normal dağılım değerleri standart normal dağılım değerlerine dönüştürülebilirler. Dönüşüm şöyle yapılmaktadır:

$$Z = \frac{X - \mu}{\sigma}$$

Standart normal dağılıma ilişkin Z tabloları genellikle kümülatif biçimde oluşturulmuşlardır. Bu tablolarda belli bir X değerine karşılık onun eksi sonsuzdan X'e kadar olasılık değeri verilmektedir. Örnek bir Z tablosu şöyledir:



Burada verilen bir z değerinin tablodaki karşılığı $P\{Z < z\}$ değerine eşittir. Başka deyişle tablodaki değerler bize eksi sonsuzdan bir değere kadar eğri altında kalan alanı vermektedir. Peki bu tablo nasıl kullanılır?

Örneğin 10000 kişinin bulunduğu bir ortamda bu bin kişinin en uzak ne kadar atlacağı belirlenmiş olsun. Sonra bu atlanabilen mesafenin normal dağılmış olduğunu belirlemiş olalım. Örneğin bu normal dağılımın ortalaması 153 santim standart sapması 32 santim olsun. Şimdi biz bu grupta rastgele çektiğimiz kişinin 200 santimin üzerinde atlama olasılığını hesaplayabiliriz.



Eğer yukarıdaki gibi bir Z tablosu kullanılacaksa bu Z tablosu standart normal dağılım için oluşturulmuş olduğundan bu 200 değerinin standart normal dağılım dönüştürülmesi gerekir. Dönüştürme şöyle yapılacaktır:

$$Z_s = (200 - 153) / 32 = 1.46$$

Z tablosundan bu değere baktığımızda 0.9279 olduğunu görürüz. Bu değer 1'den çıkartılırsa 0.072 bulunur. Bu durumda bu ana kütleden rastgele çekilen birisinin iki metrenin yukarısında atlama olasılığı %7 civarındadır.

R'da Normal Dağılıma İlişkin Temel Fonksiyonlar

R'da normal dağılıma ilişkin temel fonksiyonlar şunlardır:

- rnorm fonksiyonu ortalaması ve standart sapması verilen bir normal dağılımda rastgele değerleri bize verir. Fonksiyonun parametrik yapısı şöyledir:

```
rnorm(n, mean = 0, sd = 1)
```

rnorm fonksiyonunun birinci parametresi kaç tane rassal sayı üretileceğini belirtir. İkinci ve üçüncü parametreleri normal dağılımın ortalama ve standart sapmasını belirtmektedir. Örneğin:

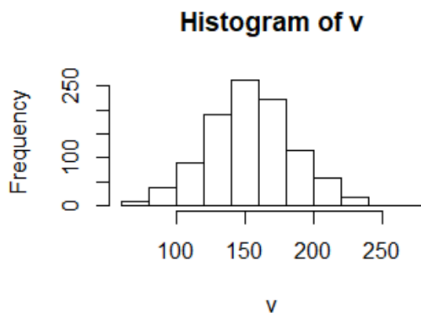
```
> rnorm(10)
[1] -0.61947978 -0.34280247 -0.28074721 -0.66503204  0.81925418  0.14025438
[7]  0.93680252  0.74977246  1.51502317 -0.04989758
```

Yukarıdaki örnekteki durum için rastgele 10 kişi çekelim:

```
> rnorm(10, 153, 32)
[1] 161.4567 139.0747 140.1961 162.4547 150.7115 207.7103 145.1817 123.5385
[9] 182.5739 108.2170
```

Şimdi örnekteki gruptan 1000 kişi çekip onun histogramını çizelim:

```
> v <- rnorm(1000, 153, 32)
> hist(v)
```



pnorm fonksiyonu eksi sonsuzdan x'e kadar eğri altında kalan alanı bize vermektedir. Bu alan da zaten Z tablosunda yazılan değerdir. Fonksiyonun parametrik yapısı şöyledir:

```
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
```

Fonksiyonun birinci parametresi eksi sonsuzdan itibaren alanı hesaplanacak x değeridir. Sonraki iki parametre ortalama ve standart sapmadır. Bu fonksiyon bize $P\{X < q\}$ değerini vermektedir. Örneğin yukarıdaki örnekte rastgele seçilen birisinin 180 santimden daha az atlayabilme olasılığı $P\{X \leq 180\}$ 'dir ve pnorm fonksiyonuyla şöyle bulunur:

```
> pnorm(180, 153, 32)
[1] 0.8005954
```

O halde rastgele seçilen birisinin 180 santimi geçebilme olasılığı da şöyledir:


```
> 1 - pnorm(180, 153, 32)
[1] 0.1994046
```

Aslında fonksiyonun dördüncü parametresi default TRUE biçimdedir. Bu parametre FALSE yapılırsa fonksiyon $P\{X > q\}$ değerini verir.

