محاسبات آماری پیشرفته

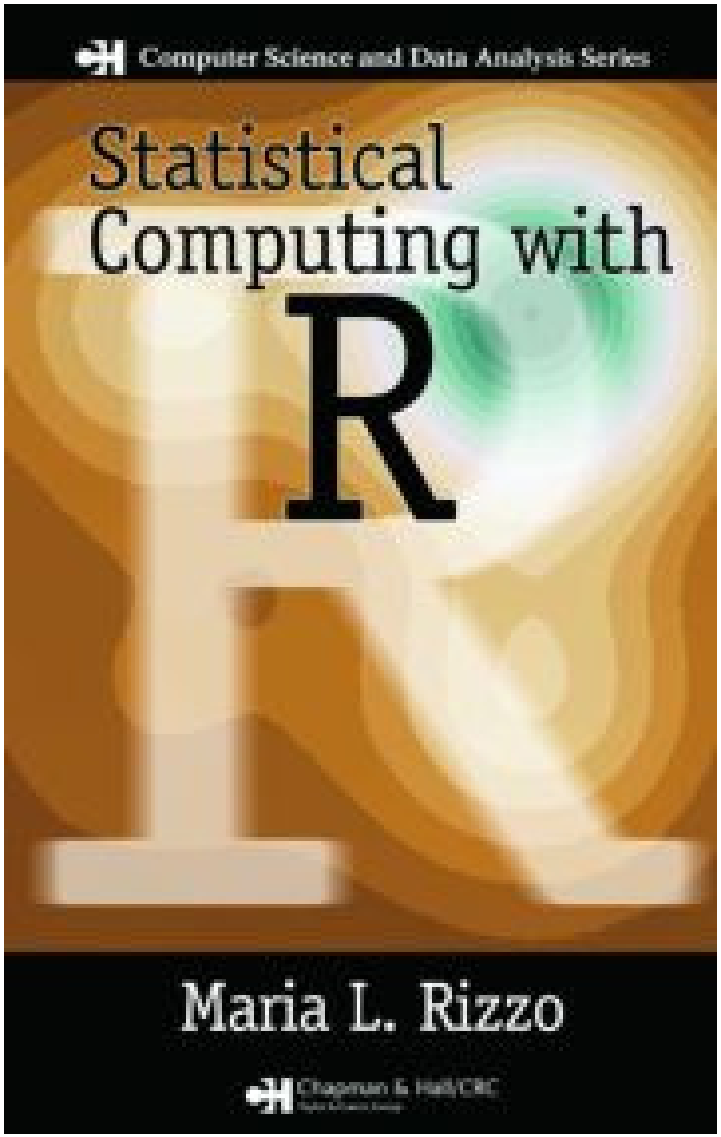
ترم اول سال تحصیلی ۹۳

جلسه اول

حسین باغیشنی

دانشگاه شاهرود

منبع


Computer Science and Data Analysis Series
Statistical Computing with R
Maria L. Rizzo
Chapman & Hall/CRC

محاسبات آماری یا آمار محاسباتی؟

محاسبات آماری یا آمار محاسباتی؟

محاسبات آماری یا آمار محاسباتی؟

آمار محاسباتی به مباحث آماری روش‌های مختلف استنباطی می‌پردازد: به عنوان مثال سازگاری برآوردگرهای به دست آمده در یک مدل خاص به وسیله روش‌های محاسباتی مختلف، یا نرخ همگرایی برآوردگرهای حاصل از روش MCMC.

محاسبات آماری یا آمار محاسباتی؟

محاسبات آماری یا آمار محاسباتی؟

آمار محاسباتی به مباحث آماری روش‌های مختلف استنباطی می‌پردازد: به عنوان مثال سازگاری برآوردگرهای به دست آمده در یک مدل خاص به وسیله روش‌های محاسباتی مختلف، یا نرخ همگرایی برآوردگرهای حاصل از روش MCMC.

محاسبات آماری که مورد نظر ما و این کلاس است، به روش‌های مختلف محاسبه کمیت‌های مورد علاقه در استنباط‌های آماری مربوط می‌شود.

محاسبات آماری یا آمار محاسباتی؟

محاسبات آماری یا آمار محاسباتی؟

آمار محاسباتی به مباحث آماری روش‌های مختلف استنباطی می‌پردازد: به عنوان مثال سازگاری برآوردگرهای به دست آمده در یک مدل خاص به وسیله روش‌های محاسباتی مختلف، یا نرخ همگرایی برآوردگرهای حاصل از روش MCMC.

محاسبات آماری که مورد نظر ما و این کلاس است، به روش‌های مختلف محاسبه کمیت‌های مورد علاقه در استنباط‌های آماری مربوط می‌شود.

البته بعضی از آماردان‌ها معتقدند این هر دو عنوان، و با جامعیت آمار محاسباتی، یکی هستند

چرا آماردان‌های خوب باید برنامه‌نویسی بلد باشند؟

- **استقلال**: در غیر این صورت باید منتظر باشید تا شخص دیگری برنامه‌های مورد نیاز شما را بنویسد و به شما تحویل دهد

چرا آماردان‌های خوب باید برنامه‌نویسی بلد باشند؟

- **استقلال**: در غیر این صورت باید منتظر باشید تا شخص دیگری برنامه‌های مورد نیاز شما را بنویسد و به شما تحویل دهد
- **صداقت**: در غیر این صورت ممکن است نتایج را طوری که مایلید اتفاق بیافتند، دستکاری کنید

چرا آماردان‌های خوب باید برنامه‌نویسی بلد باشند؟

- **استقلال**: در غیر این صورت باید منتظر باشید تا شخص دیگری برنامه‌های مورد نیاز شما را بنویسد و به شما تحویل دهد
- **صداقت**: در غیر این صورت ممکن است نتایج را طوری که مایلید اتفاق بیافتند، دستکاری کنید
- **وضوح و روشنی**: برنامه‌نویسی باعث می‌شود بتوانید فکر خود را منظم کرده و آن را با بقیه به اشتراک بگذارید

چرا آماردان‌های خوب باید برنامه‌نویسی بلد باشند؟

- **استقلال**: در غیر این صورت باید منتظر باشید تا شخص دیگری برنامه‌های مورد نیاز شما را بنویسد و به شما تحویل دهد
- **صداقت**: در غیر این صورت ممکن است نتایج را طوری که مایلید اتفاق بیافتند، دستکاری کنید
- **وضوح و روشنی**: برنامه‌نویسی باعث می‌شود بتوانید فکر خود را منظم کرده و آن را با بقیه به اشتراک بگذارید

و یادتان باشد که علم باید عمومی باشد

برنامه‌نویسی: تبدیل ورودی به خروجی

- برنامه‌نویسی، نوشتن توابعی برای تبدیل ورودی‌ها به خروجی‌هاست

برنامه‌نویسی: تبدیل ورودی به خروجی

- برنامه‌نویسی، نوشتن توابعی برای تبدیل ورودی‌ها به خروجی‌هاست
- برنامه‌نویسی خوب به معنی انجام تبدیل بیان‌شده به سادگی و درستی است

