Data Science with R and Python

Data type and data structure

Peng Zhang

School of Mathematical Sciences, Zhejiang University

2023/07/03

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Data Type

All data is represented in binary format, by **bits** (TRUE/FALSE, YES/NO, 1/0)

- **Booleans** Direct binary values: `TRUE` or `FALSE` in R
- **Integers**: whole numbers (positive, negative or zero), represented by a fixed-length block of bits
- **Characters** fixed-length blocks of bits, with special coding; **strings** = sequences of characters
- **Floating point numbers**: a fraction (with a finite number of bits) times an exponent, like 1.87×10^6, but in binary form
- **Missing or ill-defined values**: `NA`, `NaN`, etc.

More types

`typeof()` function returns the type

`is.`*foo*`()` functions return Booleans for whether the argument is of type *foo*

`as.`*foo*`()` (tries to) "cast" its argument to type *foo* --- to translate it sensibly into a *foo*-type value

```r
typeof(7)
```

```
## [1] "double"
```

```r
is.numeric(7)
```

```
## [1] TRUE
```

```r
is.na(7/0)
```

```
## [1] FALSE
```

```r
is.na(0/0)   # Why is 7/0 not NA, but 0/0 is?
```

```
## [1] TRUE
```

```
is.character(7)
```

```
## [1] FALSE
```

```
is.character("7")
```

```
## [1] TRUE
```

```
is.character("seven")
```

```
## [1] TRUE
```

```
is.na("seven")
```

```
## [1] FALSE
```

```r
as.character(5/6)
```

```
## [1] "0.833333333333333"
```

```r
as.numeric(as.character(5/6))
```

```
## [1] 0.8333333
```

```r
6*as.numeric(as.character(5/6))
```

```
## [1] 5
```

```r
5/6 == as.numeric(as.character(5/6))
```

```
## [1] FALSE
```

(why is that last FALSE?)

Data can have names

We can give names to data objects; these give us **variables**

A few variables are built in:

```
pi
```

```
## [1] 3.141593
```

Variables can be arguments to functions or operators, just like constants:

```
pi*10
```

```
## [1] 31.41593
```

```
cos(pi)
```

```
## [1] -1
```

Peculiarites of floating-point numbers

The more bits in the fraction part, the more precision

The R floating-point data type is a `double`, a.k.a. `numeric` back when memory was expensive, the now-standard number of bits was twice the default

Finite precision ⇒ arithmetic on `doubles` ≠ arithmetic on $\mathbb{R}$.

```r
0.45 == 3 * 0.15
```

```
## [1] FALSE
```

```r
0.45 - 3 * 0.15
```

```
## [1] 5.551115e-17
```

Often ignorable, but not always

- Rounding errors tend to accumulate in long calculations
- When results should be ≈ 0, errors can flip signs
- Usually better to use `all.equal()` than exact comparison

```
(0.5 - 0.3) == (0.3 - 0.1)
```

```
## [1] FALSE
```

```
all.equal(0.5 - 0.3, 0.3 - 0.1)
```

```
## [1] TRUE
```

Data structures

- Vectors
- Arrays
- Matrices
- Lists
- Dataframes
- Structures of structures

First data structure: vectors

Group related data values into one object, a **data structure**

A **vector** is a sequence of values, all of the same type

```
x <- c(7, 8, 10, 45)
x
```

```
## [1]  7  8 10 45
```

```
is.vector(x)
```

```
## [1] TRUE
```

`c()` function returns a vector containing all its arguments in order

`x[1]` is the first element, `x[4]` is the 4th element
`x[-4]` is a vector containing all but the fourth element

`vector(length=6)` returns an empty vector of length 6; helpful for filling things up later

```
weekly_hours <- vector(length=5)
weekly_hours[5] <- 8
```

Vector arithmetic

Operators apply to vectors "pairwise" or "elementwise":

```
y <- c(-7, -8, -10, -45)
x + y
```

```
## [1] 0 0 0 0
```

```
x * y
```

```
## [1]   -49   -64  -100 -2025
```

