Data Types,
Vectors, and Subsetting

Statistician's perspective

- Think in terms of variables – an ordered collection of measurements on a group of subjects
- Care about the kind of measuremet values: it informs the type of analysis we might perform, e.g., it makes sense to compute the mean/ median of numeric values, but not categorical values
- Care about missing data – we adjust our analyses depending on the amount and kind of missingness

Data Types

- R has a number of built-in data types. The three most basic types are numeric, character, and logical.
- You can check the type using the class function.

```
> class(3.5)
[1] "numeric"
> class("Hello")
[1] "character"
> Class(TRUE)
[1] "logical"
```

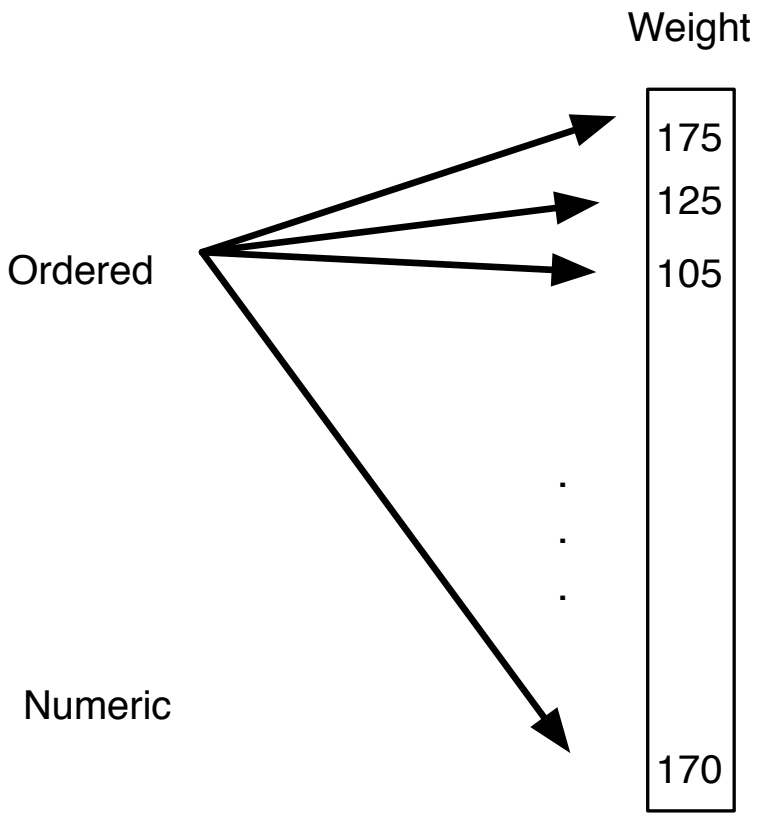
- Another important type is factor

Data Types

- Actually, the types are numeric, character, and logical ***vectors***. There's no such thing as a scalar in R, just a vector of length one.

Vectors

- Ordered container
- Primitive elements of the same type



Vectors

- We have data on a 14-member family – vectors of first names, age, gender, weight, height, whether or not they are over weight (BMI above 25).
- What are the data types?

First Names and Age

```
> fnames
 [1] "Tom" "May" "Joe" "Bob" "Sue" "Liz" "Jon" "Sal"
 [9] "Tim" "Tom" "Ann" "Dan" "Art" "Zoe"
> class(fnames)
[1] "character”
> fage
 [1] 77 33 79 47 27 33 67 52 59 27 55 24 46 48
> class(fage)
[1] "integer"
```

Gender & Over Weight

```
> fgender
 [1] m f m m f f m f m m f m m f
Levels: m f
> class(fgender)
[1] "factor"
> foverWt
 [1]  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE
[8]  FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE
> class(foverWt)
[1] "logical"
```

More on Data Types

- A *logical* vector contains values that are either TRUE or FALSE.
- A *factor* vector is a special storage class used for qualitative data. The values are internally stored as integers by each integer corresponds to a *level*, which is a character string

```
> levels(gender)
[1] "m" "f"
```

Special Values

- The missing value symbol is NA
- It stands for “Not Available”
- NA can be an element of a vector of any type
- NA is different from the character string “NA”
- You can check for the presence of NA values using the **is.na()** function.

Special Values

- Other special values are NaN, for "not a number," which typically arises when you try to compute an indeterminate form such as 0/0.

```
> 0/0
[1] NaN
```

- The result of dividing a non-zero number by zero is `Inf` (or `-Inf`).

```
> 12/0
[1] Inf
```

Special Values

- NULL is a special value that denotes an empty vector

> **names(fweight)**

NULL

- Here we asked for the names of the elements of the vector fweight. The function names returns a character vector of element names. Since this vector has no element names, the return value is a NULL vector

Finding out more information

- Retrieve the number of elements in the vector
- Examine the first 6 elements in the vector
- Elements can have names – height has names
- Are any of the elements in the vector missing?

> **length(fweight)**

[1] 14

> **head(fweight)**

[1] 175 125 185 156 105 190

> **names(fheight)**

[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n"

> **is.na(fweight)**

[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE …

Finding out more information

- Aggregator functions operate on the elements of the vector
- Functions can tell us the about the data type
- Check if a vector is empty
- Convert a vector to a specified data type

```
> min(fweight)
[1] 105
> is.logical(fweight)
[1] FALSE
> is.null(fheight)
[1] FALSE
> as.numeric(fgender)
 [1] 1 2 1 1 2 2 1 2 1 1 2 1 1 2
```

How to manage variables in the workspace

- Give names of all variables
- Remove one or more variables
- Save objects for future use
- Restore saved variables
- Save an entire workspace, and it will automatically load when you start R again

> **objects()**

[1] "age" "bmi" "desiredWt" …

> **rm(x)**

> **save(age, bmi, desiredWt, weight, height, gender, file="cdc200.rda")**

> **load("cdc200.rda")**

> **q()**

Save workspace image? [y/n/c]:

BUT IT KEEPS EVERYTHING!!