برنامه‌نویسی: تبدیل ورودی به خروجی

- برنامه‌نویسی، نوشتن توابعی برای تبدیل ورودی‌ها به خروجی‌هاست
- برنامه‌نویسی خوب به معنی انجام تبدیل بیان‌شده به سادگی و درستی است
- ماشین‌ها از ماشین‌ها ساخته می‌شوند؛ توابع نیز از توابع ساخته می‌شوند: مانند $f(a, b) = a^۲ + b^۲$

برنامه‌نویسی: تبدیل ورودی به خروجی

- برنامه‌نویسی، نوشتن توابعی برای تبدیل ورودی‌ها به خروجی‌هاست
- برنامه‌نویسی خوب به معنی انجام تبدیل بیان‌شده به سادگی و درستی است
- ماشین‌ها از ماشین‌ها ساخته می‌شوند؛ توابع نیز از توابع ساخته می‌شوند: مانند $f(a, b) = a^۲ + b^۲$
- مسیر اجرای یک برنامه‌نویسی خوب، در نظر گرفتن یک تبدیل بزرگ و شکستن آن به قطعات کوچکتر و سپس شکستن دوباره قطعات کوچکتر است مادامی که توابع ساخته‌شده وظیفه خود را به درستی و با سادگی انجام دهند

شناخت داده‌ها قبل از توابع

انواع مختلف داده‌ها

شناخت داده‌ها قبل از توابع

انواع مختلف داده‌ها

تمامی داده‌ها با ساختار دودویی، توسط بیت‌ها ($TRUE/FALSE$ یا YES/NO یا ۱/۰) نمایش داده می‌شوند.

شناخت داده‌ها قبل از توابع

انواع مختلف داده‌ها

تمامی داده‌ها با ساختار دودویی، توسط بیت‌ها ($TRUE/FALSE$ یا YES/NO یا ۱/۰) نمایش داده می‌شوند.

مقادیر دودویی مستقیم: بولی ($TRUE/FALSE$ در R).

شناخت داده‌ها قبل از توابع

انواع مختلف داده‌ها

تمامی داده‌ها با ساختار دودویی، توسط بیت‌ها ($TRUE/FALSE$ یا YES/NO یا ۱/۰) نمایش داده می‌شوند.

مقادیر دودویی مستقیم: بولی ($TRUE/FALSE$ در R).

اعداد صحیح

شناخت داده‌ها قبل از توابع

انواع مختلف داده‌ها

تمامی داده‌ها با ساختار دودویی، توسط بیت‌ها ($TRUE/FALSE$ یا YES/NO یا ۱/۰) نمایش داده می‌شوند.

مقادیر دودویی مستقیم: بولی ($TRUE/FALSE$ در R).

اعداد صحیح

کاراکترها

شناخت داده‌ها قبل از توابع

انواع مختلف داده‌ها

تمامی داده‌ها با ساختار دودویی، توسط بیت‌ها ($TRUE/FALSE$ یا YES/NO یا ۱/۰) نمایش داده می‌شوند.

مقادیر دودویی مستقیم: بولی ($TRUE/FALSE$ در R).

اعداد صحیح

کاراکترها

مقادیر گم‌شده یا بدتعریف شده: NA، NaN

شناخت داده‌ها قبل از توابع

انواع مختلف داده‌ها

تمامی داده‌ها با ساختار دودویی، توسط بیت‌ها ($TRUE/FALSE$ یا YES/NO یا ۱/۰) نمایش داده می‌شوند.

مقادیر دودویی مستقیم: بولی ($TRUE/FALSE$ در R).

اعداد صحیح

کاراکترها

مقادیر گم‌شده یا بدتعریف شده: NA، NaN

اعداد با ممیز شناور: مانند اعداد کسری. در نظر گرفتن تعداد بیت‌های بیشتر برای قسمت کسری، دقت بیشتر را نتیجه می‌دهد. برای اعداد با ممیز شناور، این تعداد دو برابر حالت استاندارد، مثلا $numeric$، است.

```
> 0.45 == 3*0.15
[1] FALSE
```

```
> 0.45 == 3*0.15
[1] FALSE
```

اغلب (نه همیشه) این اختلاف قابل صرفنظر کردن است. گرد کردن خطا در محاسبات حجیم، منجر به تجمع و بزرگی خطا میشود.

```
> 0.45 == 3*0.15
[1] FALSE
```

اغلب (نه همیشه) این اختلاف قابل صرفنظر کردن است. گرد کردن خطا در محاسبات حجیم، منجر به تجمع و بزرگی خطا میشود.

به ویژه این مساله زمانی که نتایج باید نزدیک به صفر باشند، می‌تواند خیلی مشکل‌زا باشد.
زیرا ممکن است علامت به اشتباه عوض شود

```
> 0.45-3*0.15
[1] 5.551115e-17
```

عملگرها: جمع، ضرب و ...

عملگرها: جمع، ضرب و ...

مقایسه‌ها، عملگرهای دودویی هستند

```
> 7 > 5
[1] TRUE
> 7 < 5
[1] FALSE
> 7 >= 7
[1] TRUE
> 7 <= 5
[1] FALSE
> 7 == 5
[1] FALSE
> 7 != 5
[1] TRUE
```

دقت متناهی با اعداد با ممیز شناور و مقایسه‌های دقیق، گاهی نتایجی عجیبی پدید می‌آید. دستور $all.equal()$ معمولا چنین مشکلاتی را مرتفع می‌کند.

```
(0.5-0.3) == (0.3-0.1)
[1] FALSE
> all.equal(0.5-0.3,0.3-0.1)
[1] TRUE
```

دقت متناهی با اعداد با ممیز شناور و مقایسه‌های دقیق، گاهی نتایجی عجیبی پدید می‌آید. دستور $all.equal()$ معمولا چنین مشکلاتی را مرتفع می‌کند.

```
(0.5-0.3) == (0.3-0.1)
[1] FALSE
> all.equal(0.5-0.3,0.3-0.1)
[1] TRUE
```

عملگرهای بولی برای and و or:

```
> (5 > 7) & (6*7 == 42)
[1] FALSE
> (5 > 7) | (6*7 == 42)
[1] TRUE
```

تابع $typeof$ نوع شی را نشان می‌دهد

توابع $is.foo$ یک مقدار بولی در مورد این که آرگومان تابع از نوع foo هست یا خیر نشان می‌دهند:

تابع $typeof$ نوع شی را نشان می‌دهد
توابع $is.foo$ یک مقدار بولی در مورد این که آرگومان تابع از نوع foo هست یا خیر نشان می‌دهند:

```
> typeof(7)
[1] "double"
> is.numeric(7)
[1] TRUE
> is.integer(7)
[1] FALSE
> is.character(7)
[1] FALSE
> is.character("7")
[1] TRUE
> is.character("seven")
[1] TRUE
> is.na("seven")
[1] FALSE
```

