Basics of R (cont.)

Dennis E. Slice
Gordon Erlebacher

What we will cover

- Containers
- Vectors, lists
- Adding, multiplying, subtracting vectors
- Benefits of vectors
- Functions
- How to get help from the system

Briefly

- Basic type: integers, reals, strings, logicals
- Each basic type is a vector of a single element

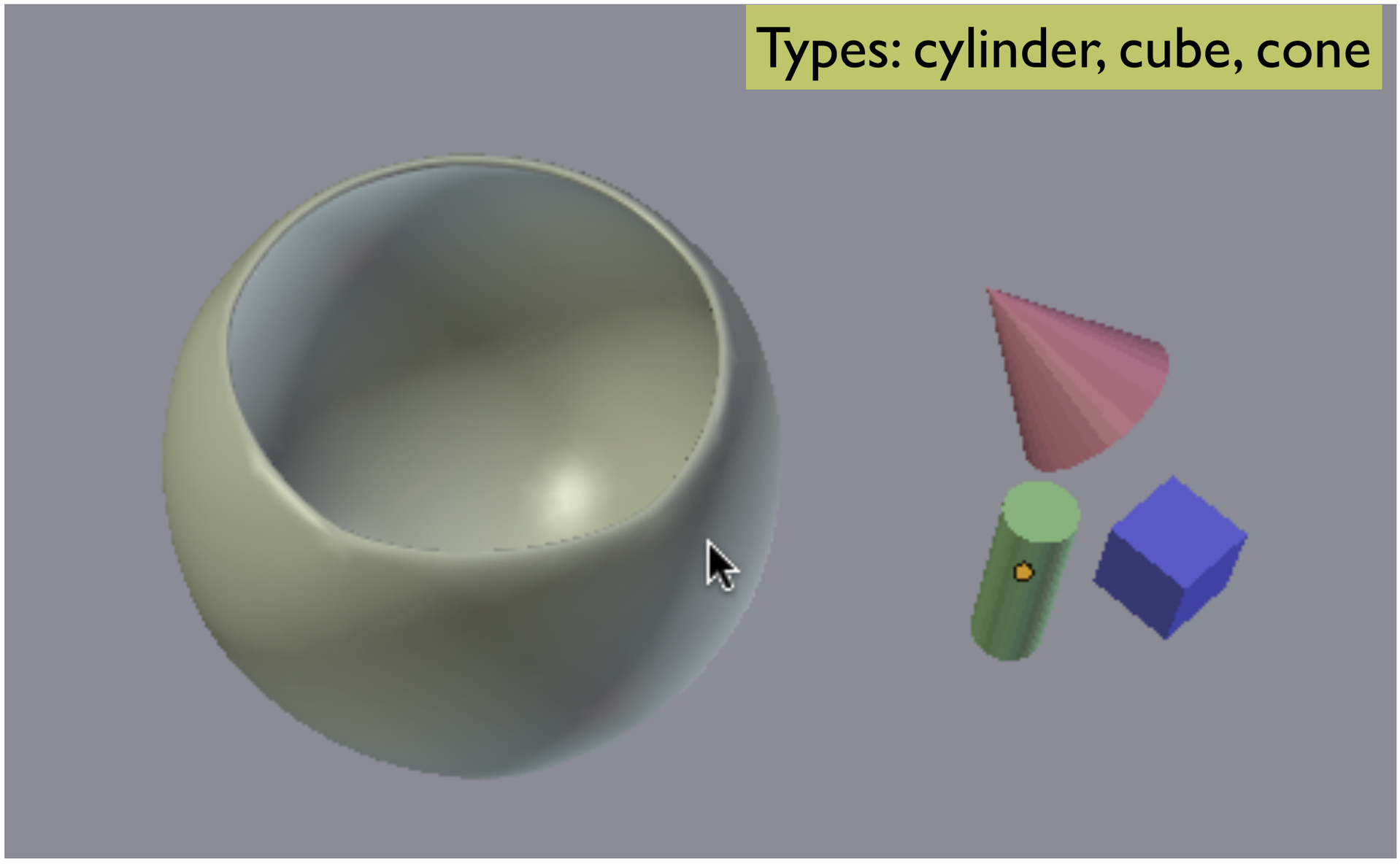
- Vectors: collections of objects of the same basic type (integers, reals, logicals, strings)
- Lists: collections of objects of different type