Subsetting

Suppose we want the:

- BMI of the 10^{th} person in the family

```
> fbmi[10]
```

Subset by position

```
[1] 30.04911
```

- Ages of all but the first person in the family

```
> fage[-1]
 [1] 33 79 47 27 33 67 52 59 27 55 24 46 48
```

Subset by exclusion

Suppose we want the:

- Height of person “j”

```
> fheight["j"]
 j
71
```

Subset by name

- Genders of the family members who are over weight

```
> fgender[foverWt]
[1] m f m m m f
Levels: m f
```

Subset by logical

Assign values to elements of a vector

- In general, the same indexing may be used to *assign* values to elements of a vector.
- Make sure the vector exists first, or you will get an error.

Assign values to elements of a vector

Can you guess what fheight will look like after each of the following lines?

```
> fheight
 a  b  c  d  e  f  g  h  i  j  k  l  m n
70 64 73 67 64 68 68 65 68 71 67 66 66 62

fheight[2] = 61          # By inclusion
fheight[-13] = 62        # By exclusion
fheight["e"] = 67        # By name
fheight[overWt] = NA     # By logical
fheight[] = 70           # No index
fheight = 70             # Watch out!
```

```
a  b  c  d  e  f  g  h  i  j  k  l  m  n
70 64 73 67 64 68 68 65 68 71 67 66 66 62
fheight[2] = 61
a  b  c  d  e  f  g  h  i  j  k  l  m  n
70 61 73 67 64 68 68 65 68 71 67 66 66 62
fheight[-13] = 62
a  b  c  d  e  f  g  h  i  j  k  l  m  n
62 62 62 62 62 62 62 62 62 62 62 62 66 62
fheight["e"] = 67
a  b  c  d  e  f  g  h  i  j  k  l  m  n
62 62 62 62 67 62 62 62 62 62 62 62 66 62
```

```
T  F  F  F  F  T  T  F T  T  T F  F  F
fheight[foverWt] = NA
a  b  c  d  e  f  g  h i  j  k l  m n
NA 62 62 62 67 NA NA 62 NA NA NA 62 66 62
fheight[] = 70
a  b  c  d  e  f  g h  i  j  k  l  m n
70 70 70 70 70 70 70 70 70 70 70 70 70 70
fheight = 70
[1] 70
```

Suppose we want the:

- BMI of every other person in the family

Subset using a vector of positions

- Weights of the women in our family

Subset using a logical vector

- Height elements “a”, “c”, “f”

Subset with character vector of element names

- Assign every one in the family the last name of “Smith”

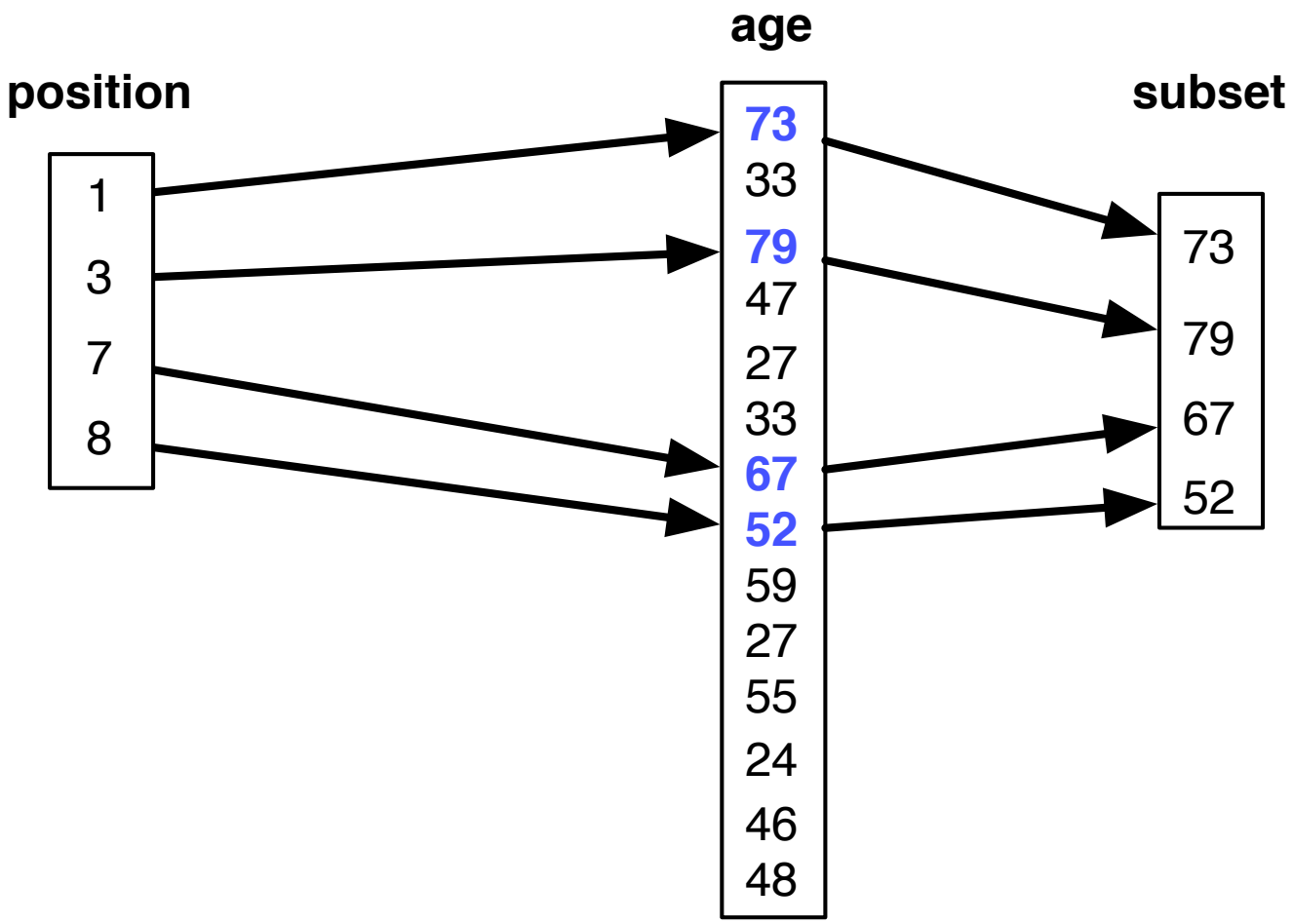
Create an empty vector and assign all elements