تابع $as.foo$:

```
> as.character(5/6)
[1] "0.833333333333333"
> as.numeric(as.character(5/6))
[1] 0.8333333
> 6*as.character(5/6)
Error in 6 * as.character(5/6) : non-numeric argument to binary ope
> 6*as.numeric(as.character(5/6))
[1] 5
> 5/6 == as.numeric(as.character(5/6))
[1] FALSE
```

چرا آخرین دستور $FALSE$ است؟

نام دادن به داده‌ها

می‌توان داده‌ها را نام‌گذاری کرد. این نام‌گذاری متغیرها را در اختیار ما قرار می‌دهد. بعخ نام‌ها قبلا در R ساخته شده‌اند:

```
> pi
[1] 3.141593
```

نام دادن به داده‌ها

می‌توان داده‌ها را نام‌گذاری کرد. این نام‌گذاری متغیرها را در اختیار ما قرار می‌دهد. بعخ نام‌ها قبلا در R ساخته شده‌اند:

```
> pi
[1] 3.141593
```

عملگر گمارش عبارتست از $<-$ یا $=$

```
> approx.pi <- 22/7
> approx.pi
[1] 3.142857
> diameter.in.cubits = 10
> approx.pi*diameter.in.cubits
[1] 31.42857
```

استفاده از نام‌ها و متغیرها، کدهای برنامه‌نویسی را پدید می‌آورند:

- خواندن راحت برای دیگران

نام دادن به داده‌ها

می‌توان داده‌ها را نام‌گذاری کرد. این نام‌گذاری متغیرها را در اختیار ما قرار می‌دهد. بعخ نام‌ها قبلا در R ساخته شده‌اند:

```
> pi
[1] 3.141593
```

عملگر گمارش عبارتست از $-$ یا $<$ یا $=$

```
> approx.pi <- 22/7
> approx.pi
[1] 3.142857
> diameter.in.cubits = 10
> approx.pi*diameter.in.cubits
[1] 31.42857
```

استفاده از نام‌ها و متغیرها، کدهای برنامه‌نویسی را پدید می‌آورند:

- خواندن راحت برای دیگران
- اصلاح راحت

نام دادن به داده‌ها

می‌توان داده‌ها را نام‌گذاری کرد. این نام‌گذاری متغیرها را در اختیار ما قرار می‌دهد. بعد نام‌ها قبلا در R ساخته شده‌اند:

```
> pi
[1] 3.141593
```

عملگر گمارش عبارتست از $-$ یا $=$

```
> approx.pi <- 22/7
> approx.pi
[1] 3.142857
> diameter.in.cubits = 10
> approx.pi*diameter.in.cubits
[1] 31.42857
```

استفاده از نام‌ها و متغیرها، کدهای برنامه‌نویسی را پدید می‌آورند:

- خواندن راحت برای دیگران
- اصلاح راحت
- سادگی طراحی

نام دادن به داده‌ها

می‌توان داده‌ها را نام‌گذاری کرد. این نام‌گذاری متغیرها را در اختیار ما قرار می‌دهد. بعخ نام‌ها قبلا در R ساخته شده‌اند:

```
> pi
[1] 3.141593
```

عملگر گمارش عبارتست از $-$ یا $=$

```
> approx.pi <- 22/7
> approx.pi
[1] 3.142857
> diameter.in.cubits = 10
> approx.pi*diameter.in.cubits
[1] 31.42857
```

استفاده از نام‌ها و متغیرها، کدهای برنامه‌نویسی را پدید می‌آورند:

- خواندن راحت برای دیگران
- اصلاح راحت
- سادگی طراحی
- وجود کمتر خطا

مثال: تخصیص منابع

کارخانه‌ای با استفاده از فولاد و توسط کارگرانش ماشین و کامیون تولید می‌کند:

- یک ماشین ۴۰ ساعت کار و ۱ تن فولاد نیاز دارد

مثال: تخصیص منابع

کارخانه‌ای با استفاده از فولاد و توسط کارگرانش ماشین و کامیون تولید می‌کند:

- یک ماشین ۴۰ ساعت کار و ۱ تن فولاد نیاز دارد
- یک کامیون ۶۰ ساعت کار و ۳ تن فولاد نیاز دارد

مثال: تخصیص منابع

کارخانه‌ای با استفاده از فولاد و توسط کارگرانش ماشین و کامیون تولید می‌کند:

- یک ماشین ۴۰ ساعت کار و ۱ تن فولاد نیاز دارد
- یک کامیون ۶۰ ساعت کار و ۳ تن فولاد نیاز دارد
- منابع عبارتند از: ۱۶۰۰ ساعت کار و ۷۰ تن فولاد در هر هفته

آیا می‌توان ۲۰ کامیون و ۸ ماشین تولید کرد؟

```
> 60*20 + 40*8 <= 1600
[1] TRUE
> 3*20 + 1*8 <= 70
[1] TRUE
```

مثال: تخصیص منابع

کارخانه‌ای با استفاده از فولاد و توسط کارگرانش ماشین و کامیون تولید می‌کند:

- یک ماشین ۴۰ ساعت کار و ۱ تن فولاد نیاز دارد
- یک کامیون ۶۰ ساعت کار و ۳ تن فولاد نیاز دارد
- منابع عبارتند از: ۱۶۰۰ ساعت کار و ۷۰ تن فولاد در هر هفته

آیا می‌توان ۲۰ کامیون و ۸ ماشین تولید کرد؟

```
> 60*20 + 40*8 <= 1600
[1] TRUE
> 3*20 + 1*8 <= 70
[1] TRUE
```

۲۰ کامیون و ۹ ماشین چطور؟

```
> 60*20 + 40*9 <= 1600
[1] TRUE
> 3*20 + 1*9 <= 70
[1] TRUE
```

۲۰ کامیون و ۱۰ ماشین چطور؟

۲۰ کامیون و ۱۰ ماشین چطور؟
برای پاسخ به آن می‌توان مشابه دو مورد قبل نوشت، اما:

۲۰ کامیون و ۱۰ ماشین چطور؟
برای پاسخ به آن می‌توان مشابه دو مورد قبل نوشت، اما:

- خسته‌کننده و تکراری است

۲۰ کامیون و ۱۰ ماشین چطور؟
برای پاسخ به آن می‌توان مشابه دو مورد قبل نوشت، اما:

- خسته‌کننده و تکراری است
- مراجعه به آن در آینده، ممکن است این سوال را پیش بیاورد که این اعداد چه بودند؟

۲۰ کامیون و ۱۰ ماشین چطور؟
برای پاسخ به آن می‌توان مشابه دو مورد قبل نوشت، اما:

- خسته‌کننده و تکراری است
- مراجعه به آن در آینده، ممکن است این سوال را پیش بیاورد که این اعداد چه بودند؟
- یافتن و مرتفع کردن خطاها کار مشکلی است

```
> hours.car <- 40; hours.truck <- 60
> steel.car <- 1; steel.truck <- 3
> available.hours <- 1600; available.steel <- 70
> output.trucks <- 20; output.cars <- 10
> hours.car*output.cars + hours.truck*output.trucks <= available.hours
[1] TRUE
> steel.car*output.cars + steel.truck*output.trucks <= available.steel
[1] TRUE
```

