

# Subsetting & data structures

**Hadley Wickham**

**October 2009**

1. Subsetting

2. Data structures

1. Basic data types

2. Vectors, matrices & arrays

3. Lists & data.frames

# Subsetting

Key to efficient use of R is  
mastering subsetting.

Take one minute to recall the 5  
basic types of subsetting

**blank** include all

**integer** **+ve:** include  
**-ve:** exclude

**logical** include TRUEs

**character** lookup by name

# Integer subsetting

```
# Nothing
str(diamonds[, ])

# Positive integers & nothing
diamonds[1:6, ] # same as head(diamonds)
diamonds[, 1:4] # watch out!

# Two positive integers in rows & columns
diamonds[1:10, 1:4]

# Repeating input repeats output
diamonds[c(1,1,1,2,2), 1:4]

# Negative integers drop values
diamonds[-(1:53900), -1]
```

```
# Useful technique: Order by one or more columns  
diamonds[order(diamonds$x), ]
```

```
# Useful technique: Combine two tables  
carats <- data.frame(table(carat = diamonds$carat))  
mtch <- match(diamonds$carat, carats$carat)  
diamonds$carat_count <- carats$Freq[mtch]
```

# Logical subsetting



```
# The most complicated to understand, but  
# the most powerful. Lets you extract a  
# subset defined by some characteristic of  
# the data
```

```
x_big <- diamonds$x > 10
```

```
head(x_big)
```

```
sum(x_big)
```

```
mean(x_big)
```

```
table(x_big)
```

```
diamonds$x[x_big]
```

```
diamonds[x_big, ]
```

```
small <- diamonds[diamonds$carat < 1, ]
lowqual <- diamonds[diamonds$clarity
  %in% c("I1", "SI2", "SI1"), ]

# Comparison functions:
# < > <= >= != == %in%

# Boolean operators: & | !
small <- diamonds$carat < 1 &
  diamonds$price > 500
lowqual <- diamonds$colour == "D" |
  diamonds$cut == "Fair"
```

# Your turn

Select the diamonds that have:

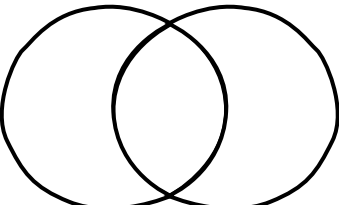
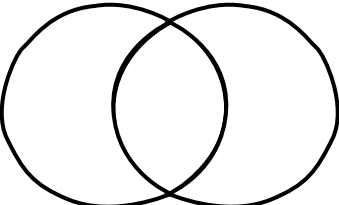
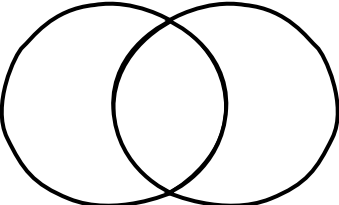
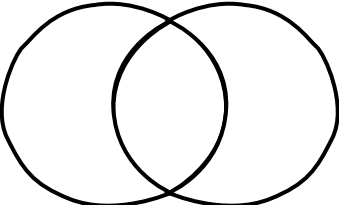
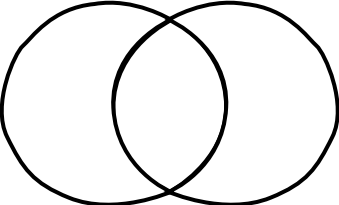
Equal x and y dimensions.

Depth between 55 and 70.

Carat smaller than the mean.

Cost more than \$10,000 per carat.

Are of good quality or better.

	A
	B
	$A \cup B$
	$A \cap B$
	$A \cap \neg B$

Easy to  
select first n

```
a <- seq(0, 100, by = 2)
```

```
b <- seq(0, 100, by = 3)
```

```
intersect(a, b) # divisible by 2 and 3
```

```
union(a, b) # divisible by 2 or 3
```

```
setdiff(a, b) # divisible by 2, but not 3
```

```
setdiff(b, a) # divisible by 3, but not 2
```

```
setdiff(union(a, b), intersect(a, b))
```

```
# divisible by either, but not both
```

```
A <- rep(c(F, T), length = 100)
B <- rep(c(F, F, T), length = 100)
```

```
A & B      # divisible by 2 and 3
A | B      # divisible by 2 or 3
A & !B     # divisible by 2, but not 3
B & !A     # divisible by 3, but not 2
xor(A, B)  # divisible by either, but not both
(A | B) & !(A & B) # same thing
```