Recycling

Recycling repeat elements in shorter vector when combined with longer

```
x + c(-7, -8)
```

```
## [1]  0  0  3 37
```

```
x ^ c(1, 0, -1, 0.5)
```

```
## [1] 7.000000 1.000000 0.100000 6.708204
```

Single numbers are vectors of length 1 for purposes of recycling:

```
2 * x
```

```
## [1] 14 16 20 90
```

Can also do pairwise comparisons:

```r
x > 9
```

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

Note: returns Boolean vector

Boolean operators work elementwise:

```r
(x > 9) & (x < 20)
```

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

To compare whole vectors, best to use `identical()` or `all.equal()`:

```
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
```

Addressing vectors

Vector of indices:

```
x[c(2,4)]
```

```
## [1]  8 45
```

Vector of negative indices

```
x[c(-1,-3)]
```

```
## [1]  8 45
```

(why that, and not `8 10`?)

Boolean vector:

```
x[x>9]
```

```
## [1] 10 45
```

```
y[x>9]
```

```
## [1] -10 -45
```

`which()` turns a Boolean vector in vector of TRUE indices:

```
places <- which(x > 9)
places
```

```
## [1] 3 4
```

```
y[places]
```

```
## [1] -10 -45
```

Named components

You can give names to elements or components of vectors

```r
names(x) <- c("v1","v2","v3","fred")
names(x)
```

```
## [1] "v1"   "v2"   "v3"   "fred"
```

```r
x[c("fred","v1")]
```

```
## fred   v1
##   45    7
```

note the labels in what R prints; not actually part of the value `names(x)` is just another vector (of characters):

```
names(y) <- names(x)
sort(names(x))
```

```
## [1] "fred" "v1"   "v2"   "v3"
```

```
which(names(x)=="fred")
```

```
## [1] 4
```

Functions on vectors

Lots of functions take vectors as arguments:

- `mean()`, `median()`, `sd()`, `var()`, `max()`, `min()`, `length()`, `sum()`: return single numbers
- `sort()` returns a new vector
- `hist()` takes a vector of numbers and produces a histogram, a highly structured object, with the side-effect of making a plot
- Similarly `ecdf()` produces a cumulative-density-function object
- `summary()` gives a five-number summary of numerical vectors
- `any()` and `all()` are useful on Boolean vectors

Not all functions have (or require) arguments:

```
date()
```

```
## [1] "Tue Jun 27 15:10:34 2023"
```

Vector structures, starting with arrays

Many data structures in R are made by adding bells and whistles to vectors, so "vector structures"

Most useful: **arrays**

```r
x <- c(7, 8, 10, 45)
x.arr <- array(x,dim=c(2,2))
x.arr
```

```
##      [,1] [,2]
## [1,]    7   10
## [2,]    8   45
```

`dim` says how many rows and columns; filled by columns

Can have $3, 4, \ldots n$ dimensional arrays; `dim` is a length n vector

Some properties of the array:

```
dim(x.arr)
```

```
## [1] 2 2
```

```
is.vector(x.arr)
```

```
## [1] FALSE
```

```
is.array(x.arr)
```

```
## [1] TRUE
```

```r
typeof(x.arr)
```

```
## [1] "double"
```

```r
str(x.arr)
```

```
##  num [1:2, 1:2] 7 8 10 45
```

```r
attributes(x.arr)
```

```
## $dim
## [1] 2 2
```

`typeof()` returns the type of the *elements*

`str()` gives the **structure**: here, a numeric array, with two dimensions, both indexed 1--2, and then the actual numbers

Exercise: try all these with `x`

Accessing and operating on arrays

Can access a 2-D array either by pairs of indices or by the underlying vector:

```
x.arr[1,2]
```

```
## [1] 10
```

```
x.arr[3]
```

```
## [1] 10
```

Omitting an index means "all of it":

