Section 4

Data, Data Structures & Data Manipulation

Data Structures & Dimensionality

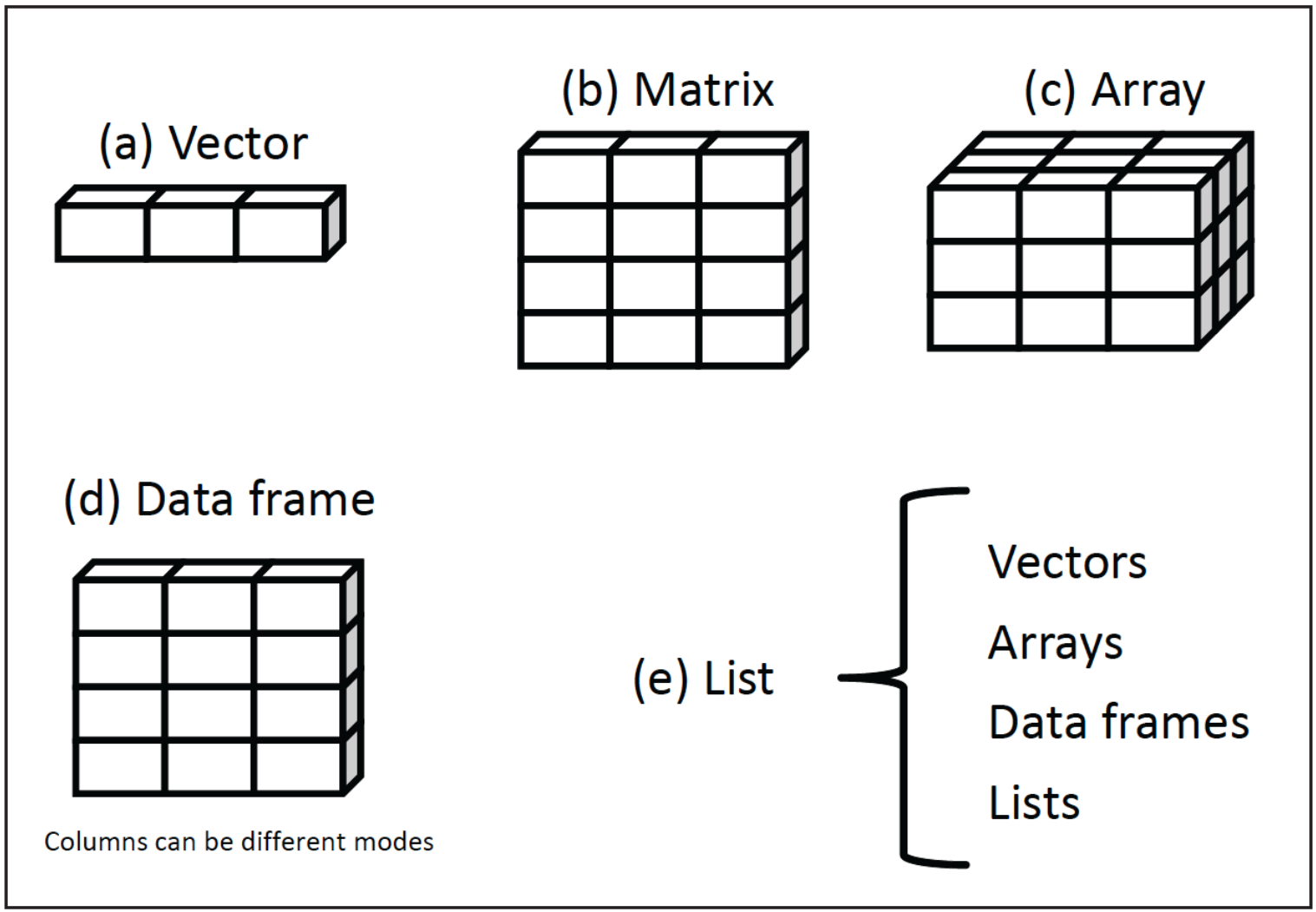
Dimension	Homogeneous	Heterogeneous
1	Atomic Vector	List
2	Matrix	Data Frame
n	Array	

Homogeneous All contents must be of the same type

Heterogeneous Contents can be of different types

n.b. There are no 0-dimensional (scalar) types in R, only vectors of length one

Data Structures



Vectors

- The basic data structure in R is the vector
- There two types of vectors: `atomic vectors` and `lists`

Properties of Vectors

- Type (`typeof()`)
- Length (`length()`)

n.b. Use `is.atomic()` or `is.list()` to determine if an object is a vector, **not** `is.vector()`

Atomic Vectors

Four Common Types of Vectors

- Logical
- Integer
- Double (numeric)
- Character

```
> doubleAtomicVector <- c(1, 3.14, 99.999)

# use L prefix to get integers instead of doubles
> integerAtomicVector <- c(1L, 3L, 19L)

> logicalAtomicVector <- c(TRUE, FALSE, T, F)

> characterAtomicVector <- c("this", "is a", "string")
```

TRY THIS

1. Create the vector `myFavNum` of you favorite **fractional** number
2. Create the vector `myNums` of your seven favorite numbers
3. Create the vector `firstNames` of the first names of two people next to you
4. Create the vector `myVec` of the last name and age of someone you know

TRY THIS TOO

1. Guess and then check what types your vectors are.
2. Check the length of each vector.
3. Did you write the code in the console window or the editor?
4. How do you execute a line of code in the editor?
5. How do you execute multiple lines of code simultaneously in the editor?
6. Did you leverage the `TAB` button for auto-completion?

Accessing Elements of a Vector

```
> (myAtomicVector <- c(1, 2, 3, 4, -99, 5, NA, 4, 22.223))
[1]   1.000   2.000   3.000   4.000 -99.000   5.000      NA
[8]   4.000  22.223

> myAtomicVector[5]
[1] -99

> myAtomicVector[c(1, 2, 5, 9)]
[1]   1.000   2.000 -99.000  22.223

> myAtomicVector[10]
[1] NA

> myAtomicVector[3:8]
[1]   3   4 -99   5  NA   4
```

TRY THIS

1. Add `myFavNum` to the seventh entry of `myNums` and store the result in a variable named `myFirstAddition`
2. Add `myFavNum` to each of the seven entries of `myNums` and store the result in a variable named `mySecondAddition`
3. Add `myFavNum` to **all** of the values in `myNums` and store the result in a variable named `myFirstSum`
4. Add `myFavNum` to the smallest number in `myNums` and store the result in a variable named `thisIsGettingMoreComplex`
5. Add the second entry of `myNums` to the age of the person you select for `myVec` and store the result in a variable named `whatTypeOfVectorIsThis`
    - Does what we did make sense? Did it work? Why?

PREAMBLE

```
myFavNum <- 3.1415
myNums <- c(1, 3, 55, 33, 86, -sqrt(2), -110)
# also works myNums <- 1:7
firstNames <- c("Jeff", "Terence", "David")
myVec <- c("Parr", 99)
```

SOLUTION

1. ```
   myFirstAddition <- myFavNum + myNums[7]
   ```

2. ```
   mySecondAddition <- myFavNum + myNums
   ```

3. ```
   myFirstSum <- myFavNum + sum(myNums)
   ```

4. ```
   thisIsGettingMoreComplex <- myFavNum + min(myNums)
   ```

5. ```
   whatTypeOfVectorIsThis <- sum(c(myNums[2], myVec[2]))
   Error in sum(c(myNums[2], myVec[2])) :
     invalid 'type' (character) of argument
   ```
   ```

Missing Values

Missing values are specified with `NA`, which is a logical vector of length one.