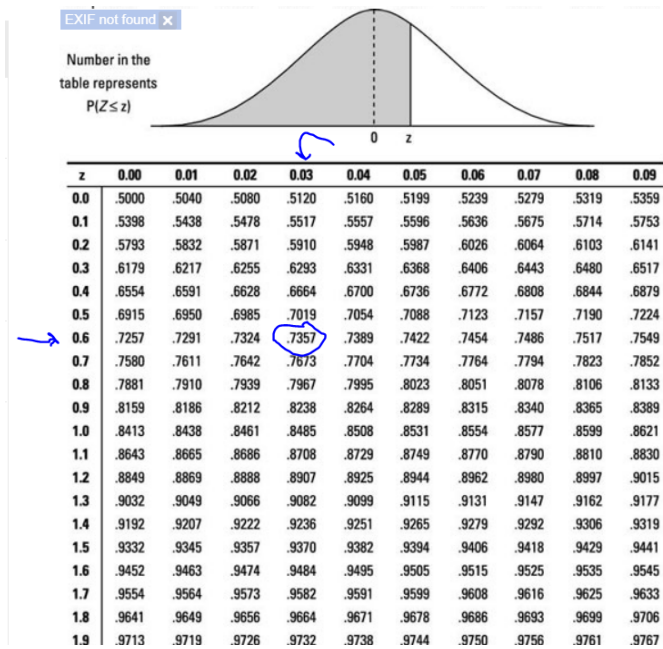
pnorm fonksiyonunun tersini yapan bir qnorm fonksiyonu vardır. qnorm bizden olasılık değerini alır ve $P\{X \leq q\}$ 'daki q değerini verir. Örneğin biz standart normal dağılımda 0.95 olasılığa karşı gelen z değerini (x değeri de diyebiliriz) bulabiliriz. Ya da yukarıdaki örneğimizde %90 olasılık kişilerin atlayabileceği uzaklık qnorm ile bulunur. Fonksiyonun parametrik yapısı şöyledir:

```
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
```

Fonksiyonun birinci parametresi olasılık değerini ikinci parametre ortalaması ve üçüncü parametresi standart sapmayı belirtir. Örneğin:

```
> qnorm(0.7357)
[1] 0.6301446
```

Bu değer z tablosundaki aşağıdaki değerdir:

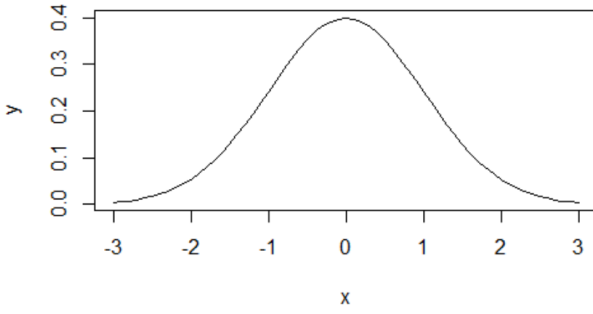


Örneğin:

```
> p <- pnorm(0.95)
> qnorm(p)
[1] 0.95
```

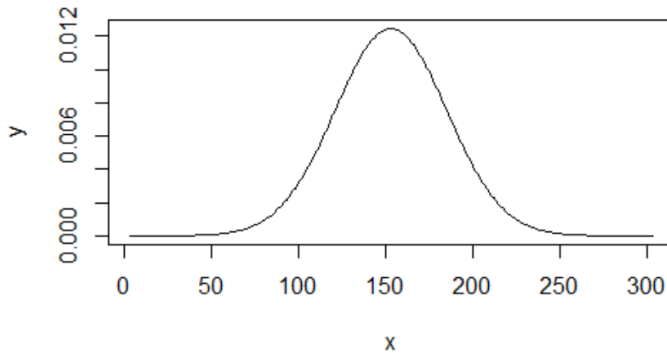
dnorm fonksiyonu ise bizden x değerini alır, Gauss eğrisindeki y değerini bize verir. Böylece biz normal dağılım eğrisinin grafiğini çizebiliriz. Örneğin:

```
> x <- seq(-3, 3, 0.1)
> y <- dnorm(x)
> plot(x, y, t = 'l')
```



Şimdi de bizim örneğimizdeki normal dağılım eğrisini çizelim:

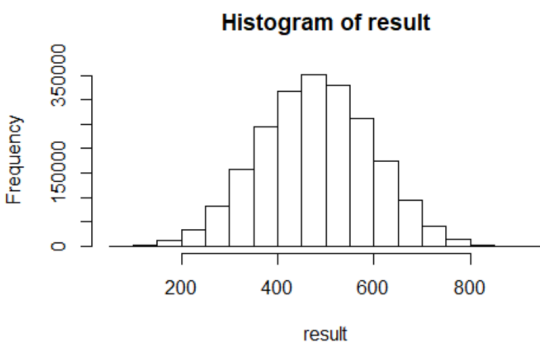
```
> x <- seq(153 - 150, 153 + 150, 1)
> y <- dnorm(x, 153, 32)
> plot(x, y, t = 'l')
```



Örneklem Dağılımları

Merkezi limit teoremi sonuç çıkarıcı istatistiğin en önemli teoremlerinden biridir. Merkezi limit teoremine göre bir ana kütleden çekilen n'li tüm örneklerinin ortalaması normal dağılır. Eğer ana kütle normal dağılmışsa n değeri ne kadar küçük olursa olsun ortalamaları yine normal dağılır. Eğer anak kütle dağılımı normal değilse n değeri yükseldikçe örneklem dağılımı normale yaklaşır $n \geq 30$ gibi bir değer için normal dağılıma çok yaklaşılmıştır. Bunu basit olarak şöyle ispatlayabiliriz. Ana kütle düzgün dağılmış olsun ve biz ondan n'li örnekler alıp onların ortalamalarını bulalım ve histogramını çizelim.

```
> population <- sample(1:1000, 50)
> result <- combn(population, 5, mean)
> hist(result)
```



Bu örnekte biz 1 ile 1000 arasında düzgün dağılmış rastgele 50 değer çektik. (Aynı işlemi runif fonksiyonuyla da yapabiliriz). Sonra bu ana kütlenin 5'li alt kümelerinin ortalamalarını bulduk. Merkezi limit teoremi bu ortalamaların normal dağılacığını söylemektedir. Histogramdan görülmektedir.