۲۰ کامیون و ۱۰ ماشین چطور؟
برای پاسخ به آن می‌توان مشابه دو مورد قبل نوشت، اما:

- خسته‌کننده و تکراری است
- مراجعه به آن در آینده، ممکن است این سوال را پیش بیاورد که این اعداد چه بودند؟
- یافتن و مرتفع کردن خطاها کار مشکلی است

```
> hours.car <- 40; hours.truck <- 60
> steel.car <- 1; steel.truck <- 3
> available.hours <- 1600; available.steel <- 70
> output.trucks <- 20; output.cars <- 10
> hours.car*output.cars + hours.truck*output.trucks <= available.hours
[1] TRUE
> steel.car*output.cars + steel.truck*output.trucks <= available.steel
[1] TRUE
```

با این شکل نوشتن، اکنون اگر لازم باشد چیزی تغییر کند فقط کافی است متغیرهای متناظر تغییر کنند و دو خط آخر را دوباره اجرا کنیم: یک مرحله به سمت ایجاز

اولین ساختار داده: بردارها

گروه‌بندی کردن مقادیر داده‌های مرتبط به هم را در یک شی، ساختار داده گویند.

یک بردار، دنباله‌ای از مقادیر هم‌نوع است:

```
> x <- c(7, 8, 10, 45)
> x
[1] 7 8 10 45
> is.vector(x)
[1] TRUE
```

اولین ساختار داده: بردارها

گروه‌بندی کردن مقادیر داده‌های مرتبط به هم را در یک شی، ساختار داده گویند.

یک بردار، دنباله‌ای از مقادیر هم‌نوع است:

```
> x <- c(7, 8, 10, 45)
> x
[1] 7 8 10 45
> is.vector(x)
[1] TRUE
```

تابع $c()$ مقادیر یک بردار را با همان ترتیبی که وارد شده است برمی‌گرداند.

$x[۱]$ اولین عنصر بردار x است، $x[۴]$ چهارمین عنصر است و $x[-۴]$ برداری است شامل همه عناصر بردار x به جز چهارمین عنصر

دستور $vector(length = ۶)$ یک بردار خالی به طول ۶ برمی‌گرداند که در بسیاری از موارد، دستور مفیدی است برای آن که بعدا توسط عناصری باید پر شوند.

```
weekly.hours <- vector(length=5)
weekly.hours[5] <- 8
```

محاسبات برداری

عملگرها بر روی بردارها به صورت جفتی عمل می‌کنند:

```
> y <- c(-7, -8, -10, -45)
> x+y
[1] 0 0 0 0
```

محاسبات برداری

عملگرها بر روی بردارها به صورت جفتی عمل می‌کنند:

```
> y <- c(-7, -8, -10, -45)
> x+y
[1] 0 0 0 0
```

دوباره‌تکراری: در عملیات جبری برداری، چنانچه برداری کوتاهتر از دیگری باشد، عناصر آن تکرار می‌شوند

```
> x + c(-7,-8)
[1] 0 0 3 37
```

بنابراین اعداد تنها، بردارهای با طول ۱ هستند که عدد مورد نظر تکرار می‌شود:

```
> x + 1
[1] 8 9 11 46
```

مقایسه‌ها نیز جفتی انجام می‌شوند:

```
[1] FALSE FALSE TRUE TRUE
```

توجه: برگرداندن بردارهای بولی

عملگرهای بولی جفتی عمل می‌کنند. اما اگر دوتایی نوشته شوند، تمام مقادیر تکی را در یک مقدار بولی تنها جمع می‌کنند:

```
> (x > 9) & (x < 20)
[1] FALSE FALSE TRUE FALSE
> (x > 9) && (x < 20)
[1] FALSE
```

برای مقایسه کل یک بردار با کل بردار دیگر، بهترین راه استفاده از $identical$ یا $all.equal$ است:

```
> x == -y
[1] TRUE TRUE TRUE TRUE
> identical(x,-y)
[1] TRUE
> identical(c(0.5-0.3,0.3-0.1),c(0.3-0.1,0.5-0.3))
[1] FALSE
> all.equal(c(0.5-0.3,0.3-0.1),c(0.3-0.1,0.5-0.3))
[1] TRUE
```

توابع بر روی بردارها

ورودی بسیاری از توابع، بردارها هستند

- $mean()$,$median()$,$sd()$,$var()$,$max()$,$min()$,$length()$,$sum()$ که خروجی همه آن‌ها یک عدد است

توابع بر روی بردارها

ورودی بسیاری از توابع، بردارها هستند

- $mean()$,$median()$,$sd()$,$var()$,$max()$,$min()$,$length()$,$sum()$ که خروجی همه آن‌ها یک عدد است
- $sort()$ که یک بردار جدید (مرتب‌شده) برمی‌گرداند

توابع بر روی بردارها

ورودی بسیاری از توابع، بردارها هستند

- $mean()$,$median()$,$sd()$,$var()$,$max()$,$min()$,$length()$,$sum()$ که خروجی همه آن‌ها یک عدد است
- $sort()$ که یک بردار جدید (مرتب‌شده) برمی‌گرداند
- $hist()$ که یک هیستوگرام تولید می‌کند

توابع بر روی بردارها

ورودی بسیاری از توابع، بردارها هستند

- $mean()$,$median()$,$sd()$,$var()$,$max()$,$min()$,$length()$,$sum()$ که خروجی همه آن‌ها یک عدد است
- $sort()$ که یک بردار جدید (مرتب‌شده) برمی‌گرداند
- $hist()$ که یک هیستوگرام تولید می‌کند
- $summary()$ که یک خلاصه شامل ۵ عدد را از بردارهای عددی تولید می‌کند

توابع بر روی بردارها

ورودی بسیاری از توابع، بردارها هستند

- $mean()$,$median()$,$sd()$,$var()$,$max()$,$min()$,$length()$,$sum()$ که خروجی همه آن‌ها یک عدد است
- $sort()$ که یک بردار جدید (مرتب‌شده) برمی‌گرداند
- $hist()$ که یک هیستوگرام تولید می‌کند
- $summary()$ که یک خلاصه شامل ۵ عدد را از بردارهای عددی تولید می‌کند
- $any()$ و $all()$ برای بردارهای بولی مفیدند

آدرس دادن عناصر بردارها

برداری از اندیس‌ها

```
> x[c(2,4)]
[1] 8 45
```

برداری از اندیس‌های منفی

```
> x[c(-1,-3)]
[1] 8 45
```

بردارهای بولی

```
> x[x>9]
[1] 10 45
> y[x>9]
[1] -10 -45
```

ورودی $which()$ یک بردار بولی است و یک بردار از اندیس‌هایی می‌دهد که مقادیر آن‌ها $TRUE$ هستند

```
> places <- which(x > 9)
> y[places]
[1] -10 -45
```