We need to better understand:

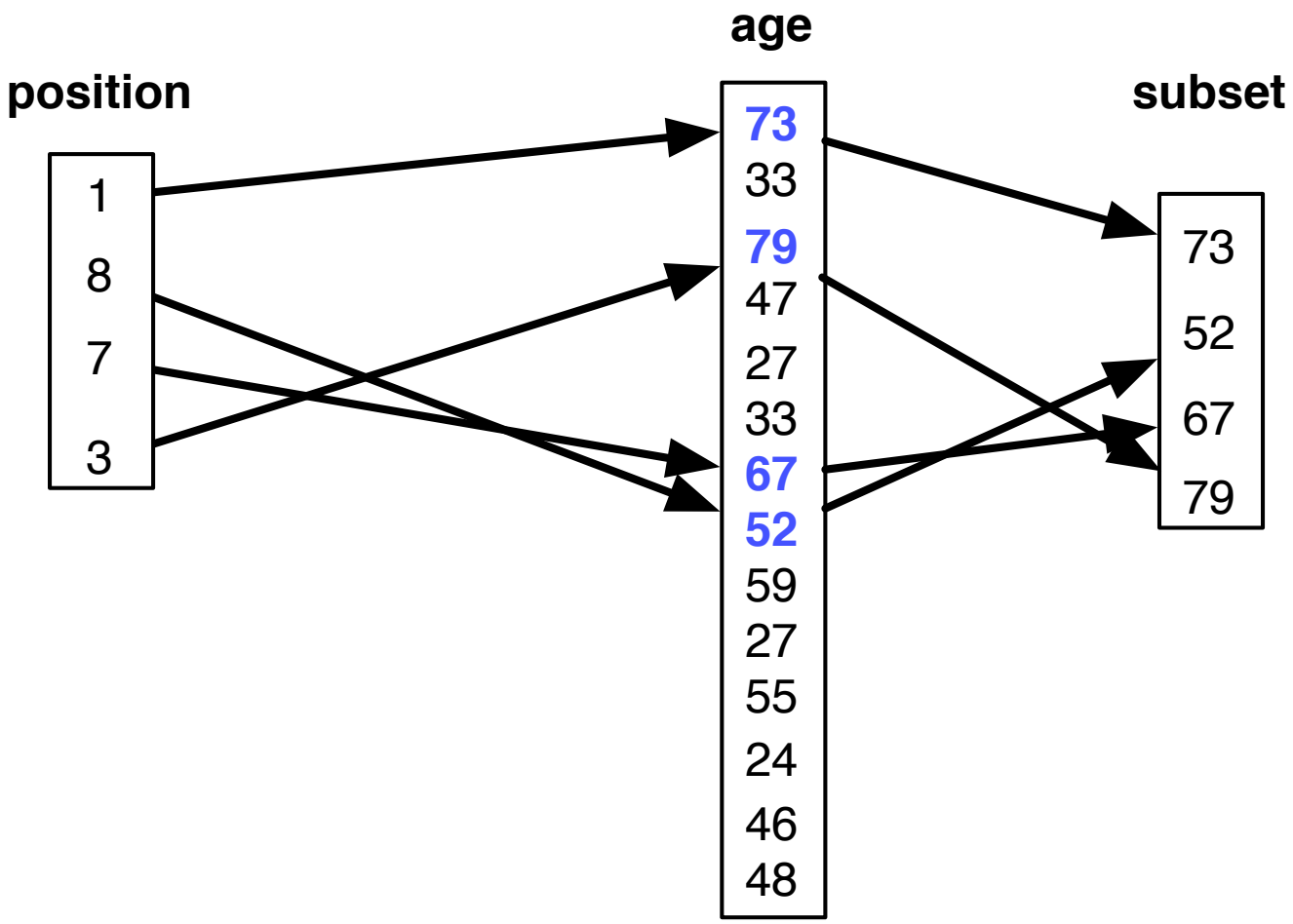
- How to use logical operators to create logical vectors
- How to create vectors with specific numbers and/or letters