Merkezi limit teoreminin en önemli diğer sonucu da örneklem ortalamalarına ilişkin dağılımın ortalama ve standart sapması ile ana kütle ortalama ve standart sapması arasındaki ilişkidir.

Merkezi limit teoremine göre örneklem ortalamalarının dağılımlarının ortalaması ana kütle ortalaması ile aynı örneklem ortalamalarının standart sapması aşağıdaki formülde olduğu gibidir.

$$\mu_{\bar{x}} = \mu$$
$$\sigma_{\bar{x}} = \frac{\sigma}{\sqrt{n}} \times \sqrt{\frac{N-n}{N-1}}$$

Ancak ana kütlelerin eleman sayısı çok fazla (örneğin teorik sonsuz) standart sapmadaki ikinci çarpım tamamen atılabilir. Genel olarak örnek ana kütle büyüklüğü örneklem büyüklüğünün en az 20 katıysa bu atılabilir. Bu durumda örneklem standart sapması şöyle olur:

$$\sigma_{\bar{x}} = \sigma / \sqrt{n}$$

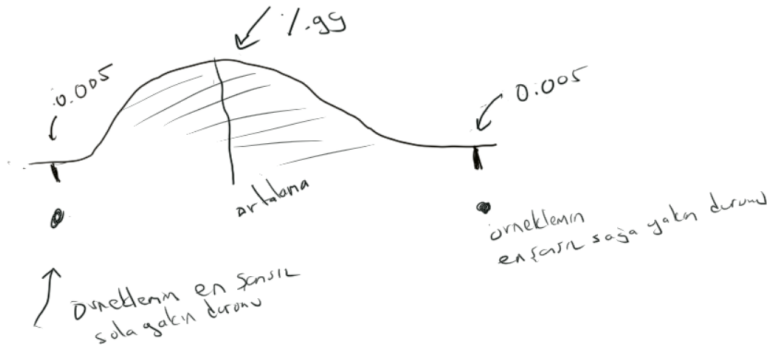
Bu eşitliklere örneğin bir anakütle içerisinde biz 100'lük tüm örneklerin ortalamalarının dağılımı bulsak onun normal dağıldığını görürüz. O dağılımın ortalaması anakütle ortalamasına eşittir. Standart sapmaları arasındaki ilişki de yukarıda belirtildiği gibidir. Şimdi ortalama testinin yukarıdaki gibi olduğunu R'da ispatlayalım:

```
> population <- sample(1:1000, 50)
> result <- combn(population, 5, mean)
> mean(population)
[1] 538.96
> mean(result)
[1] 538.96
```

Buradan da görüldüğü gibi örneklem ortalamalarının ortalaması ana kütle ortalamasına eşittir.

Örneklem İçin Güven Aralıklarının Oluşturulması ve Z Testi

İstatistikteki en önemli hipotez testlerinden birisi bir ana kütlede tek bir örnek çekip onun ortalamasına bakarak ana kütle ortalamasının tahmin edilmesidir. Böylece belli bir yanlış payıyla ana kütle ortalamasının hangi değerler arasında olabileceği hesaplanabilmektedir. Bu aralığa güven aralığı (confidence interval) denilmektedir. Örneğin bir işletmede üretilen vidaların ortalama çapını bulmak isteyelim. İşletmede neredeyse sonsuz tane vida üretilmektedir. İşte biz üretilen vidalardan 100 tanesine bakarak tüm vidaların ortalamalarının hangi aralıkta olabileceğini tahmin edebiliriz. Tahmin şu mantıkla yapılmaktadır. Ana kütledeki bütün 100'lük örneklerin ortalamaları normal dağılmıştır. Bu öyle bir normal dağılımdır ki bu örneklem dağılımın ortalaması ana kütle ortalamasına eşittir. Standart sapması ise ana kütle standart sapmasının örneklem büyüklüğünün kareköküne bölümü kadardır. İşte buradan hareketle biz aldığımız 100'lük örneğin ortalamasına bakarak ana kütle ortalamasının belli bir güvenlik derecesinde hangi aralıkta olduğunu tahmin edebiliriz. Örneğin bu güvenlik derecesinin %99 olarak alındığını varsayalım:



Bu vida örneğinde önemli bir nokta vardır: Biz fabrikada üretilen vidalardan rastgele 100 tanesini seçip onun ortalamasına baktık. Ancak 1002lük örneklemelerin dağılımını tespit edebilmemiz için ana kütle standart sapmasını bilmemiz gerekir. İşte eğer ana kütle standart sapması bilinmiyorsa örnek eğer 30'dan büyükse aldığımız örneğin standart sapması ana kütle standart sapmasına çok yaklaşıktır. (Burada bu teorem açıklanmayacaktır.) Bu durumda biz ana kütle standart sapması yerine doğrudan örneğin standart sapmasını alabiliriz.