NOW, we explain in MORE DETAIL

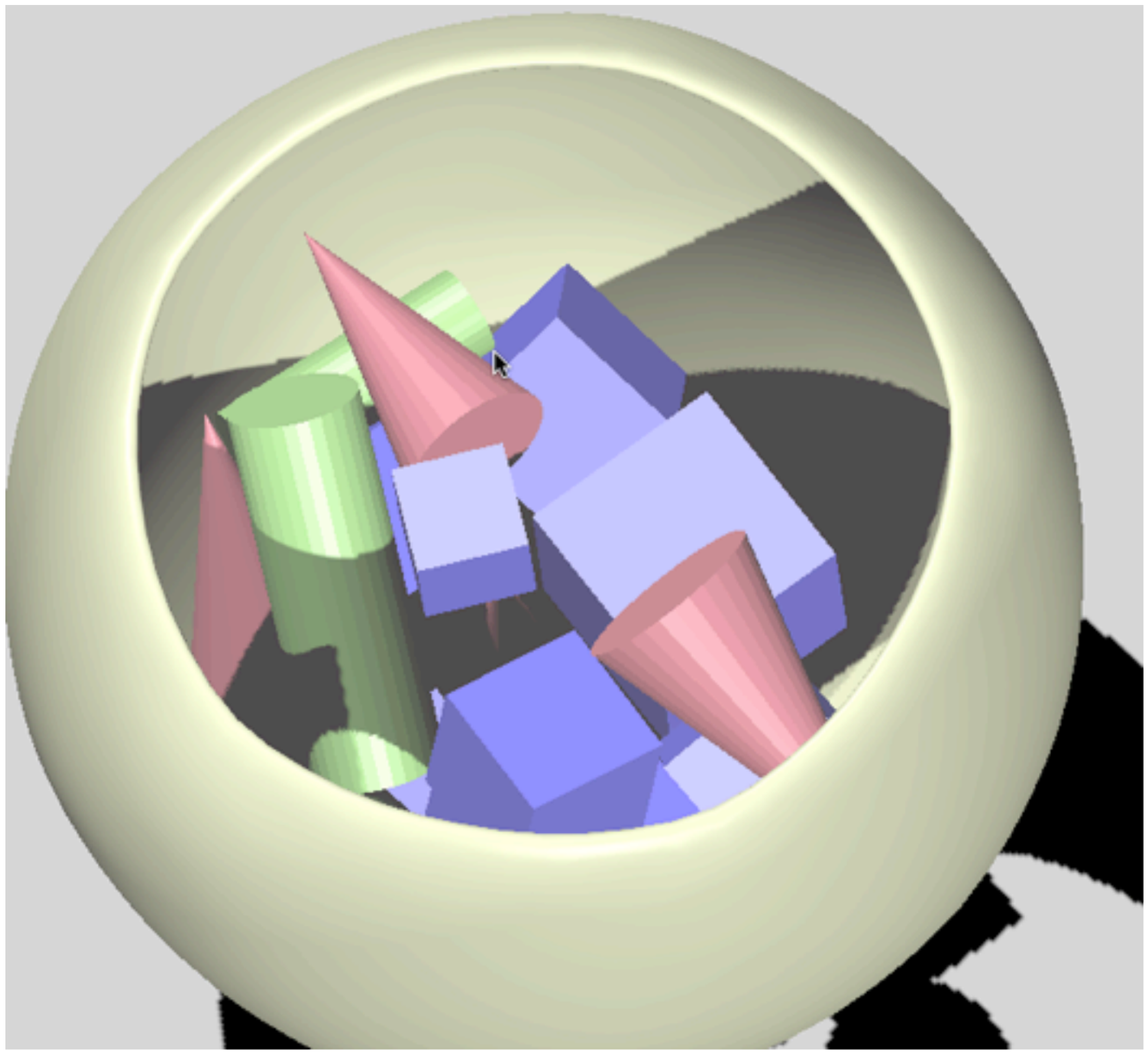
Container

- A container *contains* things

Container + 3 types of objects



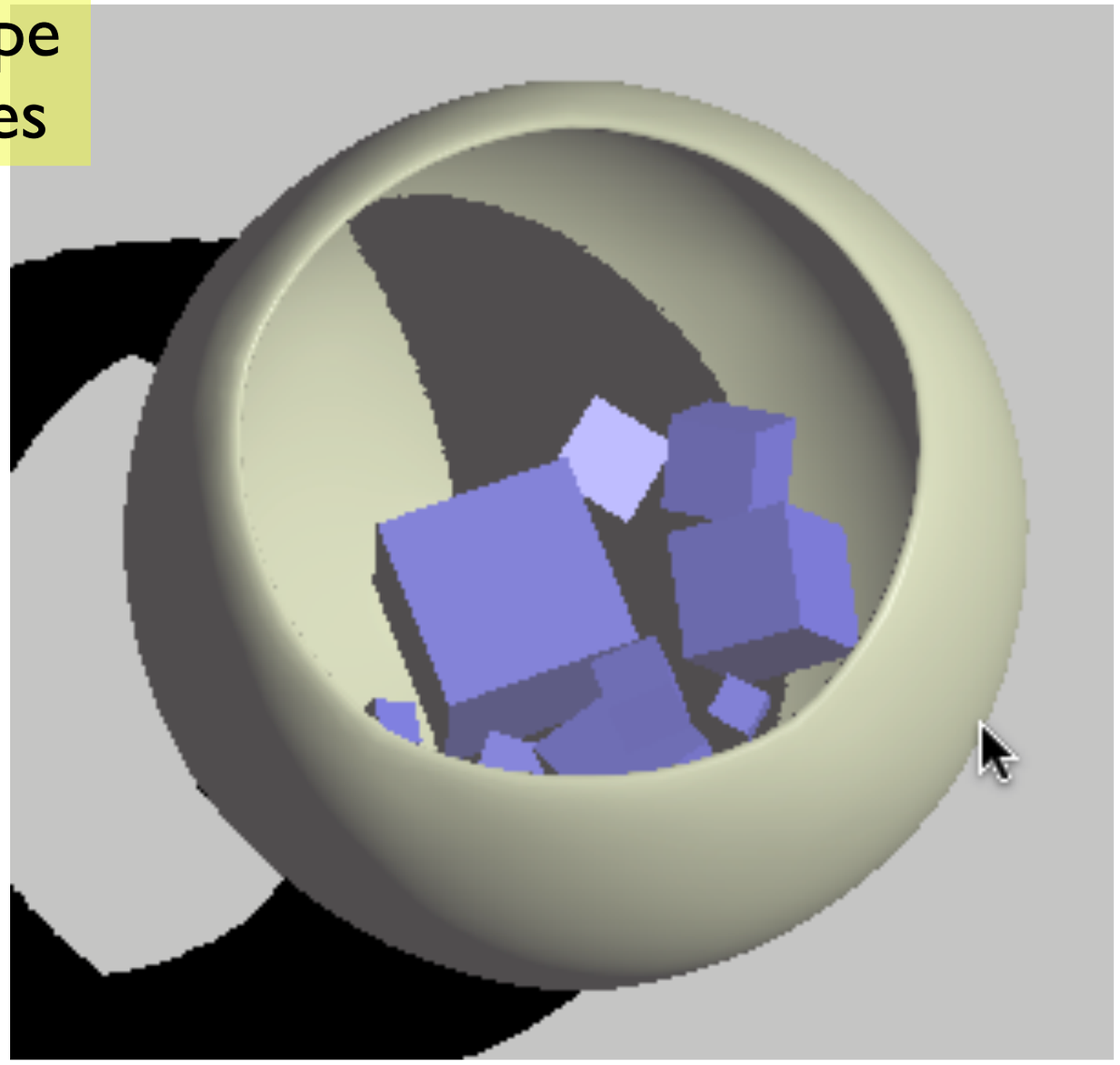
Container with objects of different types