مولفه‌های دارای نام

می‌توان به عناصر یا مولفه‌های بردارها، نام‌هایی را اختصاص داد

```
> names(x) <- c("v1","v2","v3","Hossein")
> names(x)
[1] "v1" "v2" "v3" "Hossein"
> x[c("Hossein","v1")]
Hossein v1
45 7
```

دقت کنید این برچسب‌ها قسمتی از مقادیر بردار x نیستند. $names(x)$ خود یک بردار دیگر از (کاراکترها) است:

```
> names(y) <- names(x)
> sort(names(x))
[1] "Hossein" "v1" "v2" "v3"
> which(names(x)=="Hossein")
[1] 4
```

بازگشت به مثال تخصیص منابع

استفاده از بردارها برای گروه‌بندی چیزهایی با هم

```
> hours <- c(hours.car,hours.truck)
> steel <- c(steel.car,steel.truck)
> output <- c(output.cars,output.trucks)
> available <- c(available.hours,available.steel)
```

حالا می‌شود به صورت زیر عمل کرد:

```
> all(hours[1]*output[1]+hours[2]*output[2] <= available[1],
+ steel[1]*output[1]+steel[2]*output[2] <= available[2])
[1] TRUE
```

یا حتی

```
> all(c(sum(hours*output), sum(steel*output)) <= available)
[1] TRUE
```

اما با این شکل باید ترتیب مولفه‌ها (ماشین، کامیون، کارگر و فولاد) در هر بردار را به یاد داشته باشیم و همیشه از همان ترتیب استفاده کنیم

اما با این شکل باید ترتیب مولفه‌ها (ماشین، کامیون، کارگر و فولاد) در هر بردار را به یاد داشته باشیم و همیشه از همان ترتیب استفاده کنیم

به جای آن می‌توان از نام‌گذاری استفاده کرد

```
> names(hours) <- c("cars", "trucks")
> names(steel) <- names(hours)
> names(output) <- names(hours)
> names(available) <- c("hours","steel")
> all(hours["cars"]*output["cars"] + hours["trucks"]*
+ output["trucks"] <= available["hours"],
+ steel["cars"]*output["cars"] + steel["trucks"]*
+ output["trucks"] <= available["steel"])
[1] TRUE
```

تا اینجا کد نوشته شده بهتر شد، اما هنوز جای بهتر شدن دارد. مثل زیر:

```
> needed <- c(sum(hours*output[names(hours)]),
+ sum(steel*output[names(steel)]))
> names(needed) <- c("hours","steel")
> all(needed <= available[names(needed)])
[1] TRUE
```

این یک برنامه‌نویسی بی‌نقص نیست، اما بهتر از کدهای قبلی است.

ساختارهای برداری: شروع از آرایه‌ها

ساختارهای برداری، عبارتند از بردارها با ویژگی‌های اضافه. آرایه‌های چندبعدی، ساختارهای برداری هستند.

ساختارهای برداری: شروع از آرایه‌ها

ساختارهای برداری، عبارتند از بردارها با ویژگی‌های اضافه. آرایه‌های چندبعدی، ساختارهای برداری هستند.

```
> x.arr <- array(x,dim=c(2,2))
> x.arr
     [,1] [,2]
[1,]    7   10
[2,]    8   45
```

دقت کنید ابتدا ستون اول پر می‌شود و سپس ستون دوم. dim هم به R می‌گوید چند سطر و ستون در نظر بگیرد.

ساختارهای برداری: شروع از آرایه‌ها

ساختارهای برداری، عبارتند از بردارها با ویژگی‌های اضافه. آرایه‌های چندبعدی، ساختارهای برداری هستند.

```
> x.arr <- array(x,dim=c(2,2))
> x.arr
     [,1] [,2]
[1,]    7   10
[2,]    8   45
```

دقت کنید ابتدا ستون اول پر می‌شود و سپس ستون دوم. dim هم به R می‌گوید چند سطر و ستون در نظر بگیرد.

می‌توان آرایه‌های n بعدی هم داشت. بنابراین dim برداری با طول n خواهد بود. $n = ۲, ۳, \ldots,$

بعضی از ویژگی‌های آرایه‌ها

```
> dim(x.arr)
[1] 2 2
> is.vector(x.arr)
[1] FALSE
> is.array(x.arr)
[1] TRUE
> typeof(x.arr)
[1] "double"
> str(x.arr)
num [1:2, 1:2] 7 8 10 45
```

توجه: $typeof()$ نوع عناصر آرایه را نشان می‌دهد. $str()$ ساختار آرایه را نشان می‌دهد: در مثال بالا، آرایه عددی است، با دو بعد که هر دو با ۲ − ۱ اندیس‌گذاری شده‌اند، و در نهایت مقادیر عددی

دسترسی به و عملیات بر روی آرایه‌ها

به یک آرایه دوبعدی می‌توان با جفت‌های اندیسی یا بردار زیربنایی آرایه دسترسی داشت:

```
> x.arr[1,2]
[1] 10
> x.arr[3]
[1] 10
```

حذف یک اندیس به معنی همه آن است:

```
> x.arr[c(1:2),2]
[1] 10 45
> x.arr[,2]
[1] 10 45
```

استفاده از یک تابع برداری بر روی ساختارهای برداری، بر روی بردار زیربنایی آن‌ها کار مگر این که تابع برای کار اختصاصی با آرایه‌ها تنظیم شده باشد:

```
> which(x.arr > 9)
[1] 3 4
```

بسیاری از توابع با پیش‌فرض ساختار آرایه‌ای کار می‌کنند:

```
> y.arr <- array(y,dim=c(2,2))
> y.arr + x.arr
     [,1] [,2]
[1,]    0    0
[2,]    0    0
```

برخی دیگر به طور خاص بر روی هر سطر یا هر ستون از آرایه عمل می‌کنند:

```
> rowSums(x.arr)
[1] 17 53
```

خلاصه (موضعی)

تا اینجا:

- یاد گرفتیم چطوری برای تحلیل داده‌ها، برنامه‌نویسی کنیم

خلاصه (موضعی)

تا اینجا:

- یاد گرفتیم چطوری برای تحلیل داده‌ها، برنامه‌نویسی کنیم
- با ترکیب توابع برای ساخت داده‌ها، برنامه‌هایی را نوشتیم

خلاصه (موضعی)

تا اینجا:

- یاد گرفتیم چطوری برای تحلیل داده‌ها، برنامه‌نویسی کنیم
- با ترکیب توابع برای ساخت داده‌ها، برنامه‌هایی را نوشتیم
- انواع داده پایه (بولی، کاراکتر، عدد) و توابعی که بر روی آن‌ها کار می‌کنند را شناختیم

خلاصه (موضعی)

تا اینجا:

- یاد گرفتیم چطوری برای تحلیل داده‌ها، برنامه‌نویسی کنیم
- با ترکیب توابع برای ساخت داده‌ها، برنامه‌هایی را نوشتیم
- انواع داده پایه (بولی، کاراکتر، عدد) و توابعی که بر روی آن‌ها کار می‌کنند را شناختیم
- ساختارهای داده پایه (بردارها و آرایه‌ها) و توابعی که بر روی آن‌ها کار می‌کنند را شناختیم