O halde somut örnekle R'da ana kütle ortalaması için güven aralıklarını bulalım.

Örneğin fabrikada üretilen vidaların çap ortalamasını belli bir güven aralığında belirlemek isteyelim. Bunun için 100 tane vidadan oluşan bir örnek çekelim. Bu örneğe bakarak güven aralıklarını oluşturalım. Çekilen 100'lük örneğin ortalamasının 20 mm, standart sapması da 1 mm. olsun. Şimdi biz %99 güven aralığında tüm vidaların hepsinin ortalamasını tahmin etmeye çalışalım. Sorumuzda örnek 30'dan büyük olduğu için ana kütle standart sapması yerine örnek standart sapmasını alabiliriz. Bu durumda örneklem dağılımı ortalaması 20 mm ve standart sapması 3 mm olan bir normal dağılım olacaktır. Fakat biz aslında ana kütle ortalamasını bilmiyoruz ve onu bulmaya çalışıyoruz. İşte bizim bulmamız gereken aralık değerleri şunlardır: "Ortalaması 20 mm. standart sapması 1 mm olan bir normal dağılım eğrisinin %99.5 sağındaki ve %0.05 solundaki değerler nelerdir? Bu değerleri qnorm fonksiyonuyla elde edebiliriz:

```
> qnorm(0.005, 20, 1)
[1] 17.42417
> qnorm(0.995, 20, 1)
[1] 22.57583
```

Yukarıdaki işlemi tek hamlede BSDA paketindeki x.test fonksiyonuyla yapabiliriz. Bunun için önce BSDA paketinin yüklenmesi gerekir. Örneğin elimizde bir örneklem ve anakütle standart sapması bulunuyor olsun. Ana kütle ortalaması için güven aralıklarını z.test fonksiyonuyla şöyle elde edebiliriz:

```
> x <- c(7.8, 6.6, 6.5, 7.4, 7.3, 7., 6.4, 7.1, 6.7, 7.6, 6.8)
> z.test(x, sigma.x = 0.5, conf.level = 0.95)
```

One-sample z-Test

```
data: x
z = 46.553, p-value < 2.2e-16
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 6.722706 7.313658
sample estimates:
mean of x
 7.018182
```

Bu z.test fonksiyonu anakütle ortalaması için hipotez testi de yapabilmektedir. Ayrıca iki örneklemden hareketle onların ana kütlelerinin ortalamaları arasında far olup olmadığını da z.test fonksiyonuyla elde edebiliriz. Konun ayrıntıları uygulama kursunda ele alınacaktır.

Ana Kütle Ortalamaları İçin T Testi

Z testi ile T testi birbirlerine çeşitli yönlerden benzerdir. Eğer örneklem miktarı 30'un aşağısında ise ya da ana kütle standart sapması bilinmiyorsa Z testi yerine T testi daha uygundur. T dağılımı serbestlik derecelerine göre özelleştirilmiş normal dağılıma benzemektedir. t dağılımı için örneklemden hareketle güven aralıkları oluşturan ana kütle ortalamasını test eden iki örneklemden hareketle onların ana kütle ortalamalarını karşılaştıran t.test isimli bir fonksiyon temel R paketlerinde hazır olarak bulunmaktadır. Fonksiyonun parametrik yapısı şöyledir:

```
t.test(x, y = NULL,  
       alternative = c("two.sided", "less", "greater"),  
       mu = 0, paired = FALSE, var.equal = FALSE,  
       conf.level = 0.95, ...)
```

Örneğin bir örneklem için güven aralığı (yani ana kütle ortalamasının hangi aralıkta olabileceği) şöyle yapılabilir:

```
> t.test(x, conf.level = 0.95)
```

One Sample t-test

```
data: x  
t = 50.126, df = 10, p-value = 2.413e-13  
alternative hypothesis: true mean is not equal to 0  
95 percent confidence interval:  
 6.706216 7.330148  
sample estimates:  
mean of x  
 7.018182
```

t.test fonksiyonu ana kütle ortalamasının belli bir olasılıkla belli bir değerde olup olmadığını test etmekte kullanılabilmektedir. Örneğin aşağıdaki gibi bir soru söz konusu olsun:

Toplumda bireylerin hemoglobin düzeyleri normal dağılım göstermekte ve ortalaması $m = 14.2$ mgr/dl dir. y bölgesinde yaşayan bireylerden rasgele seçilen 10 kişilik bir örnekte hemoglobin düzeyleri aşağıdaki gibidir. y bölgesinde yaşayan bireylerin hg düzeyleri farklı mıdır?