- `NA` will always be coerced to the correct type if used inside `c()`
- You can create `NA`s of a specific type with
  - `NA_real_` (double)
  - `NA_integer_`
  - `NA_character_`

```
> c(1, 2, 3, NA)
[1]  1  2  3 NA

> x <- c(1, 2, 3, NA)

> typeof(x)
[1] "double"
```

```
> x[1]
[1] 1

> x[4]
[1] NA

> typeof(x[4])
[1] "double"
```

n.b. `NA` and `NULL` are **NOT** the same

na.rm = TRUE

- Certain functions will fail when applied to vectors with one or more NAs

```
> (myAtomicVector_01 <- c(99.1, 98.2, 97.3, 96.4, NA))
[1] 99.1 98.2 97.3 96.4   NA

> sum(myAtomicVector_01)
[1] NA

> mean(myAtomicVector_01)
[1] NA

> sum(myAtomicVector_01, na.rm = TRUE)
[1] 391

> mean(myAtomicVector_01, na.rm = TRUE)
[1] 97.75
```

Types & Tests

To check the type of a vector, use `typeof()`, or more specifically

- `is.character()`
- `is.double()`
- `is.integer()`
- `is.logical()`
- `is.na()`

Coercion

Coercion is a great feature in R which can make coding easy, but may also have unintended consequences.

- All elements in an atomic vector must be the same type
- If you attempt to combine different types in an atomic vector they will be coerced to the most flexible type
- Most to least flexible types: character, double, integer, logical
- When a logical vector is coerced to numeric (double or integer), `TRUE = 1` and `FALSE = 0`

```
> x <- c("abc", 123)

> typeof(x)
[1] "character"
```

You can explicitly coerce using `as.character()`, `as.double()`, `as.integer()`, and `as.logical()`

A Brief Digression: `str()` and `glimpse()`

- A quick way to figure out what data structure an object is composed of is to use `str()`, which is short for structure
- `str()` provides a concise description for any R data structure
- Using the `tibble` package, `glimpse()` gives you a cleaner, easier-to-read view of your data
- Using `population` data from tidyr package

```
> str(population)
Classes tbl_df, tbl and 'data.frame': 4060 obs. of  3 variables:
 $ country   : chr  "Afghanistan" "Afghanistan" "Afghanistan" "Afghanistan" ...
 $ year      : int  1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 ...
 $ population: int  17586073 18415307 19021226 19496836 19987071 20595360 ...

> tibble::glimpse(population)
Observations: 4,060
Variables: 3
$ country    <chr> "Afghanistan", "Afghanistan", "Afghanistan", "Afghanistan"...
$ year       <int> 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004...
$ population <int> 17586073, 18415307, 19021226, 19496836, 19987071, 20595360...
```

Subsetting Atomic Vectors

Let's create a default atomic vector

```
> (myAtomicVector_01 <- c(99.1, 98.2, 97.3, 96.4))
[1] 99.1 98.2 97.3 96.4

> str(myAtomicVector_01)
 num [1:4] 99.1 98.2 97.3 96.4

> class(myAtomicVector_01)
[1] "numeric"

> is.atomic(myAtomicVector_01)
[1] TRUE

> typeof(myAtomicVector_01)
[1] "double"
```

n.b. The number after the decimal point gives the original position in the vector

How to Subset an Atomic Vector

Positive integers return elements at the specified positions

```
> (myAtomicVector_01 <- c(99.1, 98.2, 97.3, 96.4))
[1] 99.1 98.2 97.3 96.4

> myAtomicVector_01[c(1, 3)]
[1] 99.1 97.3

# real numbers are automatically truncated
> myAtomicVector_01[c(1.999, 3.001)]
[1] 99.1 97.3

> order(myAtomicVector_01)
[1] 4 3 2 1

> myAtomicVector_01[order(myAtomicVector_01)]
[1] 96.4 97.3 98.2 99.1
```

How to Subset an Atomic Vector

Negative integers omit elements at specified positions

```
> (myAtomicVector_01 <- c(99.1, 98.2, 97.3, 96.4))
[1] 99.1 98.2 97.3 96.4

> myAtomicVector_01[-c(1, 3)]
[1] 98.2 96.4

> myAtomicVector_01[c(-1, -3)]
[1] 98.2 96.4

> myAtomicVector_01[-c(-1, -3)]
[1] 99.1 97.3

> myAtomicVector_01[c(-1,- 3.99)]
[1] 98.2 96.4
```

How to Subset an Atomic Vector

Logical vectors select elements where the logical value is TRUE

```
> myAtomicVector_01 <- c(99.1, 98.2, 97.3, 96.4))
[1] 99.1 98.2 97.3 96.4

> myAtomicVector_01[c(TRUE, TRUE, FALSE, FALSE)]
[1] 99.1 98.2

> myAtomicVector_01[c(T, TRUE, F, FALSE)]
[1] 99.1 98.2

> myAtomicVector_01[c(T, T, F, F)]
[1] 99.1 98.2

> myAtomicVector_01 > 98
[1]  TRUE  TRUE FALSE FALSE

> myAtomicVector_01[myAtomicVector_01 > 98]
[1] 99.1 98.2
```

How to Subset an Atomic Vector

Logical vectors can be quirky at times

- If the logical vector is shorter than the vector being subsetted, it will be recycled to be the same length

```
# equivalent to myAtomicVector_01[c(T, F, T, F)]
> myAtomicVector_01[c(T, F)]
[1] 99.1 97.3

> myAtomicVector_01[c(T, F, F)]
[1] 99.1 96.4
```

- A missing value in the index always yields a missing value in the output

```
> myAtomicVector_01[c(T, F, NA, T)]
[1] 99.1   NA 96.4
```

How to Subset an Atomic Vector

Character vectors can can be employed to return elements with matching names **if the vector is named**

```
> (myAtomicVector_01 <- c(99.1, 98.2, 97.3, 96.4))
[1] 99.1 98.2 97.3 96.4

> names(myAtomicVector_01) <- letters[1:4]

> myAtomicVector_01
   a    b    c    d
99.1 98.2 97.3 96.4

> myAtomicVector_01[c("a", "d")]
   a    d
99.1 96.4
```

n.b. partial character string matches will not work, e.g., if element name is `abc`, then `myAtomicVector_01["ab"]` will return `NA`