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

```
## [1] 10 45
```

```
x.arr[,2]
```

```
## [1] 10 45
```

Functions on arrays

Using a vector-style function on a vector structure will go down to the underlying vector, *unless* the function is set up to handle arrays specially:

```
which(x.arr > 9)
```

```
## [1] 3 4
```

Many functions *do* preserve array structure:

```
y <- -x
y.arr <- array(y,dim=c(2,2))
y.arr + x.arr
```

```
##      [,1] [,2]
## [1,]    0    0
## [2,]    0    0
```

Others specifically act on each row or column of the array separately:

```
rowSums(x.arr)
```

```
## [1] 17 53
```

We will see a lot more of this idea

Example: Price of houses in PA

Census data for California and Pennsylvania on housing prices, by Census "tract"

```r
calif_penn <- read_csv("data/calif_penn_2011.csv")
penn <- calif_penn %>% filter(STATEFP == '42') #%>% filter(!(Median_
penn <- penn[!penn$TRACTCE %in% c('961102', '000800'),]
coefficients(lm(Median_house_value ~ Median_household_income, data=pe
```

```
##             (Intercept) Median_household_income
##           -26434.729858                3.655054
```

Fit a simple linear model, predicting median house price from median household income

Census tracts 24--425 are Allegheny county

Tract 24 has a median income of `$14,719`; actual median house value is `$34,100` --- is that above or below what's?

```
34100 < -26206.564 + 3.651*14719
```

```
## [1] FALSE
```

Tract 25 has income `$48,102` and house price `$155,900`

```
155900 < -26206.564 + 3.651*48102
```

```
## [1] FALSE
```

What about tract 26?

We *could* just keep plugging in numbers like this, but that's

- boring and repetitive
- error-prone (what if I forget to change the median income, or drop a minus sign from the intercept?)
- obscure if we come back to our work later (what *are* these numbers?)

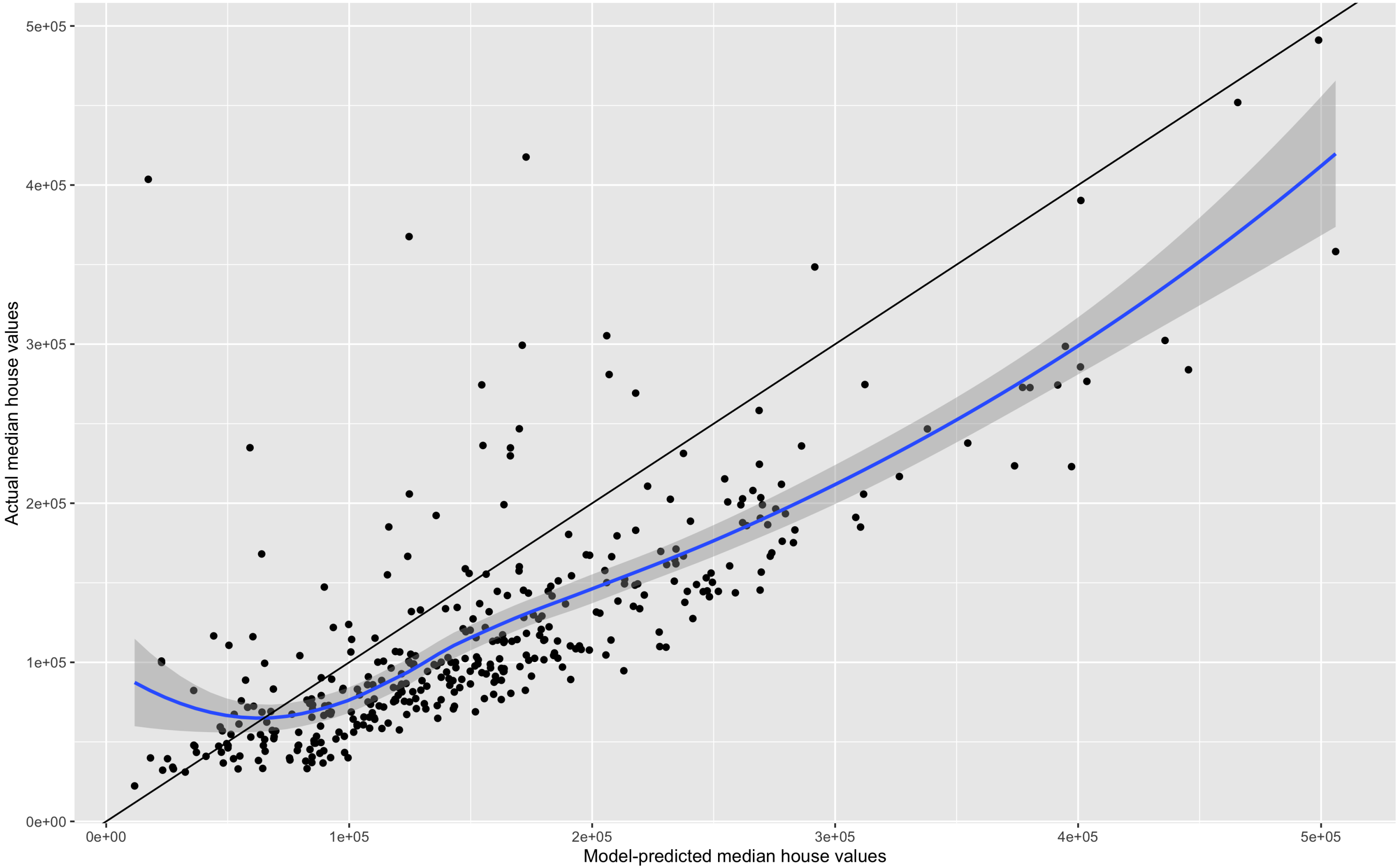
Use variables and names

```
penn.coefs <- coefficients(lm(Median_house_value ~ Median_household_
penn.coefs
```