14,0	15,4	16,0	13,7	14,2	15,2	15,8	15,8	15,0	14,0
------	------	------	------	------	------	------	------	------	------

Burada y bölgesinde yaşayan 10 kişinin hemoglobin düzeylerine bakılmıştır. Bunun toplum ortalaması olan 14.2'den farklı olup olmadığı bulunmak istenmektedir:

```
> hem = c(14, 15.4, 13.7, 14.2, 15.2, 15.8, 15.8, 15.0, 14.0)  
> t.test(hem, mu = 14.2, conf.level = 0.95)
```

One Sample t-test

```
data: hem  
t = 2.1486, df = 8, p-value = 0.06392  
alternative hypothesis: true mean is not equal to 14.2  
95 percent confidence interval:  
 14.15684 15.42093  
sample estimates:  
mean of x  
 14.78889
```

Buradan 14.2 değerinin %95 olasılıkla güven aralığı içerisinde kaldığı görülmektedir. Yani bu kişilerin hemoglobin düzeyleri genel ortalamadan anlamlı biçimde farklı değildir.

t.test fonksiyonu iki örneklemden hareketle ana kütle ortalamaları arasında fark olup olmadığına yönelik hipotez testleri yapılabilir.

Örneğin şöyle bir sorunun yanıtı için t testi kullanılabilir:

Bir turizm işletmecisi, genel müdürü olduğu otel zincirine bağlı otellerde bir yıl boyunca yapılan promosyon çalışmalarının, bir yıl önceki sayılara oranla müşteri sayısında artış sağladığını iddia etmektedir. Rastlantısal olarak seçilen yedi otelin, 2002 ve 2003 yıllarındaki Temmuz ayı müşteri sayıları aşağıda aşağıdaki tablodadır. $\alpha = 0,01$ için müdürün iddiasını destekleyecek kanıt var mıdır? Sonucu yorumlayın.

Oteller	Temmuz 2002	Temmuz 2003
1	300	320
2	280	290
3	305	290
4	350	375
5	400	415
6	190	185
7	340	360

```
> tem2002 <- c(300, 280, 305, 350, 400, 190, 240)
> tem2002 <- c(300, 280, 305, 350, 400, 190, 340)
> tem2003 <- c(320, 290, 290, 375, 415, 185, 360)
> t.test(tem2002, tem2003, "greate", conf.level = 0.99)
```

Welch Two Sample t-test

```
data: tem2002 and tem2003
t = -0.26525, df = 11.806, p-value = 0.6023
alternative hypothesis: true difference in means is greater than 0
99 percent confidence interval:
 -111.3226      Inf
sample estimates:
mean of x mean of y
 309.2857  319.2857
```

Buradan 2003 ve 2004 yıllarına ilişkin müşteri sayıları arasında anlamlı bir fark olduğu görülmektedir.

Korelasyon Analizi

Korelasyon iki değişken arasında bir ilişkinin olup olmadığı hakkında bilgi veren bir yöntemdir. İki değişkenden elde edilen değerlerden hareketle bir korelasyon değeri hesaplanır. Bu değer 1'e ya da -1'e yaklaştıkça iki değişken arasında bir ilişki olduğu anlaşılır. İlişki düz yönü ya da ters yönlü olabilir. Rastgele değerler arasında bir korelasyon ilişkisi yoktur. Korelasyon ilişkisi basit bir saçılma grafiği ile görsel olarak da temel düzeyde anlaşılabilir. Tabii iki değişken arasındaki korelasyon katsayısı bir neden sonuç ilişkisini açıklamaz. Yalnızca bir ilişki olduğunu açıklar. Örneğin yenilen dondurma miktarı ile denizde boğulma arasında bir ilişki söz konusu olabilir. Ancak bu durum dondurmanın boğulmaya neden olduğu biçiminde anlaşılmamalıdır. Korelasyon katsayısı $[-1, 1]$ arasında bir değerdir. Tipik olarak ilişki düzeyleri şöyle yorumlanmaktadır:

0		İlişki Yok
0.021	0.29	Düşük Düzeyde İlişki
0.30	0.70	Orta Düzeyde İlişki
0.71	0.99	Yüksek Düzeyde İlişki
1.0		Tam ilişki

Korelasyon katsayısı R'da cor fonksiyonuyla elde edilmektedir. Örneğin:

Subject	A	B	C	D	E	F	G	H	I
Normal	56	56	65	65	50	25	87	44	35
Hypervent	87	91	85	91	75	28	122	66	58

Buradaki iki veri kümesi arasındaki korelasyon katsayısını hesaplayalım. cor fonksiyonunun parametrik yapısı şöyledir:

```
cor(x, y = NULL, use = "everything",
    method = c("pearson", "kendall", "spearman"))
```

Şimdi yukarıdaki veriler için korelasyon katsayısını hesaplayalım:

```
> normal <- c(56, 56, 65, 65, 50, 25, 87, 44, 35)
> hypervent <- c(87, 91, 85, 91, 75, 28, 122, 66, 58)
> cor(normal, hypervent)
[1] 0.9661943
```

Doğrusal Regresyon

Doğrusal regresyon iki değişken arasındaki ilişkiyi açıklayacak bir doğru denklemin oluşturulması ile ilgilidir. Genellikle geleceğe yönelik kestirim yapmak amacıyla kullanılır. Tipik olarak bir grup x ve y değer çiftleri vardır. Buradan $y = mx + n$ gibi bir doğru geçirilir. Bu doğruya dayanılarak da x'ler muhtemel y değerleri hesaplanır. Örnek bir soru şöyle olabilir. Amerikada'da bir şehirdeki evin metre kare miktarı ile onun satış fiyatı için aşağıdaki veriler toplanmıştır:

House Price in \$1000s (Y)	Square Feet (X)
245	1400
312	1600
279	1700
308	1875
199	1100
219	1550
405	2350
324	2450
319	1425
255	1700

Bu durumda 1200 metre kare evin tahmini fiyatı nedir?