Container with objects of same type: *cubes*

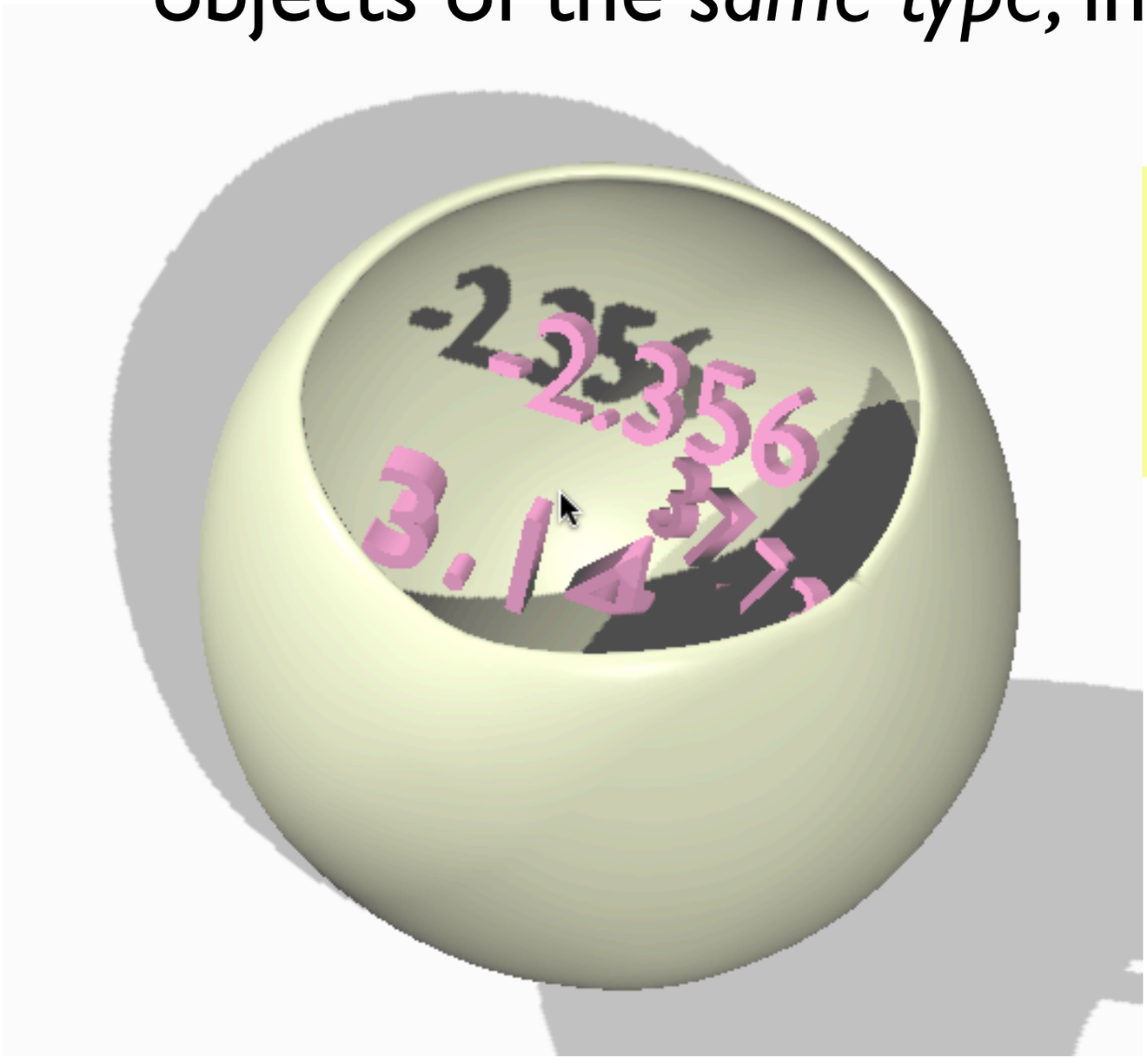
All cubes are of the same type
Cubes can have different sizes