Review: subsetting vectors

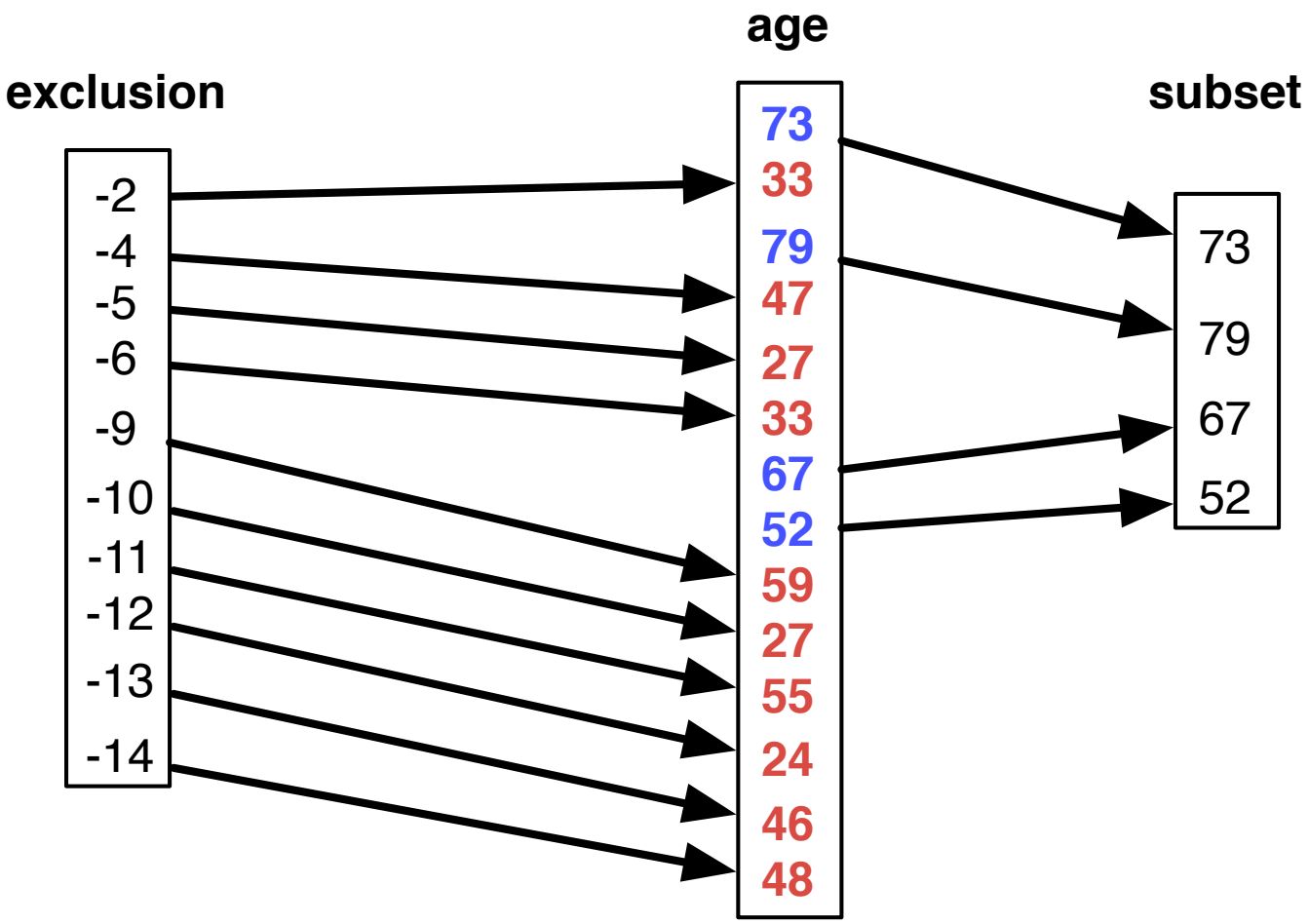
Subset by position



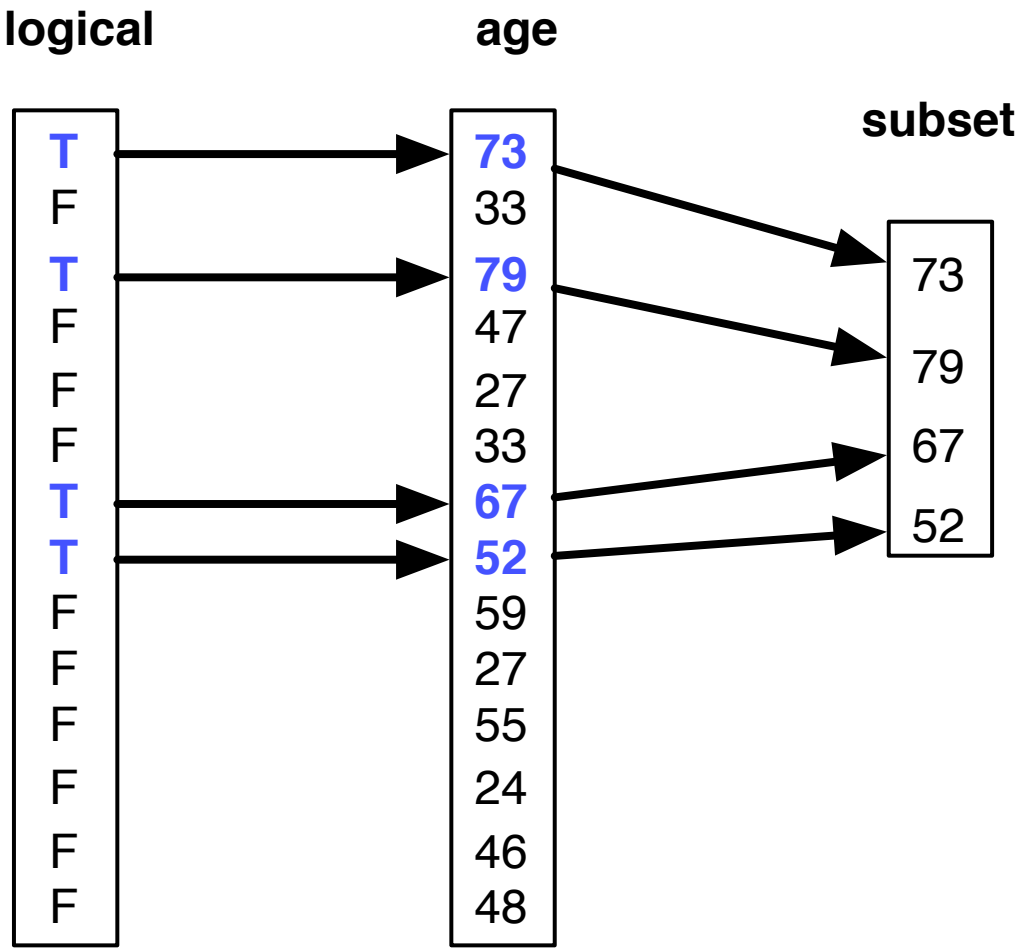
Subset by position



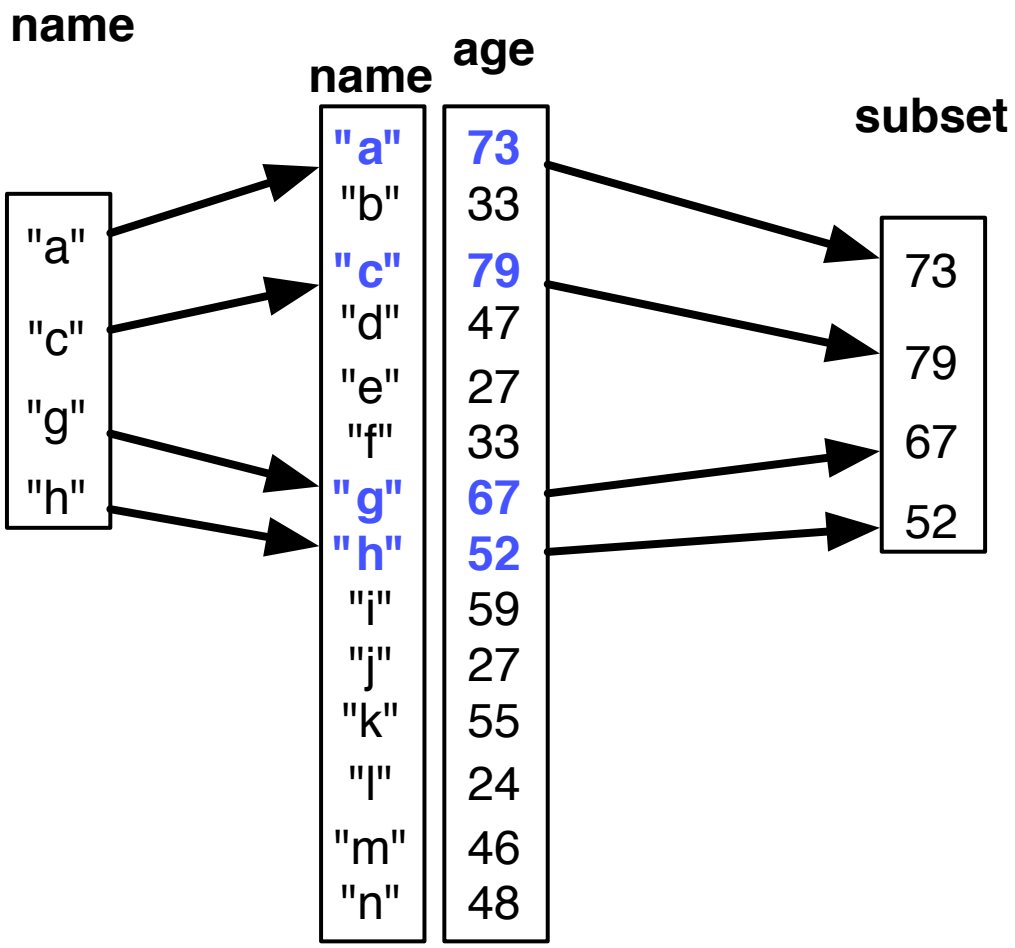
Subset by exclusion



Subset by logical



Subset by name



Five ways to subset a vector

- Position – indices of the element you want
- Exclusion – indices of elements to exclude
- Logical – logical vector the same length as the vector being subset. Keep the elements corresponding to TRUE.
- Name – character vector of names of elements to keep. Vector being subsetted must have names associated with elements
- All – all the elements

Logical Operations

Logical/Relational Operators

- In addition to operators such as +, -, *, and / R also has logical operators
- They are relational operators
  >, <, >=, <=, !=, and ==
- These return a value of TRUE or FALSE
- They are also vectorized operations

Examples

```
> 4 < 3
[1] FALSE
> "a" == "A"
[1] FALSE
> "A" == "A"
[1] TRUE
> 4 != 3
[1] TRUE
```

```
> fweight > 150
[1] TRUE FALSE  TRUE  TRUE FALSE  TRUE  TRUE FALSE
[9] TRUE  TRUE TRUE FALSE FALSE FALSE
> fgender !="m"
[1] FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE  TRUE
[9] FALSE FALSE TRUE FALSE FALSE  TRUE
> fbmi
 [1] 25.16239 21.50106 24.45884 24.48414 18.06089
 [6] 28.94981 28.18797 20.67783 26.66430 30.04911
[11] 26.05364 22.64384 24.26126 22.91060
> fbmi == 25.16239
[1] FALSE FALSE FALSE FALSE FALSE FALSE …
```

Weights of the women in our family

- Create a logical expression that identifies the women in the family

```
> fgender == "f"
[1] FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE
[8] TRUE FALSE FALSE TRUE FALSE FALSE  TRUE
```

- Use this logical expression to subset the vector of fweight

```
> fweight[ fgender == "f"]
[1] 125 105 190 124 166 125
```

Boolean Algebra

- Boolean algebra is a mathematical formalization of the truth or falsity of statements.
- It has three operations, “not,” “or,” and “and.”
- Boolean algebra tells us how to evaluate the truth or falsity of *compound statements* that are built using these operations. For example, if A and B are statements, some compound statements are
- A and B
- (not A) or B

- The “not” operation just causes the statement following it to switch its truth value.

  So not TRUE is FALSE and not FALSE is TRUE.
- The compound statement A and B is TRUE only if both A and B are TRUE.
- The compound statement A or B is TRUE if *either or both* A or B is TRUE.
- In R, we write ! for “not,” & for “and,” and | for “or.” Note: all of these are vectorized!

```
> !(fweight > 150)
[1] FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE
[8] TRUE  FALSE FALSE FALSE  TRUE  TRUE  TRUE
> (fweight > 150) & (fnames == "Tom")
[1]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
[8] FALSE  FALSE  TRUE FALSE FALSE FALSE FALSE
> (fweight > 150) | (fage > 65)
[1]  TRUE FALSE  TRUE  TRUE FALSE  TRUE  TRUE
[8] FALSE   TRUE  TRUE  TRUE FALSE FALSE FALSE
```

Two other functions

- Two other useful functions that operate on logical vectors are **all** and **any**.
- Can you guess what they do?

```
> all(fage > 18)
[1] TRUE
> any(fage < 18)
[1] FALSE
> any(fweight < 150)
[1] TRUE
> all(fweight < 150)
[1] FALSE
```