Doğrusal regresyon R'da lm fonksiyonuyla yapılmaktadır. Fonksiyonun parametrik yapısı şöyledir:

```
lm(formula, data, subset, weights, na.action,
    method = "qr", model = TRUE, x = FALSE, y = FALSE, qr = TRUE,
    singular.ok = TRUE, contrasts = NULL, offset, ...)
```

Fonksiyon en basit olarak ~ operatörü ile yalnızca birinci parametre girilerek çağırılır. Örneğin:

```
lm (y ~ x)
```

Burada x ve y değerlerinden oluşan çiftler için doğrusal regresyon hesaplanmaktadır. Örneğin:

```
> price <- c(245000, 312000, 279000, 308000, 199000, 219000, 405000, 324000, 319000, 255000)
> area <- c(1400, 1600, 1700, 1875, 1100, 1550, 2350, 2450, 1425, 1700)
> lm(price ~ area)
```

```
Call:
lm(formula = price ~ area)
```

```
Coefficients:
(Intercept)      area
  98248.3      109.8
```

Burada doğru denklemi şöyle verilmiştir:

$price = 109.8 \cdot area + 98248.3$

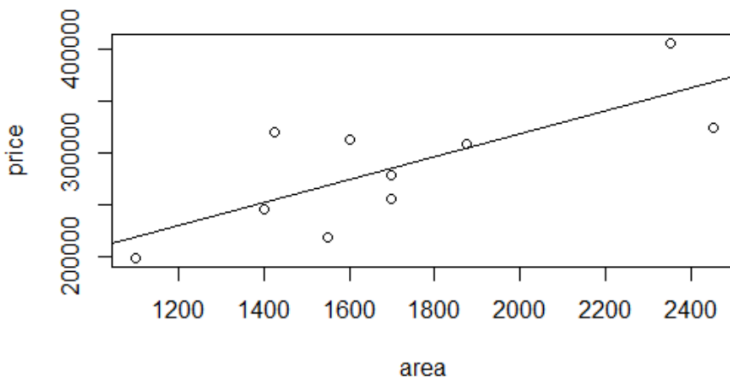
Bu durumda 1200 metrekare için ortalama fiyat:

```
> 109.8 * 1200 + 98248.3
[1] 230008.3
```

lm fonksiyonu aslında bize geri dönüş değeri olarak "lm" isimli bir sınıf nesnesi vermektedir. Bu sınıfın elemanları da regresyon hakkındaki değerleri içerir.

Şimdi de buradaki noktaların ve regresyon doğrusunun grafiğini çizelim:

```
> plot(area, price)
> ? abline
> abline(98248.3, 109.76)
```



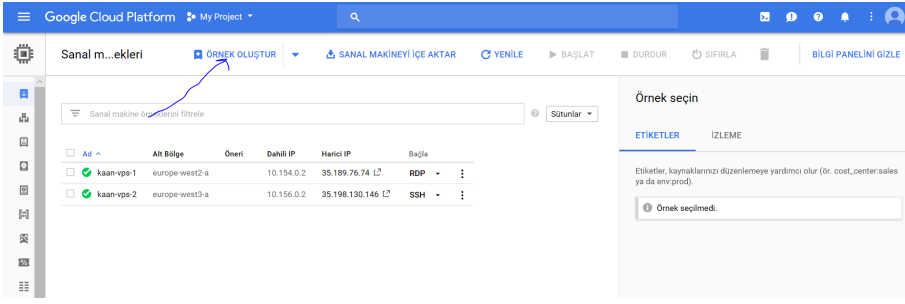
price ile area arasındaki korelasyon katsayısına da bakabiliriz:

```
> cor(price, area)
[1] 0.7621137
```

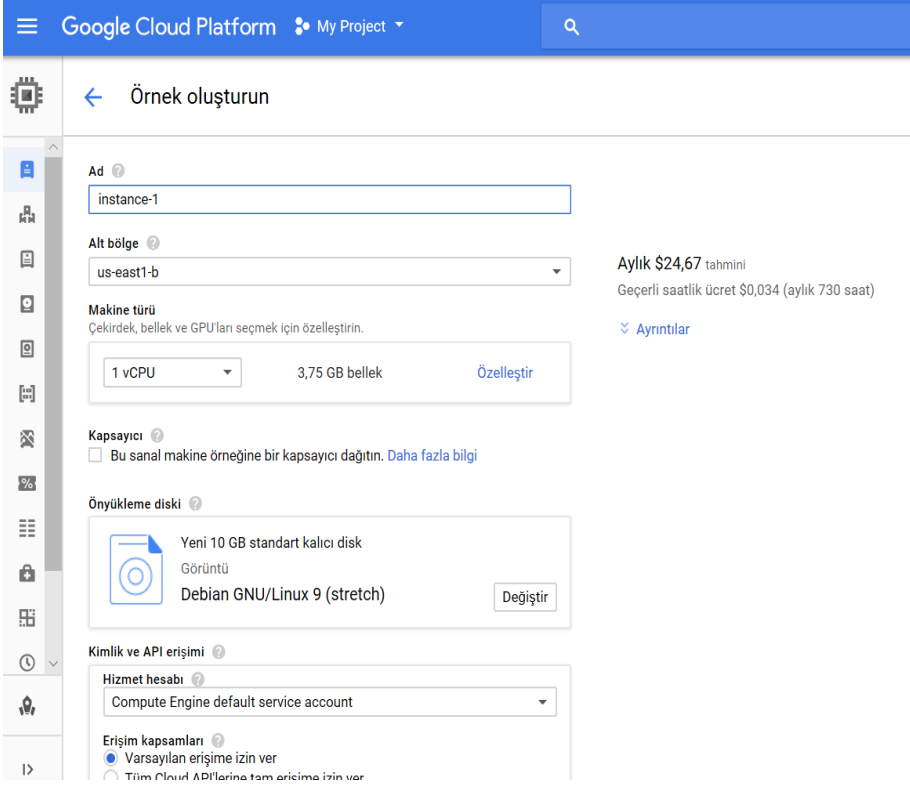
RStudioServer Programının Google Compute Cloud'ta Satın Alınan Linux Makineye Kurulması