The container cannot contain a *mixture* of cubes and cones



Vector

A vector is a *container* that can only include objects of the *same type*, in a particular order

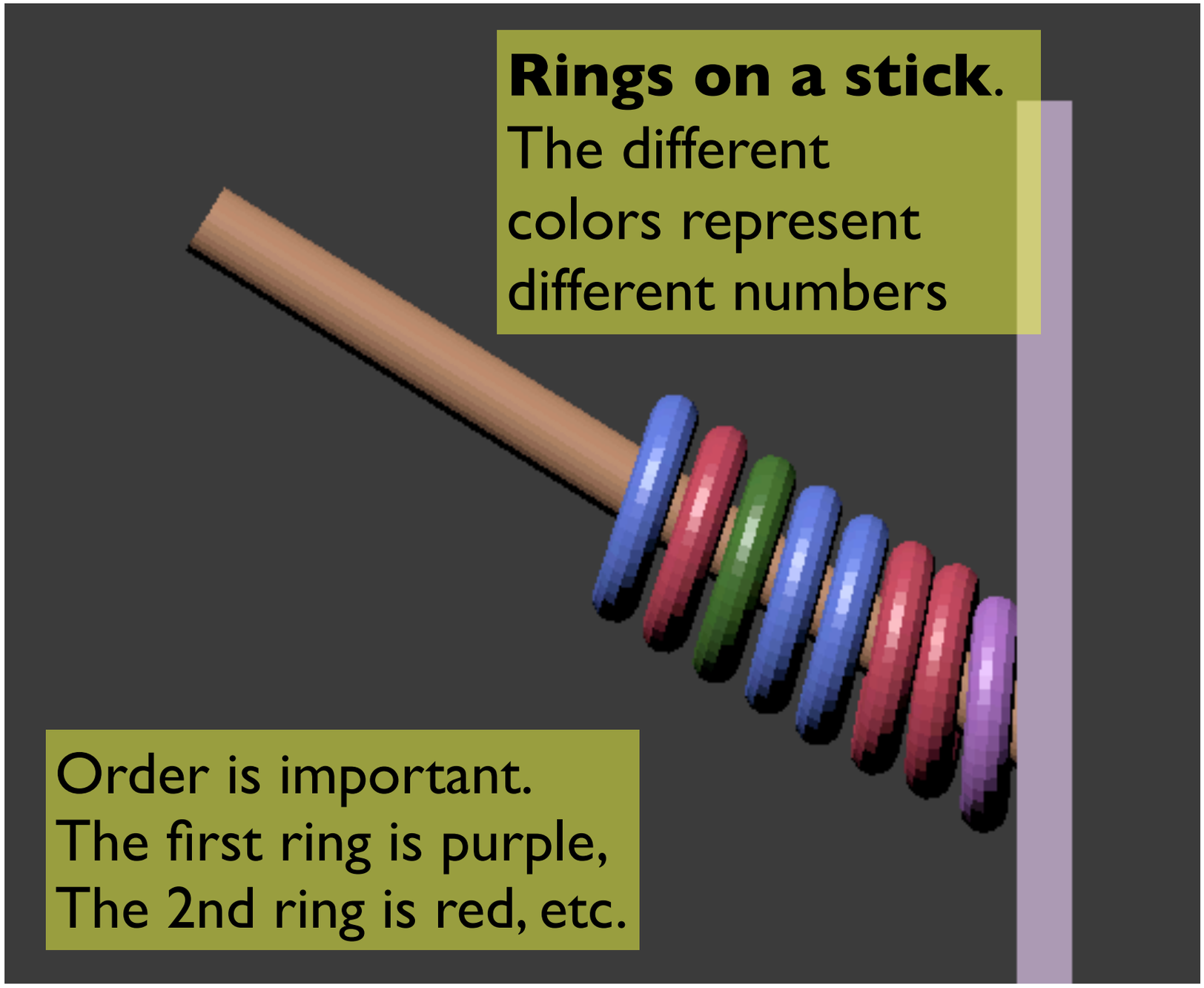


Representation in R of the bowl with numbers:

c(2.356, 3.14, 37.73)

Not exactly a vector since in a bowl, the elements are not in any particular *order*