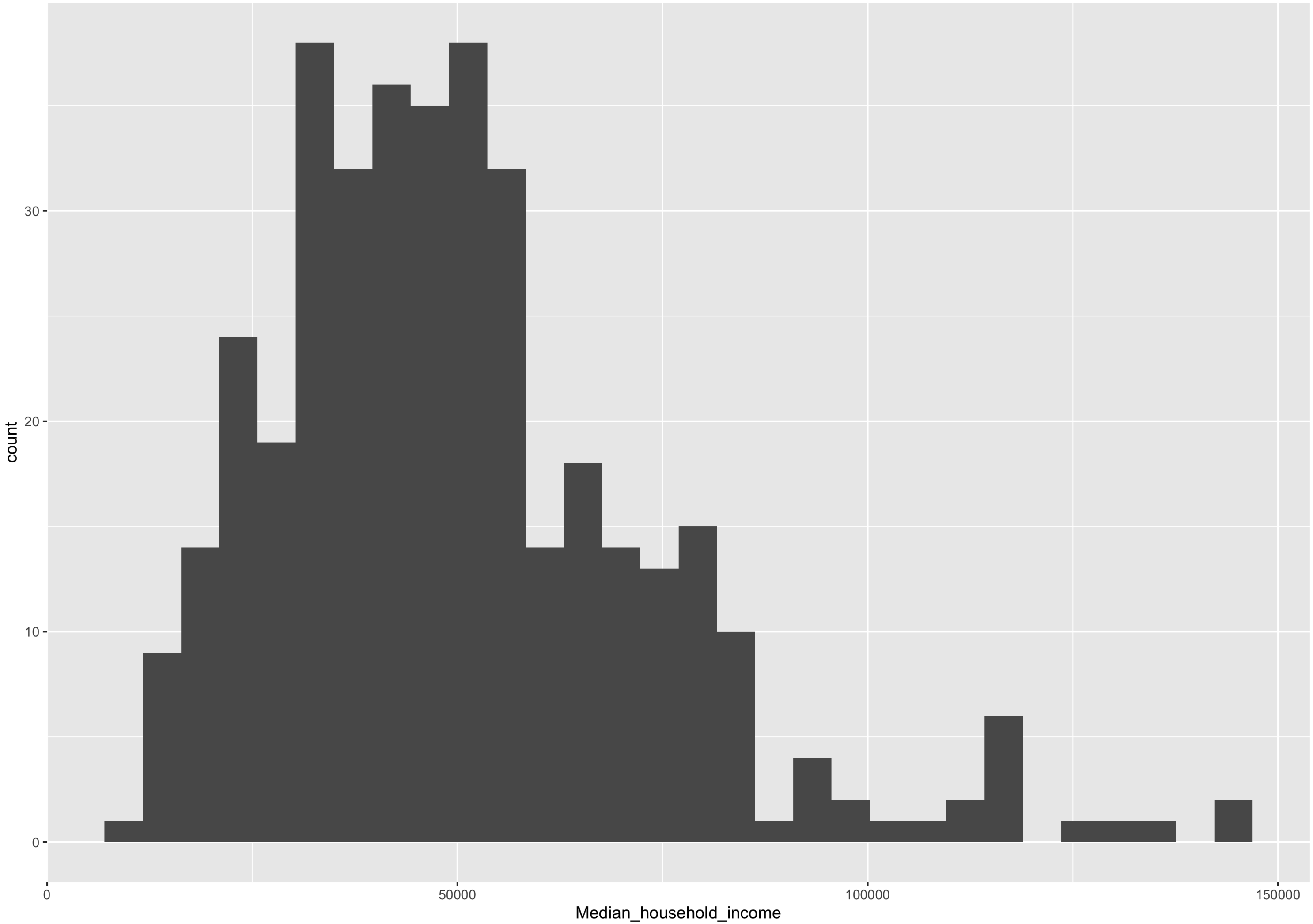
```
##              (Intercept) Median_household_income
##            -26434.729858                3.655054
```

```
allegheny.rows <- 24:425
allegheny <- penn %>% slice(allegheny.rows) %>%
  mutate(fitted = penn.coefs["(Intercept)"]+
  penn.coefs["Median_household_income"]*Median_household_income)
allegheny <- allegheny[complete.cases(allegheny),]
```

```r
allegheny %>% ggplot(aes(x = fitted, y = Median_house_value))+
  geom_point()+
  labs(x = "Model-predicted median house values",
     y ="Actual median house values")+
  geom_smooth() + geom_abline()
```



```
allegheny %>% ggplot(aes(x = Median_household_income))+
  geom_histogram()
```