Bunun için sırasıyla şu aşamalardan geçilir:

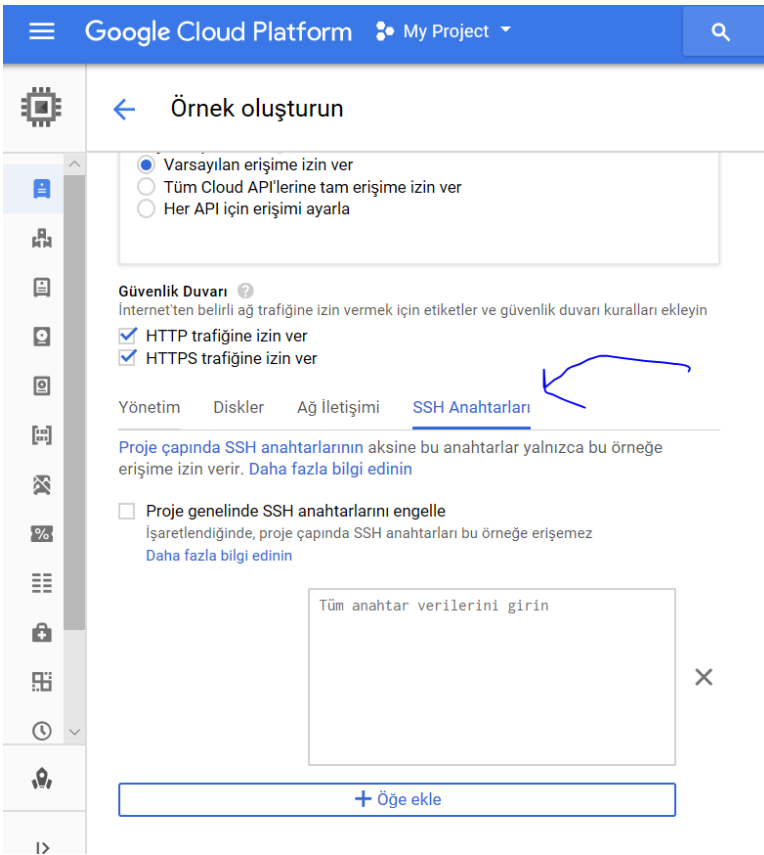
1) Önce Google Cloud'tan bir tane Compute Engine VPS kurulumu yapılır (Ücret ödemediği belli bir süre bedava kullanılabilir.) Bunun için gmail hesabı ile login olunur. Sonra cloud.google.com sitesine girilerek oradan konsol sayfasına girilir. Menüden Compute Engine/Sanal Makine Örnekleri seçilir. Buradan da "Örnek Oluştur" tuşuna basılarak yeni makine oluşturma sayfasına geçilir:



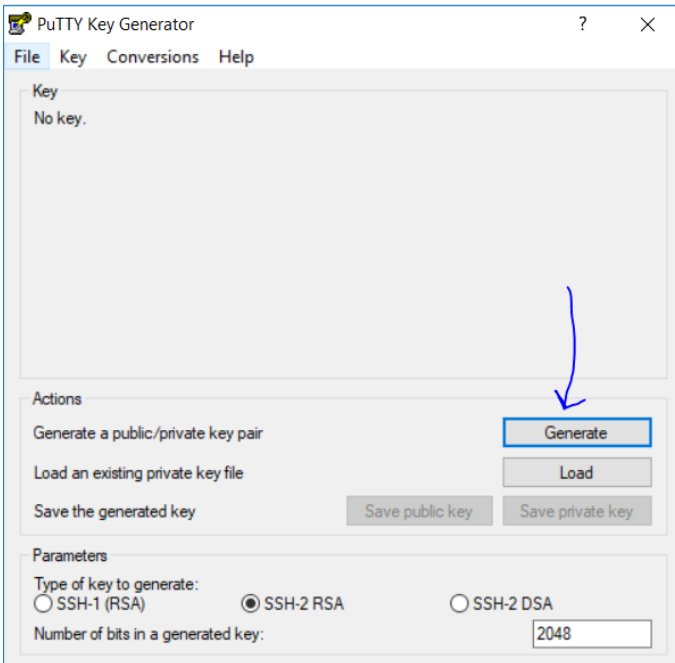
Daha sonra burada makinenin konfigürasyonu belirlenir.