Conditionally Subsetting Atomic Vectors

- The syntax is awkward and takes some time to get used to
- Once you understand the sequence of events in conditional subsetting, it will feel more natural

```
> (myAtomicVector_01 <- c(99.1, 98.2, 97.3, 96.4))
[1] 99.1 98.2 97.3 96.4

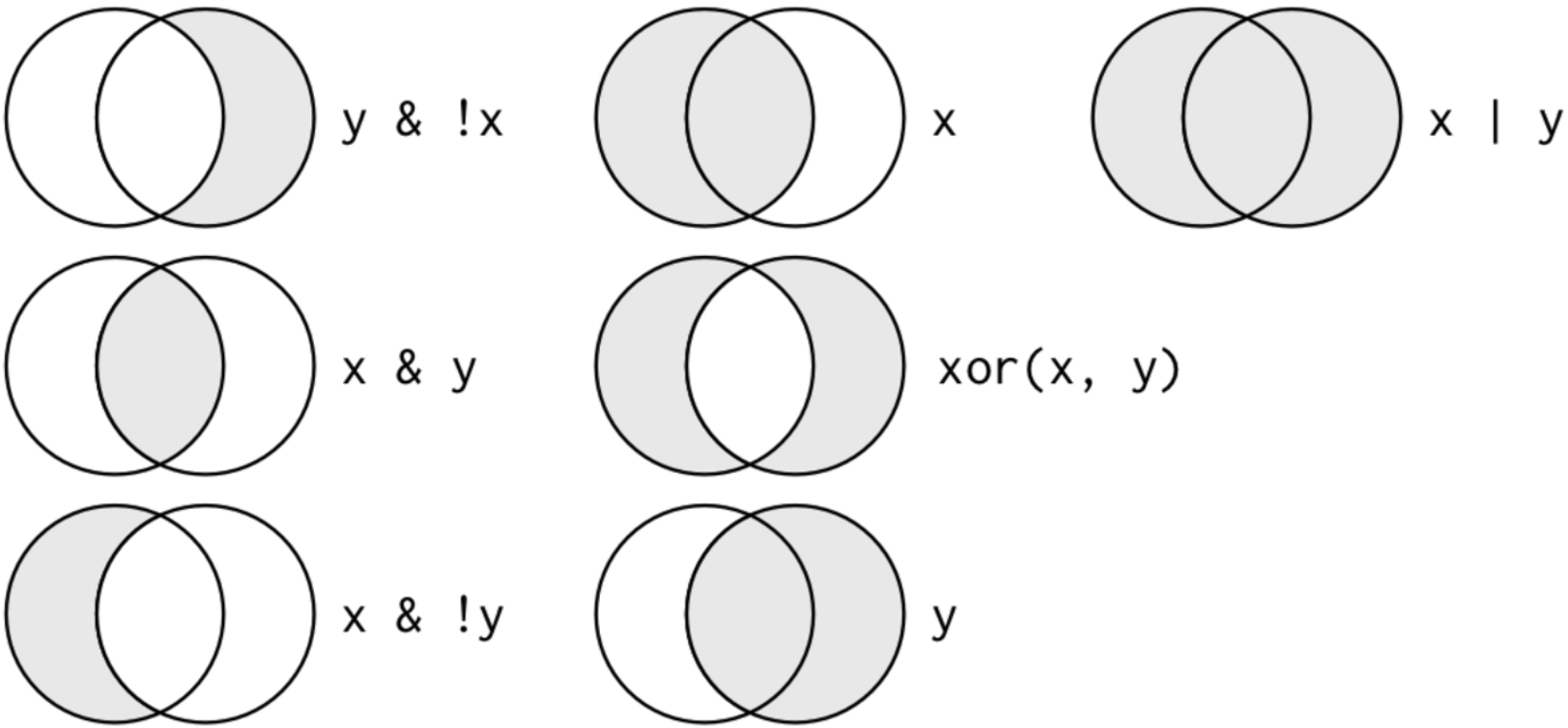
> myAtomicVector_01[myAtomicVector_01 > 98]
[1] 99.1 98.2
```

- What is actually happening
  1. The `myAtomicVector_01 > 98` part of the statement tests each element of the vector to see whether it is > 98 and returns a `LOGICAL` value for each test which, in this case, returns the logical vector (`T T F F`)
  2. The vector (`T T F F`) is passed to `myAtomicVector_01`, which returns the first two elements and omits the final two
     - An equivalent statement would be `myAtomicVector_01[c(T, T, F, F)]`

Logical Operators

Operator	Description
`<`	Less than
`<=`	Less than or equal to
`>`	Greater than
`>=`	Greater than or equal to
`==`	Exactly equal to
`!=`	Not equal to
`!x`	Not x
`x | y`	x or y
`x & y`	x and y
`isTRUE(x)`	Test if x is `TRUE`

Logical Operators



PREAMBLE

```
> myAtomicVector <- c(1, 4, 3, 2, NA, 3.22, -44, 2, NA, 0, 22, 34)
```

TRY THIS

1. How many positive numbers (> 0) are there in this vector?
2. How many negative numbers (< 0) are there in this vector?
3. How many 0's are there in this vector?
4. How many `NA`s are there is this vector?
5. How many numbers in the vector are non-zero **and** not `NA`s?
6. What is the sum of the positive numbers is this vector?
7. What is the sum of the negative numbers is this vector?

SOLUTION

1. ```
   sum(myAtomicVector > 0, na.rm = T)
   ```
2. ```
   sum(myAtomicVector < 0, na.rm = T)
   ```
3. ```
   sum(myAtomicVector == 0, na.rm = T)
   ```
4. ```
   sum(is.na(myAtomicVector))
   ```
5. ```
   sum(myAtomicVector != 0 & !is.na(myAtomicVector), na.rm = T)
   # there is a simpler solution to this question
   ```
6. ```
   sum(myAtomicVector[myAtomicVector > 0], na.rm = T)
   ```
7. ```
   sum(myAtomicVector[myAtomicVector < 0], na.rm = T)
   ```
   ```

Lists

- Lists are different from atomic vectors as elements of a list can be of any type, including lists
- A list is constructed using `list()` instead of `c()`

```
> myList <- list(aA = 10:12, bB = "abc", cC = c(3.1415, 9), dD = c(T, F, F, F))

> str(myList)
List of 4
 $ aA: int [1:3] 10 11 12
 $ bB: chr "abc"
 $ cC: num [1:2] 3.14 9
 $ dD: logi [1:4] TRUE FALSE FALSE FALSE
```

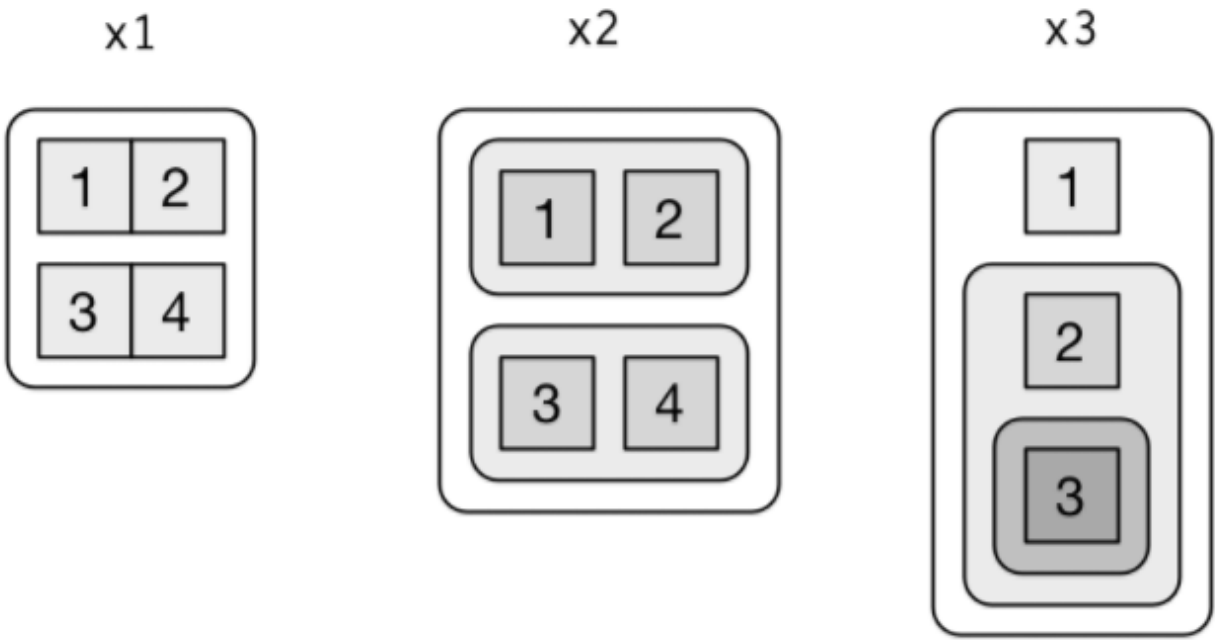
- Lists are recursive, i.e., a list can contain lists, making them fundamentally different from atomic vectors
- `is.list()` (test if list), `as.list()` (coerce to list), `unlist()` (convert to atomic vector + coercion)