خلاصه (موضعی)

تا اینجا:

- یاد گرفتیم چطوری برای تحلیل داده‌ها، برنامه‌نویسی کنیم
- با ترکیب توابع برای ساخت داده‌ها، برنامه‌هایی را نوشتیم
- انواع داده پایه (بولی، کاراکتر، عدد) و توابعی که بر روی آن‌ها کار می‌کنند را شناختیم
- ساختارهای داده پایه (بردارها و آرایه‌ها) و توابعی که بر روی آن‌ها کار می‌کنند را شناختیم
- یاد گرفتیم، استفاده از متغیرها، به جای ثابت‌های عددی، اولین مرحله برای خلاصه کردن کدهای برنامه‌نویسی است

خلاصه (موضعی)

تا اینجا:

- یاد گرفتیم چطوری برای تحلیل داده‌ها، برنامه‌نویسی کنیم
- با ترکیب توابع برای ساخت داده‌ها، برنامه‌هایی را نوشتیم
- انواع داده پایه (بولی، کاراکتر، عدد) و توابعی که بر روی آن‌ها کار می‌کنند را شناختیم
- ساختارهای داده پایه (بردارها و آرایه‌ها) و توابعی که بر روی آن‌ها کار می‌کنند را شناختیم
- یاد گرفتیم، استفاده از متغیرها، به جای ثابت‌های عددی، اولین مرحله برای خلاصه کردن کدهای برنامه‌نویسی است
- دانستیم داده‌ها، انواع مختلف و ساختارهای مختلفی دارند

خلاصه (موضعی)

تا اینجا:

- یاد گرفتیم چطوری برای تحلیل داده‌ها، برنامه‌نویسی کنیم
- با ترکیب توابع برای ساخت داده‌ها، برنامه‌هایی را نوشتیم
- انواع داده پایه (بولی، کاراکتر، عدد) و توابعی که بر روی آن‌ها کار می‌کنند را شناختیم
- ساختارهای داده پایه (بردارها و آرایه‌ها) و توابعی که بر روی آن‌ها کار می‌کنند را شناختیم
- یاد گرفتیم، استفاده از متغیرها، به جای ثابت‌های عددی، اولین مرحله برای خلاصه کردن کدهای برنامه‌نویسی است
- دانستیم داده‌ها، انواع مختلف و ساختارهای مختلفی دارند
- فهمیدیم ساختارهای داده، مقادیر مرتبط را گروه‌بندی می‌کنند

سایر ساختارهای داده

- ماتریس‌ها، *Matrices*

سایر ساختارهای داده

- ماتریس‌ها، *Matrices*
- لیست‌ها، *Lists*

سایر ساختارهای داده

- ماتریس‌ها، *Matrices*
- لیست‌ها، *Lists*
- ساختارهای داده، *Data frames*

سایر ساختارهای داده

- ماتریس‌ها، *Matrices*
- لیست‌ها، *Lists*
- ساختارهای داده، *Data frames*
- ساختارهایی از ساختارها، *Structures of structures*

ماتریس‌ها

در R یک ماتریس، حالت خاصی از یک آرایه است.

```
> factory <- matrix(c(40,1,60,3),nrow=2)
> factory
     [,1] [,2]
[1,]   40   60
[2,]    1    3
> is.array(factory)
[1] TRUE
> is.matrix(factory)
[1] TRUE
```

البته می‌توان از $ncol$ و $byrow = TRUE$ برای پر کردن سطری استفاده کرد.

ماتریس‌ها

در R یک ماتریس، حالت خاصی از یک آرایه است.

```
> factory <- matrix(c(40,1,60,3),nrow=2)
> factory
     [,1] [,2]
[1,]   40   60
[2,]    1    3
> is.array(factory)
[1] TRUE
> is.matrix(factory)
[1] TRUE
```

البته می‌توان از $ncol$ و $byrow = TRUE$ برای پر کردن سطری استفاده کرد.

عملگرهای جبری و مقایسه‌ای به صورت عنصر به عنصر عمل می‌کنند. مثلا در $factory/3$، تمام عناصر ماتریس $factory$ بر ۳ تقسیم می‌شوند.

ضرب ماتریسی، عملگر خاص خود را دارد: (برای ضرب ماتریس با بردار نیز از همین استفاده می‌شود).

```
> six.sevens <- matrix(rep(7,6),ncol=3)
> six.sevens
     [,1] [,2] [,3]
[1,]    7    7    7
[2,]    7    7    7
> factory %*% six.sevens # [2x2] * [2x3]
     [,1] [,2] [,3]
[1,]  700  700  700
[2,]   28   28   28
> six.sevens %*% factory # [2x3] * [2x2]
Error in six.sevens %*% factory : non-conformable arguments
> output <- c(10,20)
> factory %*% output
     [,1]
[1,] 1600
[2,]   70
> output %*% factory
     [,1] [,2]
[1,]  420  660
```

سایر عملگرهای ماتریسی:

```
> t(factory) # Matrix transpose
     [,1] [,2]
[1,]   40    1
[2,]   60    3
> det(factory) # Matrix determinant
[1] 60
> diag(factory) # Extracting or replacing the diagonal
[1] 40 3
> diag(factory) <- c(35,4) # Change it
> factory # See that it changed
     [,1] [,2]
[1,]   35   60
[2,]    1    4
> diag(factory) <- c(40,3) # Set it back for later
```

```
ag(c(3,4)) # Creating a diagonal matrix
     [,1] [,2]
[1,]    3    0
[2,]    0    4
> diag(2)
     [,1] [,2]
[1,]    1    0
[2,]    0    1
> solve(factory) # Inverting a matrix
        [,1]    [,2]
[1,]  0.0500 -0.7500
[2,] -0.0125  0.4375
> factory %*% solve(factory)
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

چرا برای معکوس کردن ماتریس‌ها از دستوری مثل $solve()$ استفاده می‌شود؟

چرا برای معکوس کردن ماتریس‌ها از دستوری مثل $solve()$ استفاده می‌شود؟

جواب‌های معادلات خطی $Ax = b$، بر حسب بردار x، به صورت زیر انجام می‌شود:

```
> available <- c(1600,70)
> solve(factory,available)
[1] 10 20
> factory %*% solve(factory,available)
     [,1]
[1,] 1600
[2,]   70
```

نام‌گذاری در ماتریس‌ها

در ماتریس‌ها، می‌توان سطرها، ستون‌ها، یا هر دو را با کمک توابع $rownames()$ و $colnames()$ نام‌گذاری کرد.

نام‌گذاری در ماتریس‌ها

در ماتریس‌ها، می‌توان سطرها، ستون‌ها، یا هر دو را با کمک توابع $rownames()$ و $colnames()$ نام‌گذاری کرد.

نام‌گذاری، به ما کمک می‌کند تا بهتر بفهمیم با چه چیزهایی کار می‌کنیم:

```
> rownames(factory) <- c("labor","steel")
> colnames(factory) <- c("cars","trucks")
> factory
      cars trucks
labor   40     60
steel    1      3
> output <- c(20,10)
> names(output) <- c("trucks","cars")
> available <- c(1600,70)
> names(available) <- c("labor","steel")
> factory %*% output # # But we've got cars and trucks mixed up
      [,1]
labor 1400
steel   50
> factory %*% output[colnames(factory)]
      [,1]
labor 1600
steel   70
> all(factory %*% output[colnames(factory)] <= available[rownames(factory)])
[1] TRUE
```

انجام کاری مشابه بر روی هر سطر یا ستون

$rowMeans()$، $colMeans()$، $rowSums()$، ...

انجام کاری مشابه بر روی هر سطر یا ستون

$rowMeans()$، $colMeans()$، $rowSums()$،...

برای ماتریس‌ها، تابع $summary()$ بر روی هر ستون به طور جداگانه اجرا می‌شود.

انجام کاری مشابه بر روی هر سطر یا ستون

$rowMeans()$، $colMeans()$، $rowSums()$،...

برای ماتریس‌ها، تابع $summary()$ بر روی هر ستون به طور جداگانه اجرا می‌شود.

یک تابع بسیار مفید، تابع $apply()$ است که دارای ۳ آرگومان هست: ماتریس، ۱ برای سطرها و ۲ برای ستون‌ها، و نام تابعی که باید اجرا شود:

```
> rowMeans(factory)
labor steel
   50     2
> apply(factory,1,mean)
labor steel
   50     2
```

دنباله‌ای از مقادیر، نه لزوماً هم‌نوع، را لیست گویند.

```
> my.distribution <- list("exponential",7,FALSE)
> my.distribution
[[1]]
[1] "exponential"

[[2]]
[1] 7

[[3]]
[1] FALSE
```

دنباله‌ای از مقادیر، نه لزوماً هم‌نوع، را لیست گویند.

```
> my.distribution <- list("exponential",7,FALSE)
> my.distribution
[[1]]
[1] "exponential"

[[2]]
[1] 7

[[3]]
[1] FALSE
```

اکثر چیزهایی که می‌توان برای بردارها به کار برد، برای لیست‌ها نیز می‌توان استفاده کرد.

دسترسی به عناصر لیست

برای دسترسی می‌توان مانند بردارها از [] یا از [[]] با یک اندیس تکی استفاده کرد:

```
> is.character(my.distribution)
[1] FALSE
> is.character(my.distribution[[1]])
[1] TRUE
> my.distribution[2]^ 2
Error in my.distribution[2]^2 : non-numeric argument
to binary operator
> my.distribution[[2]]^2
[1] 49
```

دسترسی به عناصر لیست

برای دسترسی می‌توان مانند بردارها از [] یا از [[]] با یک اندیس تکی استفاده کرد:

```
> is.character(my.distribution)
[1] FALSE
> is.character(my.distribution[[1]])
[1] TRUE
> my.distribution[2]^ 2
Error in my.distribution[2]^2 : non-numeric argument
to binary operator
> my.distribution[[2]]^2
[1] 49
```

اگر برای بردارها از [[]] برای آدرس‌دهی استفاده کنیم، چه اتفاقی می‌افتد؟

افزودن عناصری به لیست با تابع ()c امکان‌پذیر است (برای بردارهای نیز به همین صو
است).

```
> my.distribution <- c(my.distribution,7)
> my.distribution
[[1]]
[1] "exponential"
[[2]]
[1] 7
[[3]]
[1] FALSE
[[4]]
[1] 7
> length(my.distribution)
[1] 4
> length(my.distribution) <- 3
> my.distribution
[[1]]
[1] "exponential"
[[2]]
[1] 7
[[3]]
[1] FALSE
```

نام‌گذاری در لیست‌ها

بعضی یا همه عناصر یک لیست را می‌توان نام‌گذاری کرد:

```
> names(my.distribution) <- c("family","mean","is.symmetric")
> my.distribution
$family
[1] "exponential"
$mean
[1] 7
$is.symmetric
[1] FALSE
```

در این صورت به این عناصر می‌توان با نام آن‌ها همراه با علامت $ دسترسی داشت (البته $ نام و ساختار عناصر را حذف می‌کند).

```
> my.distribution$family
[1] "exponential"
> my.distribution[["family"]]
[1] "exponential"
> my.distribution["family"]
$family
[1] "exponential"
```

از نام‌گذاری در برنامه‌نویسی وقتی که یک لیست ایجاد می‌شود، زیاد استفاده می‌کنند

```
> another.distribution <- list(family="gaussian",mean=7,
+ sd=1,is.symmetric=TRUE)
> another.distribution
$family
[1] "gaussian"
$mean
[1] 7
$sd
[1] 1
$is.symmetric
[1] TRUE
```

```
my.distribution$was.estimated <- FALSE
my.distribution[["last.updated"]] <- "2011-08-30"
> my.distribution
$family
[1] "exponential"
$mean
[1] 7
$is.symmetric
[1] FALSE
$was.estimated
[1] FALSE
$last.updated
[1] "2011-08-30"
```

```
my.distribution$was.estimated <- FALSE
my.distribution[["last.updated"]] <- "2011-08-30"
> my.distribution
$family
[1] "exponential"
$mean
[1] 7
$is.symmetric
[1] FALSE
$was.estimated
[1] FALSE
$last.updated
[1] "2011-08-30"
```

حذف یک ورودی در لیست با مقداردهی آن با مقدار $NULL$ انجام می‌شود. مثلا با نوشتن

```
my.distribution$was.estimated<-NULL
```

ساختار داده‌ها = جداول داده‌های کلاسیک با n سطر برای اشیا و p ستون برای متغیرها

ساختارهای داده

ساختار داده‌ها = جداول داده‌های کلاسیک با n سطر برای اشیا و p ستون برای متغیرها

در بسیاری از موارد توابع موجود در R با ساختار داده‌ها سر و کار دارند.

ساختارهای داده

ساختار داده‌ها = جداول داده‌های کلاسیک با n سطر برای اشیا و p ستون برای متغیرها

در بسیاری از موارد توابع موجود در R با ساختار داده‌ها سر و کار دارند.

ساختار داده یک ماتریس نیست، (چرا؟)

ساختارهای داده

ساختار داده‌ها = جداول داده‌های کلاسیک با n سطر برای اشیا و p ستون برای متغیرها

در بسیاری از موارد توابع موجود در R با ساختار داده‌ها سر و کار دارند.

ساختار داده یک ماتریس نیست، (چرا؟)

ترکیبی است از یک ماتریس و یک لیست. می‌توان به ستون‌های آن مانند یک ماتریس یا قسمت‌های نام‌گذاری‌شده یک لیست دسترسی داشت.

ساختارهای داده

ساختار داده‌ها = جداول داده‌های کلاسیک با n سطر برای اشیا و p ستون برای متغیرها

در بسیاری از موارد توابع موجود در R با ساختار داده‌ها سر و کار دارند.

ساختار داده یک ماتریس نیست، (چرا؟)

ترکیبی است از یک ماتریس و یک لیست. می‌توان به ستون‌های آن مانند یک ماتریس یا قسمت‌های نام‌گذاری‌شده یک لیست دسترسی داشت.

بسیاری از توابع ماتریس‌ها برای ساختارهای داده هم قابل به کار گیری است ($rowSums()$، $apply()$، $summary()$ و...).

ساختارهای داده

ساختار داده‌ها = جداول داده‌های کلاسیک با n سطر برای اشیا و p ستون برای متغیرها

در بسیاری از موارد توابع موجود در R با ساختار داده‌ها سر و کار دارند.

ساختار داده یک ماتریس نیست، (چرا؟)

ترکیبی است از یک ماتریس و یک لیست. می‌توان به ستون‌های آن مانند یک ماتریس یا قسمت‌های نام‌گذاری‌شده یک لیست دسترسی داشت.

بسیاری از توابع ماتریس‌ها برای ساختارهای داده هم قابل به کار گیری است ($rowSums()$، $apply()$، $summary()$ و...).

ضرب ماتریسی را نمی‌توان برای یک ساختار داده به کار برد، حتی اگر همه ورودی آن اعداد باشند.