Examples

- Under 50 `fage < 50`
- Women `fgender == “f”`
- Not over weight `!foverWt`
- Males who are 70 in tall

```
(fgender == “m”)
&(fheight <70)
```

Use logical expressions to obtain the following subsets

- Ages of all non-overweight members of the family

fage[!foverWt]

- Genders of those over 50

fgender[fage > 50]

- BMI of the tallest member of the family

fbmi[fheight == max(fheight)]

Creating vectors

Many functions available

- c() - catenate vectors and values together
- : - create a sequence of values 1 apart
- seq() – create more complex sequences
- rep() – repeat values in a vector
- sort() – sort the values in a vector
- order() – provide the order of values

Let’s show how they work by example

concatenate

```
> c(3, 2, 1)
[1] 3 2 1
> c(2,3,1)
[1] 2 3 1
> x = c(bob =3, alice = 2,
john = 1)
> x
  bob alice  john
    3     2     1
```

- A vector of three numbers, 3, 2, 1, **in that order**
- A **different** vector with the same values in a different order
- Elements in a vector – this time with names

concatenate

```
> c(TRUE, FALSE)
[1]  TRUE FALSE
> c(1.3, 2, 8/3)
[1] 1.300000 2.000000 2.666667
> c("a", "z", "Hello")
[1] "a"     "z"     "Hello"
> y = c(100, 120)
> c(x, y)
  bob alice  john
    3     2     1   100   120
```

- We can use c() to make logical and character vectors
- Notice that the last element determines the number of digits to display
- Character vecotr with 3 elements
- c() can be used to catenate vectors

1:3 returns a numeric vector of 1-apart values

```
> 1:3
[1] 1 2 3
> 4:7
[1] 4 5 6 7
> 10:6
[1] 10  9  8  7  6
> 1.1:5.7
[1] 1.1 2.1 3.1 4.1 5.1
> 5.7:1.1
[1] 5.7 4.7 3.7 2.7 1.7
> 5.7:-1.1
[1]  5.7  4.7  3.7  2.7  1.7  0.7 -0.3
```

rep()

```
> rep(3,2)
[1] 3 3
> x = c(7,1,3)
> rep(x, 2)
[1] 7 1 3 7 1 3
> rep(x, times = 2)
[1] 7 1 3 7 1 3
> rep(x, c(3, 2, 1))
[1] 7 7 7 1 1 3
> rep(x, each = 2)
[1] 7 7 1 1 3 3
```

- Vector of two threes
- Arguments of rep can be vectors
- Repeat the vector 2 times

- Can use the argument name
- When times argument is a vector then each element is repeated individually
- The Each argument

seq() – a richer version of :

```
> seq(1, 5, by = 2)
[1] 1 3 5
> seq(1, 5, length = 3)
[1] 1 3 5
> seq(1, 5, length = 5)
[1] 1 2 3 4 5
> seq(1, length = 5, by = 2)
[1] 1 3 5 7 9
> seq(1, 5, length = 5, by = 2)
Error in seq.default(1, 5,
length = 5, by = 2) : too
many arguments
```

- seq() has several arguments
- from
- to
- by
- length
- There are many ways to call this function

Question:

How could I produce the following vectors (without typing them all out)?

0 0 0 0 0 2 2 2 2 2 4 4 4 4 4 6 6 6 6 6 8 8 8 8 8

rep(seq(0, 8, by = 2), each = 5)

1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5

rep(1:5, times = 5)

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

rep(1:5, times = 5) + rep(0:4, each = 5)

sort() and order()

```
> fage
 [1] 77 33 79 47 27 33 67 52 59 27 55 24 46 48
> sort(fage)
 [1] 24 27 27 33 33 46 47 48 52 55 59 67 77 79
> sort(fage, decreasing = TRUE)
 [1] 79 77 67 59 55 52 48 47 46 33 33 27 27 24
```

sort() and order()

```
> fage
 [1] 77 33 79 47 27 33 67 52 59 27 55 24 46 48
> order(fage)
 [1] 12  5 10  2  6 13  4 14  8 11  9  7  1  3
```

Notice that the return value from order tells us that the 12th element of fage is the smallest, the 5th is the second smallest, ..., and the 3rd is the largest

This function has a decreasing argument too.

Return to our subsets:

- BMI of every other person in the family

bmi[seq(1, 14, by = 2)]

- Weights of the women in our family

fweight[fgender == “f”]

- Height elements “a”, “c”, “f”

fheight[c(“a”, “c”, “f”)]

- Assign every one in the family the last name of “Smith”

```
> lastname = character(length = 14)
> lastname
 [1] "" "" "" "" "" "" "" "" "" "" "" "" "" ""
> lastname[] = "Smith"
> lastname
[1] "Smith" "Smith" "Smith" "Smith" "Smith" "Smith" "Smith" "Smith”
[8] "Smith" "Smith" "Smith" "Smith" "Smith" "Smith" "Smith" "Smith"
```

```
> lname = character()
> lname
character(0)
> lname[1:14] = "Smith"
> lname
 [1] "Smith" "Smith" "Smith" "Smith" "Smith" "Smith" "Smith"
 [8] "Smith" "Smith" "Smith" "Smith" "Smith" "Smith" "Smith"
```