*which() converts  
from logical to numeric*

# Character subsetting

```
# Matches by names
diamonds[1:5, c("carat", "cut", "color")]

# Useful technique: change labelling
c("Fair" = "C", "Good" = "B", "Very Good" = "B+",
  "Premium" = "A", "Ideal" = "A+")[diamonds$cut]

# Can also be used to collapse levels
table(c("Fair" = "C", "Good" = "B", "Very Good" =
  "B", "Premium" = "A", "Ideal" = "A")[diamonds$cut])

# (see ?cut for continuous to discrete equivalent)
```

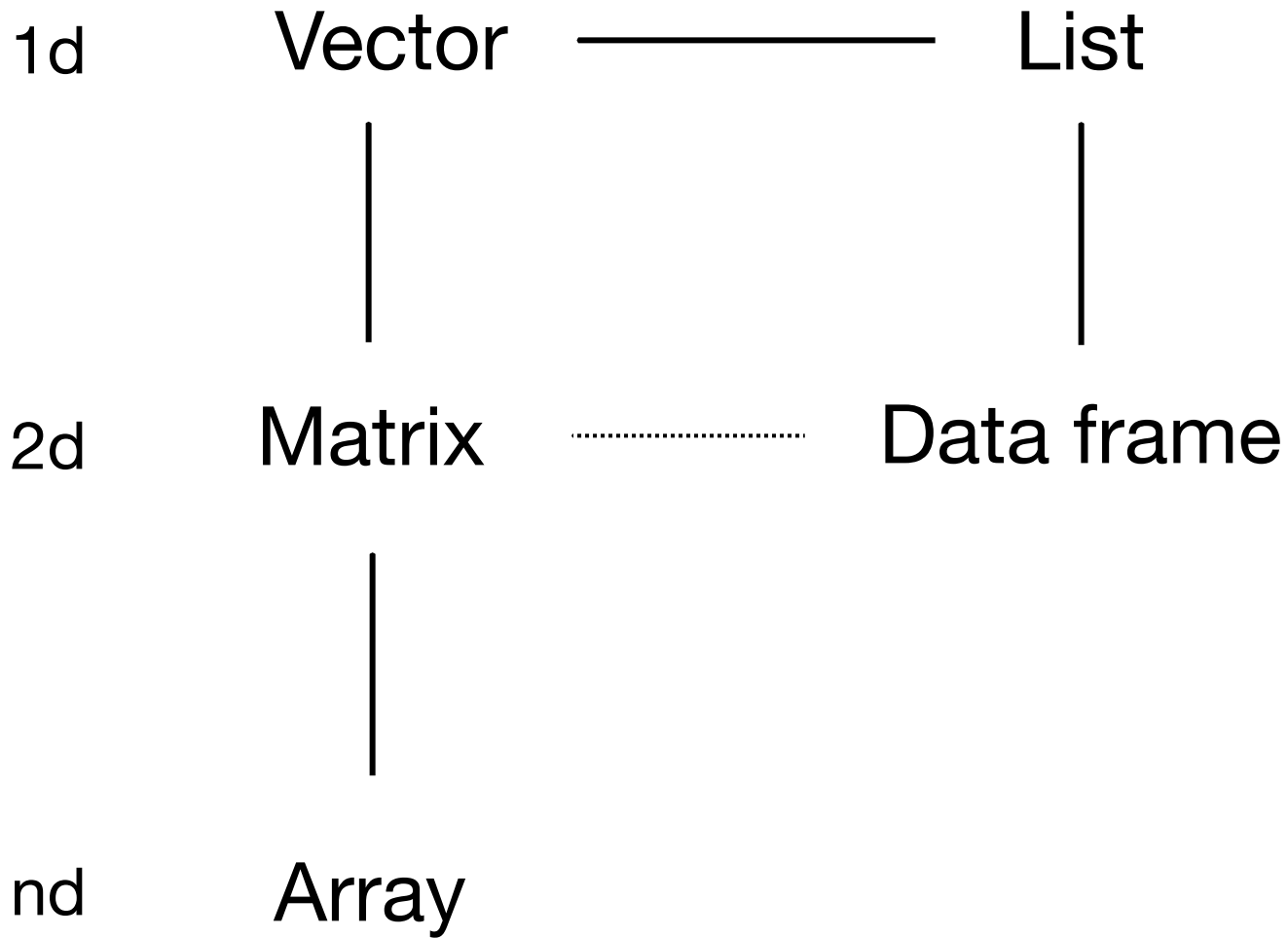


# Your turn

In the mpg dataset, create a new variable giving the origin of the manufacturer: Europe, America or Asia.