Accessing List Elements [RDS, Wickham, 2017]

```
> x1 <- list(c(1, 2), c(3, 4))

> x2 <- list(list(1, 2), list(3, 4))

> x3 <- list(1, list(2, list(3)))
```



n.b. Lists have rounded corners, atomic vectors have square corners

Accessing List Elements [CONT'D]

- Using [accesses a sub-list, i.e., the result is a list (type preservation)

```
> x1 <- list(c(1, 2), c(3, 4))

> x2 <- list(list(1, 2), list(3, 4))

> str(x1[2])
List of 1
 $ : num [1:2] 3 4

> str(x2[2])
List of 1
 $ :List of 2
  ..$ : num 3
  ..$ : num 4
```

Accessing List Elements [CONT'D]

- Using `[[` extracts elements of a list (type simplification)

```
> str(x1[2]) # generates a list
List of 1
 $ : num [1:2] 3 4

> str(x1[[2]]) # generates a vector
 num [1:2] 3 4

> str(x1[[2]][1]) # generates a singleton vector
 num 3

> str(x1[2][1]) # generates a list
List of 1
 $ : num [1:2] 3 4

> str(x1[2][[1]]) # generates a vector
 num [1:2] 3 4

> str(x1[[2]][[1]]) # generates a singleton vector
 num 3
```

Why/When to Use Lists

- Lists are a catch all, and can therefore be used for just about anything (applications are almost infinite)
- Two important and frequently-used applications of lists are
  1. Growing vectors to an unknown size, often in a `for` loop
  2. With `JSON` data

Lists: Growing Vectors to an Unknown Size

`SCEANRIO:` Imagine you are collecting user IP addresses across 100 websites, trying to figure out how many users pinged each website on a given day, and how many website pings there were in total

`OBJECTIVE:` Create a vector of IP addresses that pinged all 100 websites

- Clearly this number will change on a daily basis
- In this scenario you cannot preallocate the size of your vector as it is unknown
- Why is size preallocation important when coding?
- What happens when you don't preallocate?

Lists: Growing Vectors to an Unknown Size [CONT'D]

- What does the following code do?

n.b. For ease of exposition, instead of using IP addresses `doubles` are stored

```
set.seed(10)

means <- runif(100, 0, 10)

output <- double()

system.time(
for (i in seq_along(means)){

  n <- sample(1000000, 1)
  output <- c(output, rnorm(n, means[i]))
}
)
```

- Use the above code and note the execution time on your machine

PREAMBLE

```
set.seed(10)
means <- runif(100, 0, 10)
```

TRY THIS

Using a `for` loop and a `list`, generate the same vector as the previous example, timing the code

PREAMBLE

```
set.seed(10)
means <- runif(100, 0, 10)
```

TRY THIS

Using a `for` loop and a `list`, generate the same vector as the previous example, timing the code

SOLUTION

```
set.seed(10)
means <- runif(100, 0, 10)

out <- list()

system.time({
  for (i in seq_along(means)){
    n <- sample(1000000, 1)
    out[[i]] <- rnorm(n, means[i])
  }
  out <- unlist(out)
})
```

Lists: JSON Data

- JSON stands for JavaScript Object Notation
- JSON data is a list and is
  - light-weight
  - language-independent
  - easy to read and write
  - text-based, human readable data exchange format
- There are many packages in R that deal with JSON data including jsonlite, rjson, RJSONIO, tidyjson, etc.

Lists: JSON Data [CONT'D]

JSON data that stores key/value pairs of information about people might look like this

```
[{
  "first_name": "Laverna",
  "last_name": "Marousek",
  "email": "lmarousek0@pagesperso-orange.fr",
  "gender": "Female",
  "ip_address": "136.180.44.253"
}, {
  "first_name": "Merrill",
  "last_name": "Matuska",
  "email": "mmatuska1@businessinsider.com",
  "gender": "Female",
  "ip_address": "129.25.248.180"
}]
```

Lists: JSON Data [CONT'D]

JSON data may also be nested

```
[{
  "first_name": "Steffen",
  "last_name": "Biggs",
  "child": [
    {
      "gender": "Female",
      "name": "Alysa Terbeck"
    }
  ]
}, {
  "first_name": "Ilyssa",
  "last_name": "Padefield",
  "child": [
    {
      "gender": "Male",
      "name": "Haslett Rehor"
    },
    {
      "gender": "Male",
      "name": "Oliy McCambrois"
    }
  ]
}]
```

TRY THIS

Import `json_lab_data.json`, which contains nested data about parents and children

1. How many children exist in this data set?
2. How many female children are there? Male children?

TRY THIS

Import `json_lab_data.json`, which contains nested data about parents and children

1. How many children exist in this data set?
2. How many female children are there? Male children?

SOLUTION

```
> myJSONdata <- jsonlite::fromJSON("~/Desktop/json_lab_data.json")

> flatJSONdata <- tibble::as_data_frame(jsonlite::flatten(myJSONdata))

> unnestedJSONdata <- flatJSONdata %>% tidyr::unnest(child)

# Question 1
> nrow(unnestedJSONdata)

# Question 2
> nrow(dplyr::filter(unnestedJSONdata, gender == "Male"))
> nrow(dplyr::filter(unnestedJSONdata, gender == "Female"))
```

Names

- A name is a vector attribute
- Not all elements of a vector are required to have a name

```
> x <- c(1, 2, 3)
> names(x)
NULL

> x <- c(1, 2, 3)
> names(x) <- c("a", "b", "c")
> names(x)
[1] "a" "b" "c"

> x <- c(a = 1, b = 2, c = 3)
> names(x)
[1] "a" "b" "c"

> x <- c(a = 1, b = 2, 3)
> names(x)
[1] "a" "b" ""
```

Factors

- An important use of attributes is to define factors
- A factor is a vector of elements from a discrete set, and is used to store categorical (ordinal or nominal) data
- Factors are built on top of integer vectors using two attributes
  1. The `class()` `factor`, which makes them behave differently from regular integer vectors
  2. The `levels()`, which defines the discrete set of permissible values

```
> (x <- factor(c("M" ,"F" ,"F" ,"M"))
[1] M F F M
Levels: F M

> class(x)
[1] "factor"

> typeof(x)
[1] "integer"

> str(x)
 Factor w/ 2 levels "F","M": 2 1 1 2
```

```
> levels(x)
[1] "F" "M"

> x[2] <- "c"
Warning message:
...
invalid factor level, NA generated

> x
[1] M    <NA> F    M
Levels: F M
```