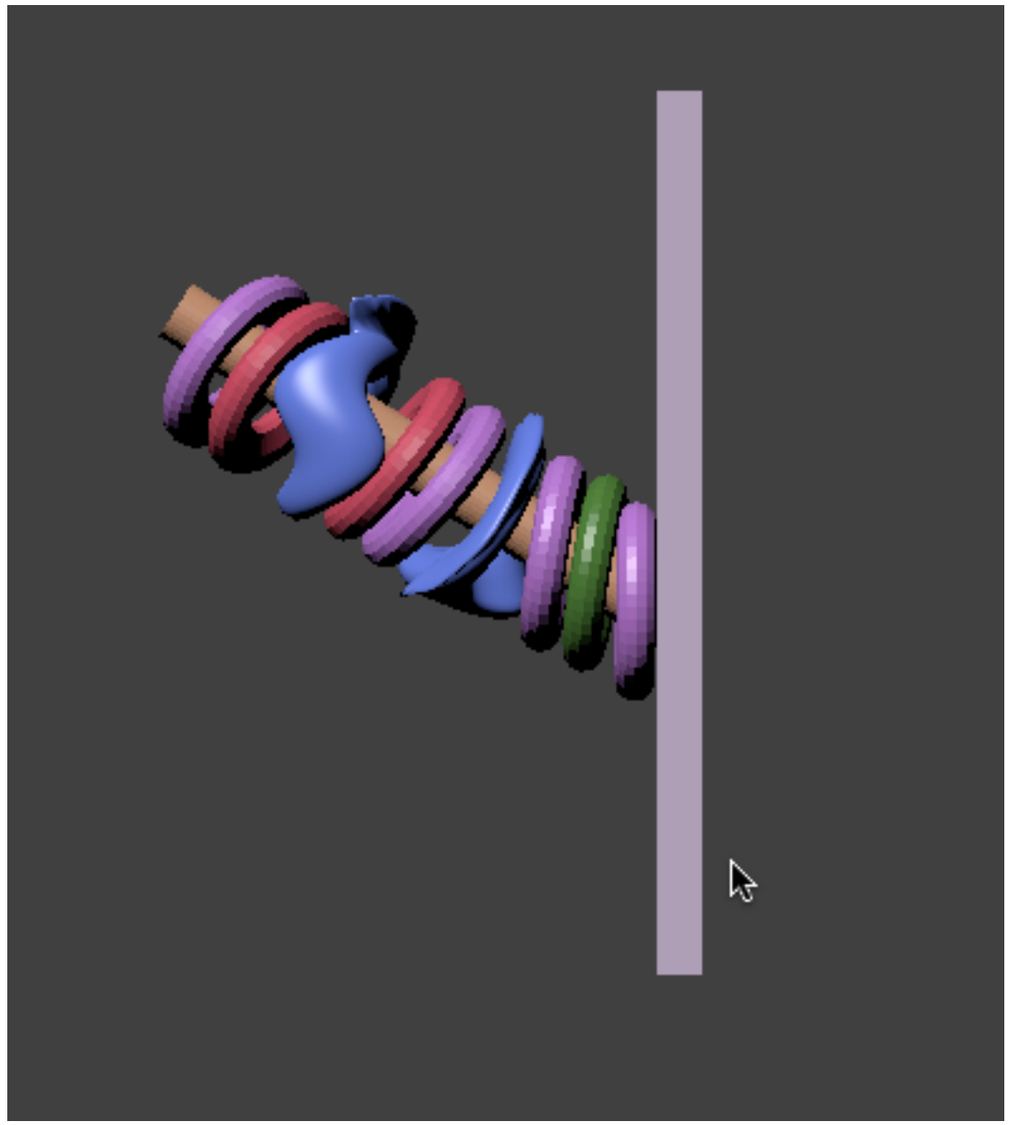
Better Representation of a vector



Order

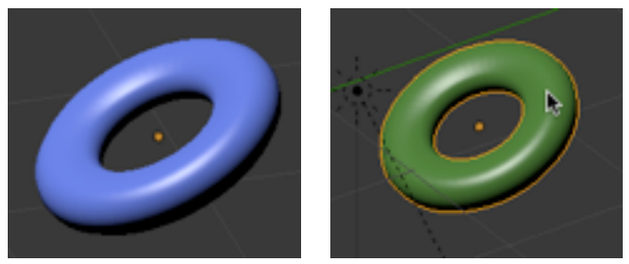
- c(2,3,4) is *not* the same as c(3,2,4)

Different types not Allowed

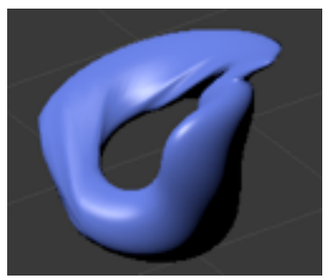


A vector cannot contain objects of different types

Type 1:



Type 2:



Examples of vectors

Use c( )

Create a vector with the three numbers 4.3, -5.2 and 6.23

Answer: c(4.3, -5.2, 6.23)

What is wrong with the vector c(5.2, “suicide”) ?

Answer: the vector contains a number and something which is not a number. All vector elements must be of the same type

Type in RStudio: c(5.2, “suicide”)

Is there an error? What happens?

Vector Elements

Consider c(5.2, -623.23, 62.1, -6.)

Each vector is composed of elements.

The vector above has four elements.
The first element is 5.2
The second element is -623.23
The third element is 62.1
The fourth element is -6.

Special uses

- 3 and c(3) are identical
- c(3) and c(c(3)) are identical
- c(c(5),c(3),4) and c(5,3,4) are identical