Data Frames

The Family

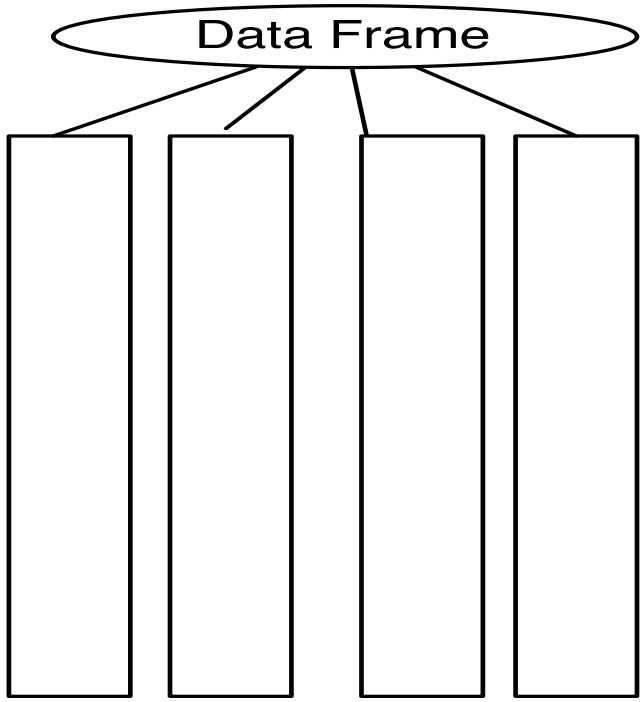
- We have all sorts of information about our family, height, weight, first name, gender, …
- The data frame gives us a way to collect all of these variables (vectors) into one object.

```
> data.frame(firstName = fnames,
gender = fgender, age = fage, height = fheight,
weight = fweight, bmi = fbmi, overWt = foverWt)
```

```
> family
   firstName gender age height weight      bmi overWt
1        Tom      m  77     70    175 25.16239   TRUE
2        May      f  33     64    125 21.50106  FALSE
3        Joe      m  79     73    185 24.45884  FALSE
4        Bob      m  47     67    156 24.48414  FALSE
5        Sue      f  27     64    105 18.06089  FALSE
6        Liz      f  33     68    190 28.94981   TRUE
7        Jon      m  67     68    185 28.18797   TRUE
8        Sal      f  52     65    124 20.67783  FALSE
9        Tim      m  59     68    175 26.66430   TRUE
10       Tom      m  27     71    215 30.04911   TRUE
11       Ann      f  55     67    166 26.05364   TRUE
12       Dan      m  24     66    140 22.64384  FALSE
13       Art      m  46     66    150 24.26126  FALSE
14       Zoe      f  48     62    125 22.91060  FALSE
```

Data Frame

- *Ordered* container of vectors
- Vectors must all be the *same length*
- Vectors can be *different types*



```
> class(family)
[1] "data.frame"
> length(family)      - number of vectors in family
[1] 7
> dim(family)         - number of rows and columns
[1] 14  7
> names(family)       - names of the vectors in family
[1] "firstName" "gender"    "age"       "height"
[5] "weight"    "bmi"       "overWt"
```

dataframe$vector

```
> family$gender
 [1] m f m m f f m f m m f m m f
Levels: m f
> mean(family$height)
[1] 67.07143
> class(family$height)
[1] "numeric"
```

Subsetting Data frames

```
> family[ 10:13, -(3:14)]
   firstName gender
10       Tom      m
11       Ann      f
12       Dan      m
13       Art      m
```

We subset rows and columns of data frames

We subset by **position**, **exclusion**, **logical, name**, and **all**

```
> family[  , c("gender", "firstName") ]
   gender firstName
1       m       Tom
2       f       May
3       m       Joe
4       m       Bob
5       f       Sue
6       f       Liz
7       m       Jon
8       f       Sal
9       m       Tim
10      m       Tom
11      f       Ann
12      m       Dan
13      m       Art
14      f       Zoe
```

Subset rows by **all** and columns by **name**

What's different about the return value?

The order of the columns is different than the order in the data frame. It matches the order of the names

```
> family[family$weight > 180, c("height", "bmi")]
   height      bmi
3      73 24.45884
6      68 28.94981
7      68 28.18797
10     71 30.04911
```

We subset the rows using a **logical** vector

We subset the columns by **name**

dataframe[ ]

```
> family["height"]
   height
1      70
2      64
3      73
4      67
5      64
6      68
7      68
8      65
9      68
10     71
11     67
12     66
13     66
14     62
```

```
> family[ , "height"]
 [1] 70 64 73 67 64 68 68 65 68 71 67 66
66 62
```

What's the difference between these two expressions?

```
> class(family["height"])
[1] "data.frame"
> class(family[, "height"])
[1] "numeric"
```

One returns a data frame and the other returns a vector

Reading data into R

- Many data sets are stored in text files.
- The easiest way to read these into R is using either the `read.table` or `read.csv` function, both of which return a data frame.
- There are quite a few options that can be changed. Some of the important ones are
  - file - name or URL
  - header - are column names at the top of the file?
  - sep - what divides elements of the table
  - na.strings - symbol for missing values, like 9999
  - Skip - number of lines at the top of the file to ignore

Earthquakes Example

- Data from the California Geological Survey

```
> CAquakes = read.table(file = "
http://www.consrv.ca.gov/cgs/rghm/quakes/Documents/
ms49epicenters.txt", header = TRUE)
> dim(CAquakes)
[1] 383   4
> CAquakes[1:3,]
      Date Latitude Longitude   M
1 18001011     36.8    -121.5 5.5
2 18001122     32.9    -117.8 6.3
3 18030000     34.2    -118.1 5.5
> class(CAquakes$Date)
[1] "integer"
```

- How can we extract the years/months/days from the Date column?

Lists

- Data frames are actually a special kind of *list*.
- Unlike a data frame each element can have a different length.

```
> Ingredients = list(cheese = c("Cheddar", "Swiss"),
+                    meat = c("Ham","Turkey", "Bologna”))
> Ingredients
$cheese
[1] "Cheddar" "Swiss"
$meat
[1] "Ham"     "Turkey"  "Bologna"
```

- Note that the elements are not associated with one another by position, as they were in a given row of a data frame.

Indexing lists

- Lists can be indexed by name, using $.
- They can also be indexed like vectors, using []. The result will be another list of length 1.

```
> Ingredients[2]
$meat
[1] "Ham"     "Turkey"  "Bologna”
> class(Ingredients[2])
[1] "list”
```

Indexing lists

- To extract individual elements of a list, enclose the index in [[]]. The result will be coerced to a simpler structure, depending on the element.