Nominal Factors

- Although we (intelligent humans) have an inherent ability to understand the ordering of the ordinal categories below, R does not, and unless told, will treat them as nominal categorical variables
- Nominal (unordered) factors are sorted automatically by R, e.g., alphabetically, numerically, etc.

n.b. The terms *ordered* and *sorted* are **not** synonymous here

Nominal Factors [EXAMPLE]

```
> (bodyType <- factor(c("healthy", "healthy", "healthy", "obese",
     "overweight", "overweight", "skinny"))
[1] healthy    healthy    healthy    obese   overweight   overweight   skinny
Levels: healthy obese overweight skinny

> levels(bodyType)
[1] "healthy"    "obese"      "overweight" "skinny"

> str(bodyType)))
 Factor w/ 4 levels "healthy","obese",..: 1 1 1 2 3 3 4

> bodyType  < "obese"
[1] NA NA NA NA NA NA NA
Warning message:
In Ops.factor(bodyType, "obese") : < not meaningful for factors
```

Nominal Factors [CONT'D]

- Even though nominal factors are ordered due to the underlying integer mapping, logical comparisons based on levels fail

```
> bodyType[bodyType < "obese"]
[1] <NA> <NA> <NA> <NA>
Levels: healthy obese overweight skinny
Warning message:
In Ops.factor(bodyWeight, "obese") : < not meaningful for factors
```

- Nominal factors *can* be filtered if we access the underlying integer mapping, but weird results may arise

```
> levels(bodyType)
[1] "healthy"    "obese"      "overweight" "skinny"

> str(bodyType)
Factor w/ 4 levels "healthy","obese",..: 4 1 3 2

> bodyType[as.integer(bodyType) < 3] # 3 is mapped to "overweight"
[1] healthy obese
Levels: healthy obese overweight skinny
```

Ordinal Factors

- We can create ordinal factors by including the option `ordered = TRUE`
- By creating an ordinal set of factors, we are telling R to explicitly use the ordering we are providing
- Let's examine a messier version of bodyType, where instead of the body type being explicit (e.g., "obese"), the body types are coded for brevity

```
(bodyType <- factor(c("h", "h", "h", "ob", "ov", "ov", "s"),
                    levels = c("s", "h", "ov", "ob"),
                    labels = c("Skinny", "Healthy", "Overweight", "Obese"),
                    ordered = TRUE))
```

Ordinal Factors [CONT'D]

Let's examine exactly what is being executed

```
(bodyType <- factor(c("h", "h", "h", "ob", "ov", "ov", "s"),
                    levels = c("s", "h", "ov", "ob"),
                    labels = c("Skinny", "Healthy", "Overweight", "Obese"),
                    ordered = TRUE))
```

1. `c("h", "h", "h", "ob", "ov", "ov", "s")` is what we want to classify as a factor
2. `levels = c("s", "h", "ov", "ob")` provides the levels
   - Omitting this enables R to order the levels itself
3. `labels = c("Skinny", "Healthy", "Overweight", "Obese")` are the *nice* labels we want to see instead of the more obscurely-coded factors

   n.b. `labels` are mapped directly to `levels`
4. `ordered = TRUE` instructs R to order the factors according to `levels`

Ordinal Factors [EXAMPLE]

```
>  bodyType <- factor(c("h", "h", "h", "ob", "ov", "ov", "s"),
                      levels = c("s", "h", "ov", "ob"),
                      labels = c("Skinny", "Healthy", "Overweight", "Obese"),
                      ordered = TRUE)

> levels(bodyType)
[1] "Skinny"     "Healthy"    "Overweight" "Obese"

> str(bodyType)
 Ord.factor w/ 4 levels "Skinny"<"Healthy"<..: 2 2 2 4 3 3 1

> bodyType  < "Obese"
[1]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE

> bodyType[bodyType  < "Obese"]
[1] Healthy    Healthy    Healthy    Overweight Overweight Skinny
Levels: Skinny < Healthy < Overweight < Obese
```

PREAMBLE

```
myCyl <- mtcars$cyl
```

YOU TRY IT

1. Create an ordered factor from `myCyl`, mapping the levels to 'Small', 'Medium' and 'Large'
2. How many observations have cylinders $<=$ 'Medium'?

PREAMBLE

```
myCyl <- mtcars$cyl
```

YOU TRY IT

1. Create an ordered factor from `myCyl`, mapping the levels to 'Small', 'Medium' and 'Large'
2. How many observations have cylinders $<=$ 'Medium'?

SOLUTION

```
> myCyl <- factor(myCyl, labels = c("Small", "Medium", "Large"), ordered = T)

> length(myCyl[myCyl <= "Medium"])
```

PREAMBLE

```
set.seed(4)
ageData <- round(runif(1000000, 0, 120))
```

YOU TRY IT

Create the following ordered factors associated with ageData: Infant (0-2 yrs); Toddler (3-5 yrs); Child (6-9 yrs); Tween (10-12 yrs); Teenager (13-19 yrs); Adult (20-65 yrs); Senior (66 yrs +).

1. How many minors are there?
2. Create a frequency table that shows the freqeuncy by factor.

PREAMBLE

```
set.seed(4)
ageData <- round(runif(1000000, 0, 120))
```

SOLUTION

```
> ageData_class <- cut(ageData, breaks = c(-Inf, 2, 5, 9, 12, 19, 65, Inf),
    labels = c("infant", "toddler", "child", "tween", "teen", "adult", "senior"),
    levels = c("infant", "toddler", "child", "tween", "teen", "adult", "senior"),
    ordered = T)

> sum(ageData_class < "adult")
[1] 162804

> table(ageData_class)
ageData_class
 infant toddler   child   tween    teen   adult  senior
  20859   24991   33527   24909   58518  382780  454416
```

Matrices and Arrays

- By giving an atomic vector a dimension attribute, it can behave like a multi-dimensional array
- A special case of the array is a matrix, a two-dimensional array
- Matrices and arrays are created with `matrix()` and `array()`

```
> x <- matrix(1:10, ncol = 5, nrow = 2)
%      # can drop ncol and nrow to shorten

> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

```
> (y <- array(1:12, c(2, 3, 2))
, , 1

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12
```

Subsetting Matrices and Arrays

- The most common way to subset matrices and arrays is to supply a one-dimensional index for each dimension, separated by a comma
- A blank index returns all entries in that dimension

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

> colnames(myMatrix) <- letters[1:3]

> myMatrix
     a b c
[1,] 1 2 3
[2,] 4 5 6
[3,] 7 8 9
```

Subsetting Matrices and Arrays [CONT'D]

```
> rownames(myMatrix) <- letters[4:6]

> myMatrix
  a b c
d 1 2 3
e 4 5 6
f 7 8 9

> myMatrix[1, 1]
[1] 1

> myMatrix[1:3, 1]
d e f
1 4 7

> myMatrix[1:2, ]
  a b c
d 1 2 3
e 4 5 6
```

```
> myMatrix[6]
[1] 8

# recycling logical row entries
> myMatrix[c(T, F), ]
  a b c
d 1 2 3
f 7 8 9

> myMatrix[1:2, "b"]
d e
2 5

# -------------------------------
# "[" will simplify results to the
# lowest possible dimensionality
# (more on this soon)
# -------------------------------
```

Selected Functional Generalizations

1D

1. `length()`
2. `names()`
3. `c()`

*n*D

1. `nrow()`, `ncol()`, `dim()`
2. `rownames()`, `colnames()`, `dimnames()`
3. `cbind()`, `rbind()`, `abind()`

n.b. a matrix or array can also be one-dimensional, e.g., an object that is defined as a matrix is permitted to only have one column or one row; although they may look and behave alike, a vector and a one-dimensional matrix behave differently and may generate strange output when using certain functions, e.g., `tapply()`