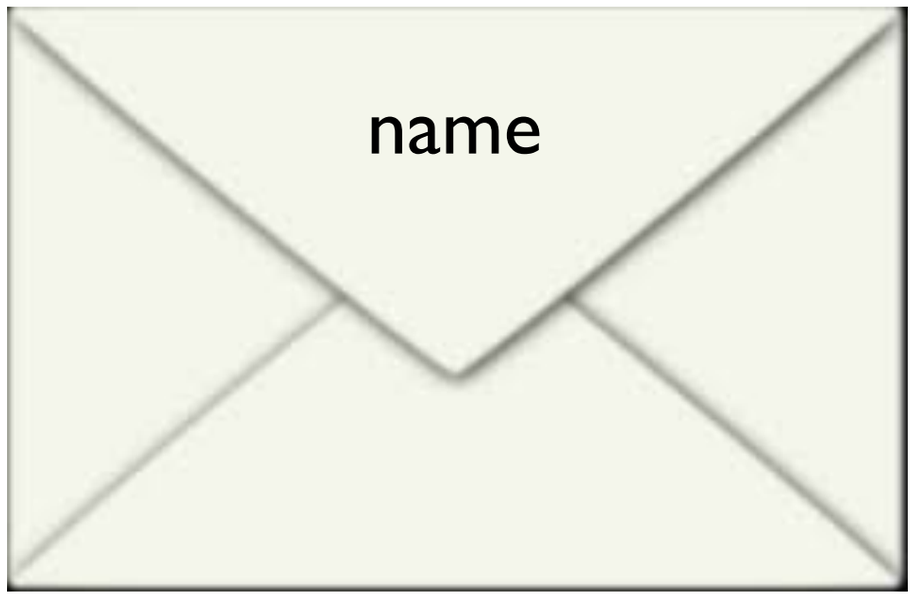
Variables

- Given a vector c(2.5, 3.6, -12.)
- How can I "carry it around"?
- How do I refer to this vector?
- I make the following statement:
  - the vector above has three elements
- What if there are three vectors on the above page? How would I refer to the particular vector I am interested in?
- To do this, I must assign a label, i.e., a *variable*

Variables

- Consider a person
- Every person has certain characteristics:
  - height, weight, eye color, etc.
- How does one refer to this person?

- Each person has a name and surname given at birth.
- Stating a person's name is a "shortcut" to describing that person by some other means



name = “Gordon”

Variable and Vectors

- Thus we will give names (also called labels) to our vectors
- Given the vector c(2.5, 3.6, -12.)
- Write:

  numbers = c(2.5, 3.6, -12.)

- Using the *variable* numbers is *equivalent* to using the vector c(2.5, 3.6, -12.)

A *Variable* is a *shortcut*

Given the vector of three grades: c(72, 86, 93)
Refer to this vector as

grade = c(72,86,93)

The first element of this vector is 72, and can be accessed as:

grade[1]

Alternatively, one can write

c(72,86,93)[1]

The vector name (grade) can be interchanged with the vector itself (c(72,86,93))

Access to vector elements

grades[1] : 1st element

grades[2] : 2nd element

grades[3] : 3rd element

grades[4] : 4th element
ERROR since there are only 3 elements

NA : **N**ot **A**vailable

```
> grades = c(93,82,75)
> grades[1]
[1] 93
> grades[2]
[1] 82
> grades[3]
[1] 75
> grades[4]
[1] NA
>
```

The [ ] operator

- To access elements from a vector, use [ ], which is a function
- This function extracts one or more elements from the vector to create a *new vector*

The [[...]] operator

- The [[...]] operator extracts *a single* element from the vector
- a = c(3,4,5)
  a[[3:4]] produces an error
  > a[[1:2]]
  Error in a[[1:2]] : attempt to select more than one element

Accessing *multiple* elements

[2:5] access elements 2 to 5

[3] access the 3rd element

```
> grade[2:3]
[1] 82 75
> grade[-2]
[1] 93 75
> grade[c(1,2,4)]
[1] 93 82 NA
> grade[c(1,3)]
[1] 93 75
```

Special case

c(2) is a vector with a single element

c(3,7) is a vector with two elements

3 is a single number

IMPORTANT

3 and c(3) are absolutely identical.

The number 3 is a shortcut for the vector c(3)

Every number is a vector with a *single* element!

A useful shortcut : seq()

```
> a = seq(2, 12, 3)

> a

[1]  2  5  8 11


# seq creates vectors with numbers at regular intervals
```

seq() is a function, which takes three arguments

?seq gives a description of this function

seq(from, to, by)

Examples

- Create a vector with the numbers 100 through 120
- Access all the even numbers in this vector

Solution

Create a vector with the numbers 100 through 120

Answer: numbers = c(100:120) # 21 elements

Access all the even numbers from this vector

Answer: numbers[c(1,3,5,7,9,11,13,15,17,19)]

Note: Given nb = c(2,3,4), the even numbers are the first and third elements of the vector.

21 elements represents a comment

Alternatively: numbers[seq(1,21,2)] reproduces all the even numbers

More detail

Create a vector with the numbers 100 through 120

Answer: numbers = c(100:120) # 21 elements

seq(1,22,2) returns 1,3,5,...,17,19, 21

numbers[seq(1,22,2)] becomes

numbers[c(1,3,5,...,17,19,21)]

which returns

c(100,102,104,...,118,120) (all the even entries)

Slight modification

Create a vector with the numbers 101 through 121

Answer: numbers = c(101:121) # 21 elements

Access all the even numbers from this vector

Answer: numbers[c(2,4,6,8,10,12,14,16,18,20)]

Alternatively: numbers[c(seq(2,20,2)]
The first element of *numbers* is odd, the 2nd element is even, as is the 4th, 6th, ... and 20th element.

Example

Consider the vector

c(-2.,4.,6.,3.,7.3, -623., 7.23)

What is the 6th element of this vector?
What is the 8th element of this vector?
What is the 1st element of this vector?