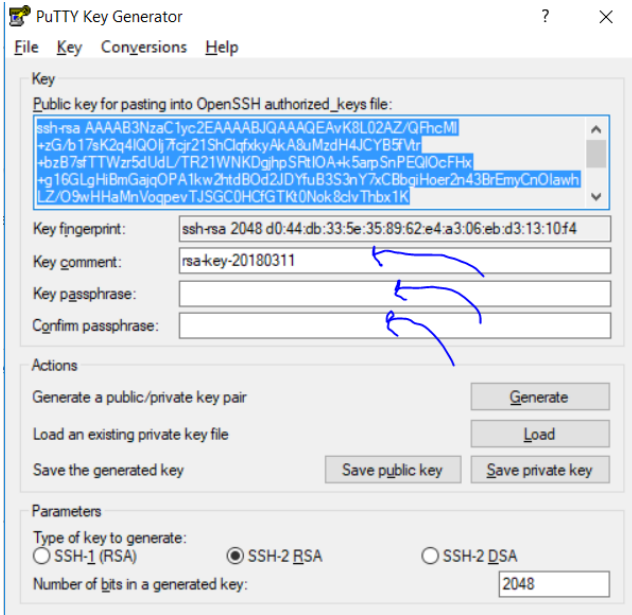
Eğer sanal miakineye SSH ile uzaktan bağlanılacaksa bir private/public kriptografi bilgisinin oluşturması gerekmektedir. Ayrıca server'ın HTTP trafiğine açık olması da gerekmektedir:



SSH anahtarı için Putty paketinin indirilmesi ve PuttyGen programının çalıştırılması gerekmektedir.



Generate tuşuna basılıp private/public üretildikten sonra private key'in dosyada saklanması gerekmektedir. Burada key comment kısmına bir kullanıcı ismi aşağıdaki edit alanına da bir password girmek gerekir. Sonra da oluşan key'i web sayfasındaki uygun yere girmek gerekir.



Bu işlemlerin adımları için Youtube'ta "Google Compute Engine ssh putty" yazılarak ilgili videolar da görsel olarak izlenebilir.

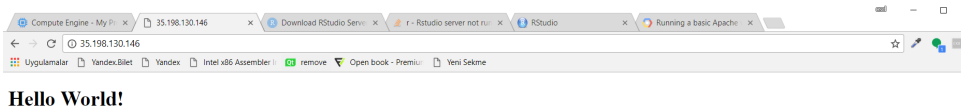
2) Cloud'ta oluşturduğumuz Linux makineye Apache Server'ın kurulması gerekir. Bu işlem Ubuntu için şöyle yapılabilir:

```
sudo apt-get update && sudo apt-get install apache2 -y
```

Aşağıdaki komutu uygulayarak web server'ın çalışıp çalışmadığını anlayabilirsiniz:

```
echo '<!doctype html><html><body><h1>Hello World!</h1></body></html>' | sudo tee /var/www/html/index.html
```

Artık Google'ın makine için bize verdiği IP'ye bağlanırsak (http olmasına dikkat ediniz. https olmasın). Örneğin:



2) Ana makine hazır olduktan sonra oaraya bağlanıp RStudio şu komutlar uygulanarak kurulur:

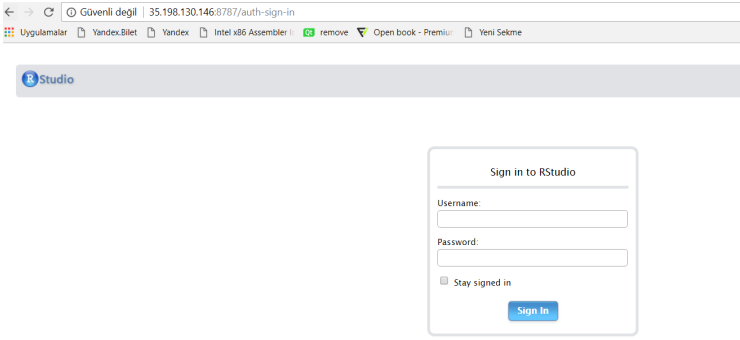
```
sudo apt-get install r-base
sudo apt-get install gdebi-core
wget https://download2.rstudio.org/rstudio-server-1.1.423-amd64.deb
sudo gdebi rstudio-server-1.1.423-amd64.deb
```

Bunun için <https://www.rstudio.com/products/rstudio/download-server/> sayfasına bakabilirsiniz.

3) Bu işlemten sonra 8787 portunu makinemizde açmak gerekir. Bu işlem şöyle yapılabilir:

```
gcloud compute firewall-rules create allow-rstudio --allow tcp:8787
```

4) Artık 8787 portuna http ile bağlandığımızda karşımıza RStudio Server'ın login ekranı çıkacaktır:



5) Artık Linux makinemize gelip RStudioServer'a bağlanmak için bir kullanıcı yaratmamız gerekir. Şu komutla kullanıcı yaratılabilir:

```
adduser <isim>
```

Burada bize bir şifre sorulacaktır. Bu işlemten sonra artık RStudio Server'ın login ekranında Username ve password burada belirlenen olarak girilirse başarılı bir biçimde login olunur.