Running example: resource allocation ("mathematical programming")

Factory makes cars and trucks, using labor and steel

- a car takes 40 hours of labor and 1 ton of steel
- a truck takes 60 hours and 3 tons of steel
- resources: 1600 hours of labor and 70 tons of steel each week

Matrices

In R, a matrix is a specialization of a 2D array

```r
factory <- matrix(c(40,1,60,3),nrow=2)
is.array(factory)
```

```
## [1] TRUE
```

```r
is.matrix(factory)
```

```
## [1] TRUE
```

could also specify `ncol`, and/or `byrow=TRUE` to fill by rows.

Element-wise operations with the usual arithmetic and comparison operators (e.g., `factory/3`)

Compare whole matrices with `identical()` or `all.equal()`

Matrix multiplication

Gets a special operator

```
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
```

What happens if you try `six.sevens %*% factory`?

Multiplying matrices and vectors

Numeric vectors can act like proper vectors:

```
output <- c(10,20)
factory %*% output
```

```
##      [,1]
## [1,] 1600
## [2,]   70
```

```
output %*% factory
```

```
##      [,1] [,2]
## [1,]  420  660
```

R silently casts the vector as either a row or a column matrix

Matrix operators

Transpose:

```
t(factory)
```

```
##      [,1] [,2]
## [1,]   40    1
## [2,]   60    3
```

Determinant:

```
det(factory)
```

```
## [1] 60
```

The diagonal

The `diag()` function can extract the diagonal entries of a matrix:

```
diag(factory)
```

```
## [1] 40  3
```

It can also *change* the diagonal:

```
diag(factory) <- c(35,4)
factory
```

```
##      [,1] [,2]
## [1,]   35   60
## [2,]    1    4
```

Re-set it for later:

```
diag(factory) <- c(40,3)
```

Creating a diagonal or identity matrix

```
diag(c(3,4))
```

```
##      [,1] [,2]
## [1,]    3    0
## [2,]    0    4
```

```
diag(2)
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

Inverting a matrix

```
solve(factory)
```

```
##                [,1]       [,2]
## [1,]  0.05000000 -1.0000000
## [2,] -0.01666667  0.6666667
```

```
factory %*% solve(factory)
```

```
##                 [,1]          [,2]
## [1,] 1.000000e+00 -2.220446e-15
## [2,] 3.469447e-18  1.000000e+00
```

Why's it called "solve"" anyway?

Solving the linear system $\mathbf{A}\vec{x} = \vec{b}$ for $\vec{x}$:

```
available <- c(1600,70)
solve(factory,available)
```

```
## [1] 10 20
```

```
factory %*% solve(factory,available)
```

```
##      [,1]
## [1,] 1600
## [2,]   70
```

Names in matrices

We can name either rows or columns or both, with `rownames()` and `colnames()`

These are just character vectors, and we use the same function to get and to set their values

Names help us understand what we're working with

Names can be used to coordinate different objects

```
rownames(factory) <- c("labor","steel")
colnames(factory) <- c("cars","trucks")
factory
```

```
##       cars trucks
## labor   40     60
## steel    1      3
```

```
available <- c(1600,70)
names(available) <- c("labor","steel")
```

```
output <- c(20,10)
names(output) <- c("trucks","cars")
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(facto
```

```
## [1] TRUE
```

Notice: Last lines don't have to change if we add motorcycles as output or rubber and glass as inputs (abstraction again)

Doing the same thing to each row or column

Take the mean: `rowMeans()`, `colMeans()`: input is matrix, output is vector. Also `rowSums()`, etc.

`summary()`: vector-style summary of column

```
colMeans(factory)
```

```
##   cars trucks
##   20.5   31.5
```

```
summary(factory)
```

```
##       cars           trucks
##  Min.   : 1.00   Min.   : 3.00
##  1st Qu.:10.75   1st Qu.:17.25
##  Median :20.50   Median :31.50
##  Mean   :20.50   Mean   :31.50
##  3rd Qu.:30.25   3rd Qu.:45.75
##  Max.   :40.00   Max.   :60.00
```

`apply()`, takes 3 arguments: the array or matrix, then 1 for rows and 2 for columns, then name of the function to apply to each

```
rowMeans(factory)
```

```
## labor steel
##    50     2
```

```
apply(factory,1,mean)
```

```
## labor steel
##    50     2
```

What would `apply(factory,1,sd)` do?

Lists

Sequence of values, *not* necessarily all of the same type

```
my.distribution <- list("exponential",7,FALSE)
my.distribution
```

```
## [[1]]
## [1] "exponential"
## 
## [[2]]
## [1] 7
## 
## [[3]]
## [1] FALSE
```

Most of what you can do with vectors you can also do with lists

Accessing pieces of lists

Can use `[ ]` as with vectors
or use `[[ ]]`, but only with a single index
`[[ ]]` drops names and structures, `[ ]` does not

```
is.character(my.distribution)
```

```
## [1] FALSE
```

```
is.character(my.distribution[[1]])
```

```
## [1] TRUE
```

```
my.distribution[[2]]^2
```

```
## [1] 49
```

What happens if you try `my.distribution[2]^2`? What happens if you try `[[ ]]` on a vector?