Answer

6th element: -623. # c[6]
8th element: there are only 7 elements
1st element: -2. # c[1]

Creating Your Own Vectors

- The function "c" combines things
- c(thing1, thing2, thing3,...)
- If "thing"s are the same basic type, then the result is a vector of that type.
- If "thing"s are different, but still basic types, they are all converted to the most inclusive type, usually character strings.
- ...otherwise...to be discussed later.

Type Conversion

- Integers 3,-5,10,35,40
- floats/reals: 5.237, -53.235
  - note the decimal point "."
- strings: "grades", "psych"

```
> c(5,62,-33)

[1]   5  62 -33
```

3 integers

```
 c(5,63.3,-33)

[1]   5.0  63.3 -33.0
```

one real + 2 integers

convert to 3 reals

```
> c(5,"63.3",-33.5)

[1] "5"     "63.3"  "-33.5"
```

one string + a real + an integer

convert to 3 strings

Lists

- Lists are similar to vectors, *except:*
  - lists are collections of objects of *different type*
- li = list(3, 4, “gordon”)
  li[1] ==> ?
  li[3] ==> ?
- ve = c(3,4,”gordon”)
  ve[1] ==> ?
  ve[2] ==> ?

Adding a number to a vector of numbers

- Given the vector (2,10,13), add 2 to each to produce the vector (4,12,15)

```
> y = c(2,10,13)
> y + 2
[1]  4 12 15
>
```

c(2,10,13) creates a vector of numbers
c(...) is a function
2,10,13 are the arguments to this function
c() can take any number of arguments

Adding two vectors of numbers

```
> y1 = c(2,10,13)
> y2 = c(14,-2,15)
> y1 + y2
[1] 16  8 28
```

In most cases, the vectors added to each other should have the same number of arguments.

Stated differently, the two vectors should have the same length:

length(y1) and length(y2) should be the same (equal to 3)

Reserved Words

TRUE
FALSE

reserved words
cannot be used
as variable names

if
else
repeat
while
function
for
in
next
break

NULL
Inf
NaN
NA
NA_integer_
NA_real_
NA_complex
_
NA_characte
r_

Variable Names

- Can't use reserved words
- Use meaningful names - “grades” not “g”
- Use dots to make variables more readable.
  E.g., y
  y.bar
  y.dev.squared
- *Advice:* If you can't already, learn to type.

Variables and reserved words

```
# valid expression
grades = c(2,3,5)
```

```
# invalid expression
if = c(2,3,5)
```

```
if : reserved word
```

Answers?

- 3 + 5 = ?
- 3 + 5 + 7 = ?
- 3 + 5 + 7 * 2 = ?

Operator Precedence

1) *, /

2) +, -

- Use parentheses to specify order

3 + 5 + (7 * 2) = ?

(3 + 5 + 7) * 2 = ?

Operator precedence

- Consider 3+5*2
- This could be interpreted as either
  - (3+5)*2 or
  - 3+(5*2)
- Solution: '*' has higher precedence than '+'
  - If there is a choice between executing '+' and '*', execute '*' first.

Adding two vectors of numbers

```
> y1 = c(2,10,13)
> y2 = c(14,-2,15)
> y1 + y2
[1] 16  8 28
```

Mean value of this set of numbers

```
> mean(y1+y2)
[1] 17.33333

# = 1/3*(16+8+28)
# is a comment line
```

Variance of this set of numbers

```
> var(y1+y2)
[1] 101.3333

# var = ((16-17.333)^2+(8-17.333)^2+(28-17.333)^2)/(3-1)
```

Standard deviation of this set of numbers

```
> sd(y1+y2)
[1] 10.06645
# std = sqrt(var)
```

Functions

Function= list of instructions

ingredients

+

recipe

produces a cake!

Analogy

Ingredients = function arguments

The **recipe** uses the **ingredients** to produce a **cake**

The **function body** uses its **arguments** to produce a **return value**

The function is a *container* that is a set of lines that provides the computer with instructions, just as the recipe is a list of lines with instructions to the cook

Function Arguments

- bake.cake = function(eggs,sugar,flour) {
  put "eggs" in bowl
  mix-in the flour
  add the sugar
  cake = bake mixture in oven
  return(cake)
  }

- Function name: "cake"
  Argument one: "eggs"
  Argument two: "sugar"
  Recipe (also called the function body)
  return value

Using the cake function

```
> chocolate.cake = bake.cake()

    INVALID since the function requires 3 arguments

> chocolate.cake = bake.cake("brown.eggs",
"confectionate.sugar","flour")
```

Functions

- c(), mean(), var(), and sd() are **functions**
- functions contain a set of lines that execute commands (i.e., instructions)
- These lines are not important at this stage (this is for the geeks!)
- Most of our analyses will be done by writing programs that execute a series of functions

- Let us analyze a few common functions

The function c()

Use c() to create vectors

```
grades = c(100, 90, 70)

# the vector c(100, 90, 70) has 3 arguments

Argument 1: 100
Argument 2: 90
Argument 3: 70

Arguments are separated by commas

c() takes an arbitrary number of arguments
```

The function c()

- c(3) has a single argument
- c(5,6,7,9) has four arguments
- c(seq(3,100,3)) has a single argument
- c() can have an arbitrary number of arguments

How many arguments?

c(seq(3,4,5),5,list(2,3))

c(seq(5,2),3,5,c(2,3))

c(c(2,3,4))

c(c(2,3),c(4))