# Data structures

Take two minutes to come up with  
the 5 basic data structures



Same types

Different types

**stro()**

**character**

```
as.character(c(T, F))
```

```
as.character(seq_len(5))
```

**numeric**

```
as.logical(c(0, 1, 100))
```

```
as.logical(c("T", "F", "a"))
```

**logical**

```
as.numeric(c("A", "100"))
```

```
as.numeric(c(T, F))
```

When vectors of different types occur in an expression, they will be automatically coerced to the same type: character > numeric > logical

mode()

names()

Optional, but useful

length()

A scalar is a vector of length 1

Technically, these are all **atomic** vectors

# Your turn

Experiment with automatic coercion.  
What is happening in the following cases?

```
104 & 2 < 4
```

```
mean(diamonds$cut == "Good")
```

```
diamonds$color == "D" | "E" | "F"
```

# Matrix (2d)

# Array (>2d)

Just like a vector. Has  
`mode()` and `length()`.

Create with `matrix()` or  
`array()`, or from a vector  
by setting `dim()`

`as.vector()` converts  
back to a vector

```
a <- seq_len(12)
dim(a) <- c(1, 12)
dim(a) <- c(4, 3)
dim(a) <- c(2, 6)
dim(a) <- c(3, 2, 2)
```

```
a <- 1:10
b <- 11:20
```

```
cbind(a, b)
rbind(a, b)
```

```
# What's the difference between a & b?
```

```
a <- matrix(x, 4, 3)
```

```
b <- array(x, c(4, 3))
```

```
# What's the difference between x & y
```

```
y <- matrix(x, 12)
```



# List

Is also a vector (so has mode, length and names), but is different in that it can store any other vector inside it (including lists).

Use `unlist()` to convert to a vector. Use `as.list()` to convert a vector to a list.

```
c(1, 2, c(3, 4))  
list(1, 2, list(3, 4))
```

```
c("a", T, 1:3)  
list("a", T, 1:3)
```

```
a <- list(1:3, 1:5)  
unlist(a)  
as.list(a)
```

```
b <- list(1:3, "a", "b")  
unlist(b)
```

Technically a **recursive** vector

# Data frame

List of vectors, each of the same length. (Cross between list and matrix)

Different to matrix in that each column can have a different type

```
data.frame(  
  a = 1:10,  
  b = letters[1:10]  
)
```

```
load(url("http://had.co.nz/stat405/data/quiz.rdata"))
```

```
# What is a? What is b?
```

```
# How are they different? How are they similar?
```

```
# How can you turn a in to b?
```

```
# How can you turn b in to a?
```

```
# What are c, d, and e?
```

```
# How are they different? How are they similar?
```

```
# How can you turn one into another?
```

```
# What is f?
```

```
# How can you extract the first element?
```

```
# How can you extract the first value in the first
```

```
# element?
```

```
# a is numeric vector, containing the numbers 1 to 10
# b is a list of numeric scalars
# they contain the same values, but in a different format
identical(a[1], b[[1]])
identical(a, unlist(b))
identical(b, as.list(a))

# c is a named list
# d is a data.frame
# e is a numeric matrix
# From most to least general: c, d, e
identical(c, as.list(d))
identical(d, as.data.frame(c))
identical(e, data.matrix(d))
```

```
# f is a list of matrices of different dimensions
```

```
f[[1]]
```

```
f[[1]][1, 2]
```

1d	names()	length()	c()
2d	colnames() rownames()	ncol() nrow()	cbind() rbind()
nd	dimnames()	dim()	abind() (special package)

```
# What does these subsetting operations do?
```

```
# Why do they work? (Remember to use str)
```

```
diamonds[1]
```

```
diamonds[[1]]
```

```
diamonds["cut"]
```

```
diamonds[["cut"]]
```

```
diamonds$cut
```

```
# How are these subsetting operations different?
```

```
a <- matrix(1:12, 4, 3)
```

```
a[, 1]
```