```
#my.distribution[2]^2
available[[2]]
```

Expanding and contracting lists

Add to lists with `c()` (also works with vectors):

```
my.distribution <- c(my.distribution,7)
my.distribution
```

```
## [[1]]
## [1] "exponential"
##
## [[2]]
## [1] 7
##
## [[3]]
## [1] FALSE
##
## [[4]]
## [1] 7
```

Chop off the end of a list by setting the length to something smaller (also works with vectors):

```
length(my.distribution)
```

```
## [1] 4
```

```
length(my.distribution) <- 3
my.distribution
```

```
## [[1]]
## [1] "exponential"
##
## [[2]]
## [1] 7
##
## [[3]]
## [1] FALSE
```

Naming list elements

We can name some or all of the elements of a list

```
names(my.distribution) <- c("family","mean","is.symmetric")
my.distribution
```

```
## $family
## [1] "exponential"
##
## $mean
## [1] 7
##
## $is.symmetric
## [1] FALSE
```

```
my.distribution[["family"]]
```

```
## [1] "exponential"
```

```
my.distribution["family"]
```

```
## $family
## [1] "exponential"
```

Lists have a special short-cut way of using names, `$` (which removes names and structures):

```
my.distribution[["family"]]
```

```
## [1] "exponential"
```

```
my.distribution$family
```

```
## [1] "exponential"
```

Names in lists (cont'd.)

Creating a list with names:

```
another.distribution <- list(family="gaussian",mean=7,sd=1,is.symmetr
```

Adding named elements:

```
my.distribution$was.estimated <- FALSE
my.distribution[["last.updated"]] <- "2011-08-30"
```

Removing a named list element, by assigning it the value `NULL`:

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

Key-Value pairs

Lists give us a way to store and look up data by *name*, rather than by *position*

A really useful programming concept with many names: **key-value pairs**, **dictionaries**, **associative arrays**, **hashes**

If all our distributions have components named `family`, we can look that up by name, without caring where it is in the list

Dataframes

Dataframe = the classic data table, n rows for cases, p columns for variables

Lots of the really-statistical parts of R presume data frames `penn` from last time was really a dataframe

Not just a matrix because *columns can have different types*

Many matrix functions also work for dataframes (`rowSums()`, `summary()`, `apply()`)

but no matrix multiplying dataframes, even if all columns are numeric

```r
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
```

```r
a.matrix[,"v1"]  # Try a.matrix$v1 and see what happens
```

```
## [1] 35  8
```

```
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
```

```
## [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
```

Adding rows and columns

We can add rows or columns to an array or data-frame with `rbind()` and `cbind()`, but be careful about forced type conversions

```
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
```

Structures of Structures

So far, every list element has been a single data value

List elements can be other data structures, e.g., vectors and matrices:

```
plan <- list(factory=factory, available=available, output=output)
plan$output
```

```
## trucks   cars
##     20     10
```

Internally, a dataframe is basically a list of vectors

Structures of Structures (cont'd.)

List elements can even be other lists
which may contain other data structures
including other lists
which may contain other data structures…

This **recursion** lets us build arbitrarily complicated data structures from the basic ones

Most complicated objects are (usually) lists of data structures

Example: Eigenstuff

`eigen()` finds eigenvalues and eigenvectors of a matrix
Returns a list of a vector (the eigenvalues) and a matrix (the eigenvectors)

```
eigen(factory)
```

```
## eigen() decomposition
## $values
## [1] 41.556171  1.443829
##
## $vectors
##            [,1]       [,2]
## [1,] 0.99966383 -0.8412758
## [2,] 0.02592747  0.5406062
```

```
class(eigen(factory))
```

```
## [1] "eigen"
```

With complicated objects, you can access parts of parts (of parts...)

```
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
```

Summary

- Arrays add multi-dimensional structure to vectors
- Matrices act like you'd hope they would
- Lists let us combine different types of data
- Dataframes are hybrids of matrices and lists, for classic tabular data
- Recursion lets us build complicated data structures out of the simpler ones