```
> a.matrix <- matrix(c(35,8,10,4),nrow=2)
> colnames(a.matrix) <- c("v1","v2")
> a.matrix
     v1 v2
[1,] 35 10
[2,]  8  4
> a.matrix$v1 # The $ access operator doesn't work on a matrix
Error in a.matrix$v1 : $ operator is invalid for atomic vectors
> a.data.frame <- data.frame(a.matrix,logicals=c(TRUE,FALSE))
> a.data.frame
  v1 v2 logicals
1 35 10     TRUE
2  8  4    FALSE
> a.data.frame$v1 # But $ does work on a data frame
[1] 35 8
> a.data.frame[,"v1"]
[1] 35 8
> a.data.frame[1,]
  v1 v2 logicals
1 35 10     TRUE
> colMeans(a.data.frame)
v1 v2 logicals
21.5 7.0 0.5
```

با $rbind()$ و $cbind()$ می‌توان سطرها یا ستون‌هایی را به یک آرایه یا ساختار داده اضافه کرد. البته باید مراقب نوع تبدیل‌های القاشده باشید:

```
> rbind(a.data.frame,list(v1=-3,v2=-5,logicals=TRUE))
  v1 v2 logicals
1 35 10     TRUE
2  8  4    FALSE
3 -3 -5     TRUE
> rbind(a.data.frame,c(3,4,6))
  v1 v2 logicals
1 35 10        1
2  8  4        0
3  3  4        6
```

ساختارهایی از ساختارها

لیست‌هایی از لیست‌ها، لیست‌هایی از بردارها، لیست‌هایی از لیست‌هایی از لیست‌های بردارها، و ....

ساختارهایی از ساختارها

لیست‌هایی از لیست‌ها، لیست‌هایی از بردارها، لیست‌هایی از لیست‌هایی از لیست‌های بردارها، و ....

این قابلیت ایجاد ساختارها، به ما این امکان را می‌دهد تا ساختارهای داده پیچیده مورد نیاز در برخی از موارد را بتوانیم بسازیم.

ساختارهایی از ساختارها

لیست‌هایی از لیست‌ها، لیست‌هایی از بردارها، لیست‌هایی از لیست‌هایی از لیست‌های بردارها، و ....

این قابلیت ایجاد ساختارها، به ما این امکان را می‌دهد تا ساختارهای داده پیچیده مورد نیاز در برخی از موارد را بتوانیم بسازیم.

بسیاری از اشیاء پیچیده، لیست‌هایی از ساختارهای داده هستند.

مثال

تابع $eigen()$ مقادیر و بردارهای ویژه یک ماتریس را محاسبه می‌کند.

مثال

تابع $eigen()$ مقادیر و بردارهای ویژه یک ماتریس را محاسبه می‌کند.

مقادیر خروجی این تابع، لیستی از یک بردار (مقادیر ویژه) و یک ماتریس (از بردارهای ویژه) است.

```
> eigen(factory)
$values
[1] 41.556171  1.443829
$vectors
           [,1]       [,2]
[1,] 0.99966383 -0.8412758
[2,] 0.02592747  0.5406062
> class(eigen(factory))
[1] "list"
> str(eigen(factory))
List of 2
 $ values : num [1:2] 41.56 1.44
 $ vectors: num [1:2, 1:2] 0.9997 0.0259 -0.8413 0.5406
```

با اشیاء پیچیده، می‌توانید به قسمت‌هایی از قسمت‌هایی (از قسمت‌ها...) دسترسی داشته باشید:

```
> factory %*% eigen(factory)$vectors[,2]
             [,1]
labor -1.2146583
steel  0.7805429
> eigen(factory)$values[2] * eigen(factory)$vectors[,2]
[1] -1.2146583 0.7805429
> eigen(factory)$values[2]
[1] 1.443829
> eigen(factory)[[1]][[2]] # NOT [[1, 2]]
[1] 1.443829
```

خلاصه (موضعی)

- در بسیاری از مواقع با ساختارهای ماتریسی سر و کار داریم

خلاصه (موضعی)

- در بسیاری از مواقع با ساختارهای ماتریسی سر و کار داریم
- لیست‌ها این امکان را می‌دهند تا انواع متفاوتی از داده‌ها را با هم ترکیب کنیم

خلاصه (موضعی)

- در بسیاری از مواقع با ساختارهای ماتریسی سر و کار داریم
- لیست‌ها این امکان را می‌دهند تا انواع متفاوتی از داده‌ها را با هم ترکیب کنیم
- ساختارهای داده، ترکیبی از ماتریس‌ها و لیست‌ها هستند که برای جداول داده معمول به کار می‌روند

خلاصه (موضعی)

- در بسیاری از مواقع با ساختارهای ماتریسی سر و کار داریم
- لیست‌ها این امکان را می‌دهند تا انواع متفاوتی از داده‌ها را با هم ترکیب کنیم
- ساختارهای داده، ترکیبی از ماتریس‌ها و لیست‌ها هستند که برای جداول داده معمول به کار می‌روند
- استفاده از نام‌گذاری مولفه‌ها، باعث می‌شود داده‌ها معنی‌دار شده و دسترسی به آن‌ها با کنترل بیشتری صورت گیرد

خلاصه (موضعی)

- در بسیاری از مواقع با ساختارهای ماتریسی سر و کار داریم
- لیست‌ها این امکان را می‌دهند تا انواع متفاوتی از داده‌ها را با هم ترکیب کنیم
- ساختارهای داده، ترکیبی از ماتریس‌ها و لیست‌ها هستند که برای جداول داده معمول به کار می‌روند
- استفاده از نام‌گذاری مولفه‌ها، باعث می‌شود داده‌ها معنی‌دار شده و دسترسی به آن‌ها با کنترل بیشتری صورت گیرد
- استفاده مکرر از لیست‌های ساختارهای داده، منجر به تولید ساختارهای پیچیده می‌شود