LAB

- Let's create some data for an experiment

1. First Name (`chr`)
2. Last Name (`chr`)
3. Total months of work experience (`num`)
4. Whether or not you are married (`T/F`)
5. Area code of your phone number (`num`)

LAB

- Let's create some data for an experiment

1. First Name (`chr`)
2. Last Name (`chr`)
3. Total months of work experience (`num`)
4. Whether or not you are married (`T/F`)
5. Area code of your phone number (`num`)

- **How do we input this into R?**

LAB

- Let's create some data for an experiment

1. First Name (`chr`)
2. Last Name (`chr`)
3. Total months of work experience (`num`)
4. Whether or not you are married (`T/F`)
5. Area code of your phone number (`num`)

- **How do we input this into R?**

```
# single quotes also work if you prefer
f_name <- c("paul", "john", "aasif", "saurin", "alex")
l_name <- c("intrevado", "smith", "ragna", "patel", "ozsen")

mosWorked <- c(22, 32, 7, 6, 87)
married <- c(TRUE, FALSE, FALSE, TRUE, FALSE)
areaCode <- c(415, 707, 415, 510, 510)
```

LAB [CONT'D]

- Is the data in the format we want?

LAB [CONT'D]

- Is the data in the format we want?
- You can examine the Environment pane in RStudio (easier)...

LAB [CONT'D]

- Is the data in the format we want?
- You can examine the Environment pane in RStudio (easier)...
- ... or use code (cumbersome)

CHECK DATA TYPES

```
> str(f_name)
 chr [1:5] "paul" "john" "aasif" "saurin" "alex"

> str(l_name)
 chr [1:5] "intrevado" "smith" "ragna" "patel" "ozsen"

> str(mosWorked)
 num [1:5] 22 32 7 6 87

> str(married)
 logi [1:5] TRUE FALSE FALSE TRUE FALSE

> str(areaCode)
 num [1:5] 415 707 415 510 510
```

LAB [CONT'D]

- `areaCode` should probably be changed to a nominal `factor` as it isn't truly meant to be a `numeric` value, and no summary statistics need be computed on `areaCode`

LAB [CONT'D]

- areaCode should probably be changed to a nominal factor as it isn't truly meant to be a numeric value, and no summary statistics need be computed on areaCode

RECODE areaCode AS FACTOR

```
> areaCode <- as.factor(areaCode)

> str(areaCode)
 Factor w/ 3 levels "415","510","707": 1 3 1 2 2

> areaCode
[1] 415 707 415 510 510
Levels: 415 510 707
```

LAB [CONT'D]

- What is the
  1. mean
  2. median
  3. standard deviation
  4. variance
  5. maximum
  6. minimum
  7. 75^{th} percentile

  of the number of `mosWorked`?

LAB [CONT'D]

COMPUTE DESCRIPTIVE STATISTICS

```
> mean(mosWorked)
[1] 30.8

> median(mosWorked)
[1] 22

> sd(mosWorked)
[1] 33.23703

> var(mosWorked)
[1] 1104.7

> max(mosWorked)
[1] 87

> min(mosWorked)
[1] 6

> quantile(mosWorked)
  0%  25%  50%  75% 100%
   6    7   22   32   87
```

LAB [CONT'D]

- Take a shortcut by using the `summary()` function

```
> summary(mosWorked)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
          6.0     7.0    22.0    30.8    32.0    87.0
```

which is a very similar output to the `quantile()` function except for the inclusion of the mean when calling `summary()`

LAB [CONT'D]

1. How many people are married?
2. What is the mean number of people who are married?

LAB [CONT'D]

1. How many people are married?
2. What is the mean number of people who are married?

SOLUTION

1.
```
> sum(married)
[1] 2
```

2.
```
> mean(married)
[1] 0.4
```

n.b. Numerical summary statistics can be computed on a non-numeric (logical) vector

LAB [CONT'D]

- What is the mean number of months worked by married persons participated in this survey?
- This is a more complex question than any of the questions previously asked and requires us to create a link between the previously independent vectors

Data Frames

This is why we use

Data Frames

- Most common way of storing data in R
- A data frame is a list with equal-length vectors
- Each vector must be of the same data type

```
> str(ToothGrowth)
'data.frame': 60 obs. of  3 variables:
 $ len : num  4.2 11.5 7.3 5.8 6.4 10 11.2 11.2 5.2 7 ...
 $ supp: Factor w/ 2 levels "OJ","VC": 2 2 2 2 2 2 2 2 2 2 ...
 $ dose: num  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 ...

 > ?ToothGrowth
```

Summary of Sample Data Frame

A data frame with 60 observations on 3 variables.

- [, 1] len numeric Tooth length
- [, 2] supp factor Supplement type (VC or OJ)
- [, 3] dose numeric Dose in milligrams/day

Creating and Manipulating Data Frames

Create a data frame using `data.frame()`

```
# this is sloppy coding etiquette and is only for exposition

> (xyz <- data.frame(1:3, c("a", "b", "c")))
  X1.3 c..a....b....c..
1    1                a
2    2                b
3    3                c

> str(xyz)
'data.frame': 3 obs. of  2 variables:
 $ X1.3           : int  1 2 3
 $ c..a....b....c..: Factor w/ 3 levels "a","b","c": 1 2 3
```

- Surround code with `()` to automatically print the result to the console

Creating and Manipulating Data Frames [CONT'D]

Create a data frame using `data.frame()`

```
> (xyz <- data.frame(numberColumn = 1:3, letterColumn = c("a", "b", "c")))
  numberColumn letterColumn
1            1            a
2            2            b
3            3            c

> str(xyz)
'data.frame': 3 obs. of  2 variables:
 $ numberColumn: int  1 2 3
 $ letterColumn: Factor w/ 3 levels "a","b","c": 1 2 3
```

- After creating the data frame, the first column of untitled numbers are row numbers
- Observe that even though the entries in `letterColumn` are characters that an `str(letterColumn)` shows the column to be a `Factor`

Creating and Manipulating Data Frames [CONT'D]

- If you want to suppress R's default behavior of turning strings into factors, use the options `stringsAsFactors = FALSE`

```
> (xyz <- data.frame(numberColumn = 1:3, letterColumn = c("a", "b", "c"),
    stringsAsFactors = F))
  numberColumn letterColumn
1            1            a
2            2            b
3            3            c

> str(xyz)
'data.frame': 3 obs. of  2 variables:
 $ numberColumn: int  1 2 3
 $ letterColumn: chr  "a" "b" "c"

 ...OR...

 (xyz <- tibble::data_frame(1:3, c("a", "b", "c")))
# A tibble: 3 x 2
  `1:3` `c("a", "b", "c")`
  <int>              <chr>
1     1                  a
2     2                  b
3     3                  c
```

Tibbles with `tibble`

- Data frames can be fickle, with their `stringsAsFactors = FALSE` requirements, etc.
- Tibbles—from the `tibble` package—**are data frames**, with some minor modifications which make operating with tibbles a more pleasant and worry-free experience than dealing with raw data frames
- Creating a tibble is the same as creating a data frame, save the suffix now changes to `tibble()` instead of `data.frame()`
- Coercing a data frame (or other data structure) is done with the `as_tibble()` function