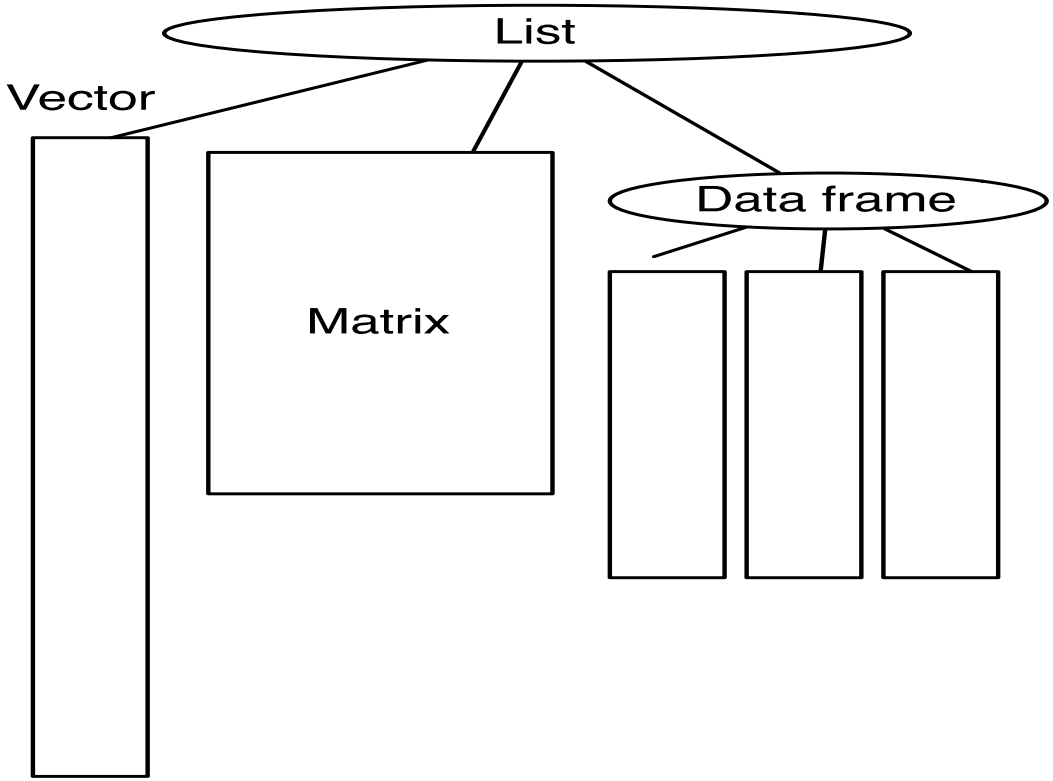
```
> Ingredients[[2]]
[1] "Ham"     "Turkey"  "Bologna"
> class(Ingredients[[2]])
[1] "character”
```

- You will often encounter lists as return values of function calls in R.

```
> x = 1:100
> y = x * 3 + rnorm(100)
> regression.results = lm(y~x) # Regress y on x
> is.list(regression.results)
[1] TRUE
> names(regression.results)
 [1] "coefficients"  "residuals"     "effects"
 [4] "rank"          "fitted.values" "assign"
 [7] "qr"            "df.residual"   "xlevels"
[10] "call"          "terms"         "model"
> regression.results$coef # Note partial matching
(Intercept)           x
  0.2433211   2.9950379
```

Lists

- *Ordered* container of objects
- Objects can be anything, vector, data frame, list, etc.



Matrices and Arrays

Matrices and Arrays

- Rectangular collection of elements
- Dimensions are two, three, or more
- Homogeneous primitive elements (e.g. all numeric or all character)

- You can create a matrix in R using the `matrix` function.
- By default, matrices in R are assigned by *column-major* order.
- You can assign them by *row-major* order by setting the `byrow` argument to TRUE. Note that the first argument to `matrix` is a vector, so all elements must be of the same type (numeric, character, or logical).

```
> m = matrix(1:6, nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> m = matrix(1:6, nrow = 2, ncol = 3, byrow = TRUE)
 m
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

- Assign names to the rows and columns of a matrix:

```
> rownames(m) = letters[1:2]
> colnames(m) = letters[1:3]
> m
  a b c
a 1 2 3
b 4 5 6
```

- Find the dimensions of a matrix:

```
> dim(m); nrow(m); ncol(m)
[1] 2 3
[1] 2
[1] 3
```

- Exchange rows and columns:

```
> t(m)  # t for transpose
  a b
a 1 4
b 2 5
c 3 6
```

- To index elements of a matrix, use the same five methods of indexing we covered for vectors, but with the first index for rows and the second for columns.

- Aside: by default the result is coerced to a vector if possible, rather than a matrix with a single row or column. To override, use `drop = FALSE`.

- What will each line return?

```
> m
  a b c
a 1 3 5
b 2 4 6
> m[-1, 2]              # Exclusion & inclusion by position
> m["a",]                  # By name, empty column index
> m[, c(TRUE, TRUE, FALSE)]   # Empty row index, logical
```

Summary of Data Structures

Types of structures

- To summarize, the data structures we have encountered so far are:
  - vector
  - data frame
  - list
  - matrix
- Matrices and arrays are actually just stored as vectors with shape information, so our discussions of "vectorized" calculations hold for matrices as well.
- This is NOT true for lists and data frames.

Indexing data structures

- Vectors: [index]

```
> x[1:10]; x[-3]; x[x>3]
```

- Data frames: [rowindex, colindex], $name

```
> family$weight; family[,3:4];
family[family$height > 70, ]
```

- Lists: $name, [index], [[index]]

```
> Ingredients$meat; Ingredients[1:2];
Ingredients[[1]]
```

- Matrices: [rowindex, colindex]

```
> m[1,2]; m[1:2, ]; m[ ,"a"]
```

- Note: both $ and [[]] can index only *one* element.