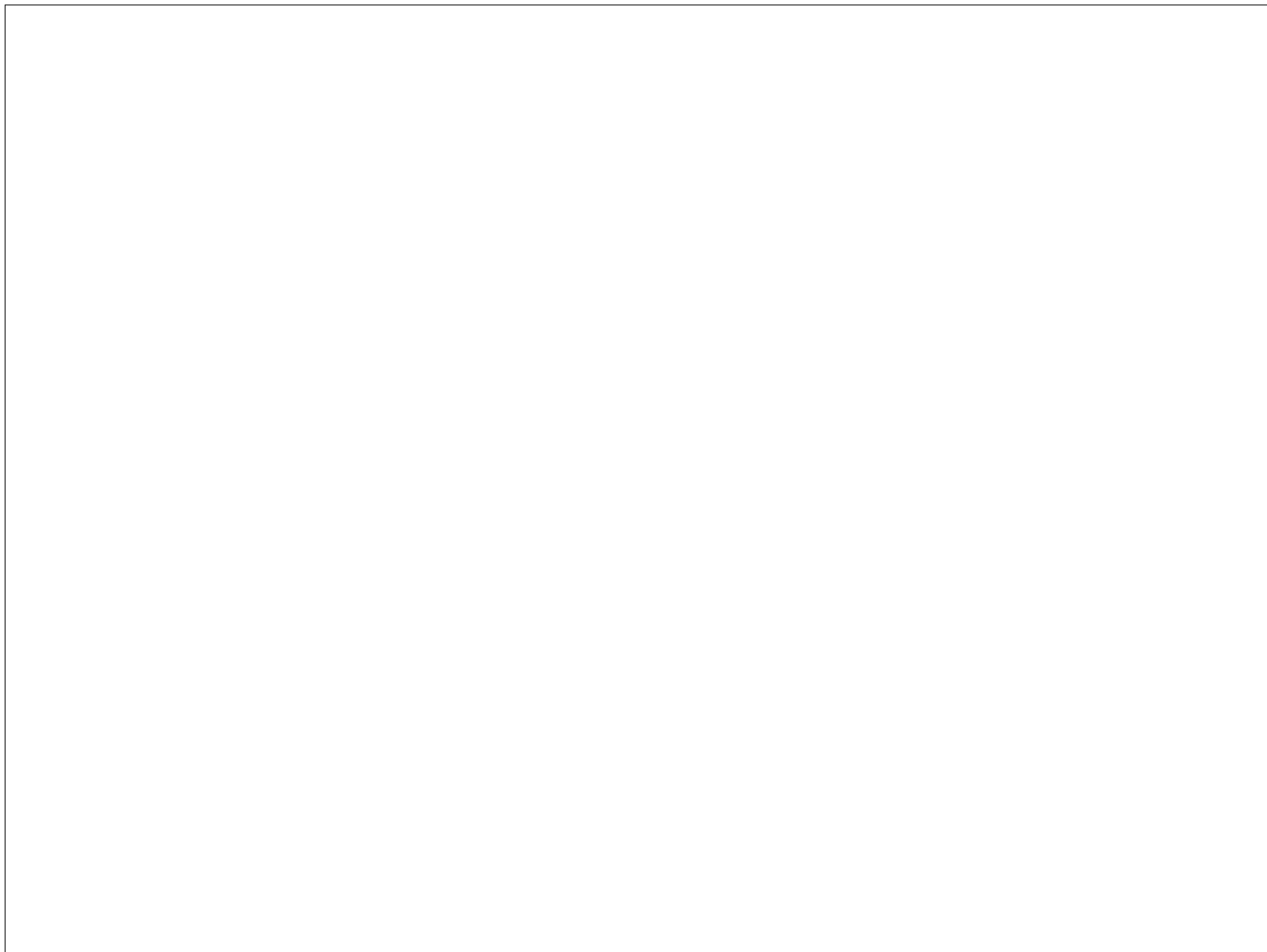
```
a[, 1, drop = FALSE]
```

```
a[1, ]
```

```
a[1, , drop = FALSE]
```

<b>Vectors</b>	<code>x[1:4]</code>	<code>-</code>
<b>Matrices Arrays</b>	<code>x[1:4, ]</code> <code>x[, 2:3, ]</code>	<code>x[1:4, ,</code> <code>drop = F]</code>
<b>Lists</b>	<code>x[[1]]</code> <code>x\$name</code>	<code>x[1]</code>





This work is licensed under the Creative Commons Attribution-Noncommercial 3.0 United States License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/3.0/us/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.