Tibbles with `tibble` [CONT'D]

- Tibbles have some very convenient advantages
- When importing (e.g., `csv`'s) or coercing data, `tibble()` function:
  1. **will not** automatically convert strings to factors
  2. **will not** change the names of variables
  3. **will not** create row names
  4. **will not** do any partial matching when subsetting
- Subsetting tibbles can be done with either the `$` or `[[` operators

Creating and Manipulating Data Frames [CONT'D]

- Recall that a data frame is a list, which means that `typeof(myDataFrame)` will output a list
- Instead use `is.data.frame()`
- An object can be coerced to a data frame using `as.data.frame()` or `tibble::as_data_frame()`

Subsetting Data Frames

As data frames possess the characteristics of both lists and matrices, you can subset using two methods

```
> (myDataFrame <- data.frame(x = 1:3, y = -4:-6, z = LETTERS[1:3]))
  x  y z
1 1 -4 A
2 2 -5 B
3 3 -6 C

> myDataFrame[1:2]
  x  y
1 1 -4
2 2 -5
3 3 -6

> myDataFrame[1, 3]
[1] A
Levels: A B C
```

Subsetting Data Frames [CONT'D]

- If you subset using a single vector, the result behaves as a list
- If you subset using two vectors, the result behaves as a matrix

```
> myDataFrame[1]
  x
1 1
2 2
3 3

> myDataFrame[1, ]
  x  y z
1 1 -4 A

> myDataFrame[c(1, 3)]
  x z
1 1 A
2 2 B
3 3 C

> myDataFrame[2, 2]
[1] -5
```

```
> myDataFrame[, c(1, 3)]
  x z
1 1 A
2 2 B
3 3 C

> myDataFrame["x"]
  x
1 1
2 2
3 3

> str(myDataFrame["x"]) # index as a list
'data.frame': 3 obs. of  1 variable:
 $ x: int  1 2 3

> str(myDataFrame[, "x"]) # index as a matrix
 int [1:3] 1 2 3
```

$ for Data Frames

- $ is a shorthand operator often used to access variables within a data frame

E.g. For the dataset `iris`, the code that returns the first three `Sepal.Length` values is `iris$Sepal.Length[1:3]`

LAB [REVISITED]

- What is the mean age of juniors and seniors who participated in this survey?
- This is a more complex question than any of the questions previously asked and requires us to create a link between the previously independent vectors

LAB [REVISITED]

- What is the mean age of juniors and seniors who participated in this survey?
- This is a more complex question than any of the questions previously asked and requires us to create a link between the previously independent vectors
- We can collect and link all these vectors into a single data frame

```
> myExperimentalData <- data.frame(l_name, f_name, juniorSenior,
                                   areaCode, mosWorked)

> myExperimentalData
     l_name f_name juniorSenior areaCode  mosWorked
1 intrevado   paul         TRUE      415         22
2     smith   john        FALSE      707         32
3     ragna  aasif        FALSE      415          7
4     patel saurin         TRUE      510          6
5     ozsen   alex        FALSE      510         87
```

LAB [CONT'D]

- A more convenient, accessible, and aesthetically pleasing way is to observe the data frame in the Environment pane
- By clicking on the name of the `data frame`, `myExperimentalData`, a spreadsheet-like display will pop up in the Console pane

LAB [CONT'D]

- Let's explore this new data frame

```
> str(myExperimentalData)
  'data.frame'       : 5 obs. of  5 variables:
   $ l_name          : Factor w/ 5 levels "intrevado","ozsen",..: 1 5 4 3 2
   $ f_name          : Factor w/ 5 levels "aasif","alex",..: 4 3 1 5 2
   $ juniorSenior    : logi  TRUE FALSE FALSE TRUE FALSE
   $ areaCode        : Factor w/ 3 levels "415","510","707": 1 3 1 2 2
   $ mosWorked       : num  22 32 7 6 87
```

- What has changed in the process of joining the individual vectors into a single data frame?

LAB [CONT'D]

- Let's explore this new data frame

```
> str(myExperimentalData)
  'data.frame'       : 5 obs. of  5 variables:
   $ l_name          : Factor w/ 5 levels "intrevado","ozsen",..: 1 5 4 3 2
   $ f_name          : Factor w/ 5 levels "aasif","alex",..: 4 3 1 5 2
   $ juniorSenior    : logi  TRUE FALSE FALSE TRUE FALSE
   $ areaCode        : Factor w/ 3 levels "415","510","707": 1 3 1 2 2
   $ mosWorked       : num  22 32 7 6 87
```

- What has changed in the process of joining the individual vectors into a single data frame?
- Both `l_name` and `f_name` have been converted from `character` vectors into `factors` while being merged into the data frame (and we weren't even asked)

LAB [CONT'D]

- To avoid this automatic `character` to `factor` coercion, include the additional argument `stringsAsFactors = FALSE` in the `data.frame` function

```
> myExperimentalData <- data.frame(l_name, f_name, juniorSenior,
                                   areaCode, mosWorked,
                                   stringsAsFactors = FALSE)

> str(myExperimentalData)
  'data.frame': 5 obs. of  5 variables:
   $ l_name         : chr  "intrevado" "smith" "ragna" "patel" ...
   $ f_name         : chr  "paul" "john" "aasif" "saurin" ...
   $ juniorSenior   : logi  TRUE FALSE FALSE TRUE FALSE
   $ areaCode       : Factor w/ 3 levels "415","510","707": 1 3 1 2 2
   $ mosWorked      : num  22 32 7 6 87
```

- With the experimental data combined into a `data frame`, subsetting data is now far easier

TRY THIS

What is the mean number of months worked by juniors or seniors who participated in this survey?

TRY THIS

What is the mean number of months worked by juniors or seniors who participated in this survey?

SOLUTION

```
> mean(myExperimentalData$mosWorked[myExperimentalData$juniorSenior])
[1] 14
```

PREAMBLE

```
> install.packages("hflights") # if not already installed
> library(hflights)
```

YOU TRY IT

1. Return all columns of the data frame where the flight departed from IAH.
2. How many flights departed from IAH? How many departed from HOU?
3. What percentage of flights departed from IAH?
4. What is the mean flight time (`AirTime`) for all flights leaving IAH, and arriving in DFW (Detroit) in January 2011?

PREAMBLE

```
> install.packages("hflights") # if not already installed
> library(hflights)
```

SOLUTION

1.
```
> hflights[hflights$Origin == "IAH", ]
```

2.
```
> sum(hflights$Origin == "IAH")
> sum(hflights$Origin == "HOU")
```

3.
```
> sum(hflights$Origin == "IAH") / nrow(hflights) * 100
```

4.
```
> mean(hflights$AirTime[hflights$Origin == "IAH" & hflights$Dest == "DFW" &
    hflights$Month == 1 & hflights$Year == 2011], na.rm = T)
```

Partial Matching with $

```
> (myDataFrame <- data.frame(abc = 1:3,
    abd = -4:-6, xyz = LETTERS[1:3],
    stringsAsFactors = F))
  abc abd xyz
1   1  -4   A
2   2  -5   B
3   3  -6   C

# when the call returns more than one
#   column of a data frame, a value
#   of NULL is returned
> myDataFrame[["a", exact = F]]
NULL

# ibid
> myDataFrame[["ab", exact = F]]
NULL
```

```
# exact match to column name
> myDataFrame[["abd", exact = F]]
[1] -4 -5 -6

# partial matching b/c exact = F
#   call returns single column
#   therefore non-NULL return
> myDataFrame[["x", exact = F]]
[1] "A" "B" "C"

> myDataFrame$a
NULL

> myDataFrame$x
[1] "A" "B" "C"
```

Dynamic Column Referencing with $

If you want to dynamically access a column of a data frame, you might choose to store the name of of that column in variable. When doing so, be sure not access the column incorrectly.

```
> (myDataFrame <- data.frame(abc = 1:3, xyz = -4:-6))
  abc xyz
1   1  -4
2   2  -5
3   3  -6

> (dynColName <- "xyz")
[1] "xyz"

> myDataFrame$dynColName
NULL

> myDataFrame[dynColName]
  xyz
1  -4
2  -5
3  -6

> myDataFrame[[dynColName]]
[1] -4 -5 -6
```

Subsetting Operators

- We have seen the use of `[` to subset, but we can also use `[[` to subset
- Recall, when `[` is used to subset a list, it returns a list, whereas using `[[` to subset a list returns the **contents** of said list
- As data frames are lists of columns, you can use `[[` to extract a column from a data frame

```
> myDataFrame <- data.frame(
    x = 1:3, y = -4:-6, z = LETTERS[1:3])

> class(myDataFrame["x"])
[1] "data.frame"

> class(myDataFrame[1])
[1] "data.frame"

> class(myDataFrame[["x"]])
[1] "integer"
```

```
> class(myDataFrame[[1]])
[1] "integer"

> typeof(myDataFrame["x"])
[1] "list"

> typeof(myDataFrame[1])
[1] "list"

> typeof(myDataFrame[["x"]])
[1] "integer"

> typeof(myDataFrame[[1]])
[1] "integer"
```

$ versus [[

- $ does partial matching
- [[does **not** do partial matching

```
> (myDataFrame <- data.frame(abc = 1:3, xyz = -4:-6))
  abc xyz
1   1  -4
2   2  -5
3   3  -6

> myDataFrame$a
[1] 1 2 3

> myDataFrame["a"]
Error in `[.data.frame`(myDataFrame, "a") : undefined columns selected

> myDataFrame[["a"]]
NULL
```

Simplifying vs. Preserving Subsetting

- **Simplifying** subsets returns the simplest possible data structure that can represent the output
- **Preserving** subsetting keep the structure of the output the same as the input and is generally better programming etiquette because the result will always be the same
- The `drop` option when subsetting is one of the most common sources of programming error

`drop` `logical. If TRUE the result is coerced to the lowest possible dimension. The default is to drop if only one column is left, but not to drop if only one row is left.`

- Preserving is the same for all data types, but simplifying behavior varies slightly across data types

Simplifying vs. Preserving: ATOMIC VECTORS [EXAMPLE]

```
> myAtomicVector_01 <- c(99.1, 98.2, 97.3, 96.4)
> names(myAtomicVector_01) <- LETTERS[1:4]

> myAtomicVector_01
   A    B    C    D
99.1 98.2 97.3 96.4

> (x_01 <- myAtomicVector_01[1])
   A
99.1

> x_01["A"]
   A
99.1

> (x_02 <- myAtomicVector_01[[1]]) # strips away the names
[1] 99.1

> x_02["A"]
[1] NA
```

Simplifying vs. Preserving: LISTS [EXAMPLE]

```
> myList <- list(A = 1, B = 2)

> str(myList[1])
List of 1
 $ A: num 1

> typeof(myList[1])
[1] "list"

> class(myList[1])
[1] "list"

> str(myList[[1]])
 num 1

> typeof(myList[[1]])
[1] "double"

> class(myList[[1]])
[1] "numeric"
```

Simplifying vs. Preserving: FACTORS [EXAMPLE]

```
> (myFactor <- factor(LETTERS[1:3]))
[1] A B C
Levels: A B C

> myFactor[1]
[1] A
Levels: A B C

> myFactor[[1]]
[1] A
Levels: A B C

> typeof(myFactor[1])
[1] "integer"

> class(myFactor[1])
[1] "factor"

> myFactor[1, drop = TRUE]  # drops unused levels
[1] A
Levels: A

> typeof(myFactor[1, drop = TRUE])
[1] "integer"

> class(myFactor[1, drop = TRUE])
[1] "factor"
```

Simplifying vs. Preserving: MATRICES/ARRAYS [EXAMPLE]

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

> myMatrix[1, , drop = F]
     [,1] [,2] [,3]
[1,]    1    2    3

> str(myMatrix[1, , drop = F])
 int [1, 1:3] 1 2 3

> class(myMatrix[1, , drop = F])
[1] "matrix"

> typeof(myMatrix[1, , drop = F])
[1] "integer"
```

```
# if any dimension has length 1,
#   dimension is dropped

> myMatrix[1, ]
[1] 1 2 3

> str(myMatrix[1, ])
 int [1:3] 1 2 3

> class(myMatrix[1, ])
[1] "integer"

> typeof(myMatrix[1, ])
[1] "integer"

# why are
#   typeof(myMatrix[1, , drop = F])
#   and typeof(myMatrix[1, ])
#   both "integer" ?
```

Simplifying vs. Preserving: DATA FRAMES [EXAMPLE] [1/2]

```
> (myDataFrame <- data.frame(x = 1:3, y = -4:-6))
  x  y
1 1 -4
2 2 -5
3 3 -6

> str(myDataFrame[1])
'data.frame': 3 obs. of  1 variable:
 $ x: int  1 2 3

> typeof(myDataFrame[1])
[1] "list"

> class(myDataFrame[1])
[1] "data.frame"

> str(myDataFrame[[1]])
 int [1:3] 1 2 3

> typeof(myDataFrame[[1]])
[1] "integer"

> class(myDataFrame[[1]])
[1] "integer"

# if the output is a single column, returns a vector instead of a data frame
```

Simplifying vs. Preserving: DATA FRAMES [EXAMPLE] [2/2]

```
> (myDataFrame <- data.frame(x = 1:3, y = -4:-6))
  x  y
1 1 -4
2 2 -5
3 3 -6

> str(myDataFrame[, "x", drop = F])
'data.frame': 3 obs. of  1 variable:
 $ x: int  1 2 3

> typeof(myDataFrame[, "x", drop = F])
[1] "list"

> class(myDataFrame[, "x", drop = F])
[1] "data.frame"

> str(myDataFrame[, "x"])
 int [1:3] 1 2 3

> typeof(myDataFrame[, "x"])
[1] "integer"

> class(myDataFrame[, "x"])
[1] "integer"

# if the output is a single column, returns a vector instead of a data frame
```

TRY THIS

1. Using the simple linear regression model
   `myMod <- lm(mpg ~ wt, data = mtcars)`
   1. Extract the residual degrees of freedom
   2. Extract R^2 and R^2_A
2. Write code that would randomly permute the columns of a data frame.
3. Write code that would randomly permute the rows of a data frame.

SOLUTION

1.
```
> myMod$df.residual    # hint: str(myMod)
[1] 30

> summary(myMod)$r.squared  # hint: str(summary(myMod))
[1] 0.7528328

> summary(myMod)$adj.r.squared  # hint: str(summary(myMod))
[1] 0.7445939
```

2.
```
> myDF[, sample(ncol(myDF), replace = F)]  # shuffle cols
```

3.
```
> myDF[sample(nrow(myDF), replace = F), ] # shuffle rows
```