The function mean()

?mean

Usage:

```
mean(x, ...)

## Default S3 method:

mean(x, trim = 0, na.rm = FALSE, ...)
```

x represents a vector of values whose mean we wish to calculate

Key point: "x represents a single argument, which is a vector"

"x" must be the first argument

... means "any number of arguments"

Mean: examples

```
> mean(5,6,7)

[1] 5
```

The result is **not** (5+6+7)/3 as you might expect. Why? Because the vector must be the first argument, which is 5!

```
> mean(c(5,6,7))

[1] 6
```

Produces the correct result
Single argument

```
> v = c(5,6,7)

> mean(v)

[1] 6
```

The first argument is "v", which is a vector of 3 elements: 5, 6, and 7

trim

- mean(x, trim = 0, na.rm = FALSE, ...)
- What is the meaning of trim?
- From the help (?mean):

trim: the fraction (0 to 0.5) of observations to be trimmed from each end of ‘x’ before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.

Next slide: an example

Trim

Generate a vector from 207 to 800 by increments of 7:

```
> v = c(207,800,7)
```

```
 [1] 207 214 221 228 235 242 249 256 263 270 277 284 291 298 305 312 319 326 333
[20] 340 347 354 361 368 375 382 389 396 403 410 417 424 431 438 445 452 459 466
[39] 473 480 487 494 501 508 515 522 529 536 543 550 557 564 571 578 585 592 599
[58] 606 613 620 627 634 641 648 655 662 669 676 683 690 697 704 711 718 725 732
[77] 739 746 753 760 767 774 781 788 795
```

```
> mean(v)
[1] 501
```

Trim

```
> w = c(1,2,3,4,5,4,3,2,1)

> mean(w)

[1] 2.777778
```

Argument 1: x

Argument 2: trim=0

```
> mean(w, trim=0.2)
```

Ignore the 3rd argument na.rm=FALSE

mean now has two arguments
Argument 1: w
Argument 2: 0.2, which refers to trim

```
mean(x, trim = 0, na.rm = FALSE, ...)
```

Trim

- Meaning of trim argument: remove from random sequence a certain percentage of elements from the front and back, and compute the mean of the elements that are left.
- Meaning of mean(x,trim=0) in the definition of *mean:* if the second argument is omitted, trim is set to zero by the computer

Unfortunately, I have not been able to get trim argument to work as I would expect.

mean and trim

```
x = 1:1000

mean(x)

y = x

y[1] = 10000

mean(y,trim=.2)        # incorrect answer 501.5
                       # but close

mean(y[2],y[999])      # 500.5
mean(y[3],y[998])      # 550.5
```

NA.rm

- Consider the sequence
  grades = c(80,90,NA,70)
- the 3rd element corresponds to a student who did not take the test

```
> mean(grades)

[1] NA
```

Arithmetic with NA returns NA

```
> mean(grades,na.r=T)

[1] 80
```

1) remove the NAs

2) compute the mean of the resulting grades: c(80,90,70)

Calling a Function

```
mean(x)          # trim=0, na.rm = F
mean(x,2)        # trim=2, na.rm = FALSE
mean(x,2,T)      # trim=2, na.rm = TRUE
                 # T works for TRUE, but be careful!

mean(x, na.rm = T)     # na.rm=  is required

mean(x,T)              # invalid
```

```
> mean(c(3,6),T)
Error in mean.default(c(3, 6),T) : 'trim' must be numeric of length one
```

Arguments can be used *out of order* by using keywords:

na.rm=T (na.rm is a keyword)

Functions

- function_name(arg1, arg2, arg3, ...)
- Example

```
> mean(c(3,4,6))
[1] 4.333333
```

Help: use ?mean

```
?mean
mean(x, trim = 0, na.rm = FALSE, ...)
```

3 main arguments: **x, trim, na.rm**
Default values: 0 for trim, FALSE for na.rm

Arguments

- Can be confusing
- ? (help) gives information on arguments to any function
- *default values* (used if a value is not specified)
- Order of arguments might be important

?getwd

getwd()

setwd(dir)

Arguments are simple:

getwd(): no arguments

setwd(..) : one argument

?dir

```
dir(path = ".", pattern = NULL, all.files = FALSE,
    full.names = FALSE, recursive = FALSE,
    ignore.case = FALSE, include.dirs = FALSE)
```

Arguments:

path, pattern, all.files, full.names ,recursive, ignore.case, include.dirs

Each argument has a default value

Example
mean and standard deviation

- Create a vector of 10 to 20 (10,12,14,...,18,20)
- Compute the mean value of this vector
- Subtract the mean from the vector
- Square the vector
- Sum all the elements of this vector
- Divide by the number of elements in this vector minus one
- Take the square root

Create a sequence 10,12,...,18,20

```
> y = seq(10,20,2)
> y
[1] 10 12 14 16 18 20
```

same as

```
> y = seq(from=10,to=20,by=2)
> y
[1] 10 12 14 16 18 20
```

seq() is a function

?seq

```
seq(from = 1, to = 1, by = ((to - from)/(length.out - 1)),
    length.out = NULL, along.with = NULL, ...)
```

Create a sequence 10,12,...,18,20

```
> y = seq(10,20,2)
> y
[1] 10 12 14 16 18 20
```

same as

```
> y = seq(from=10,to=20,by=2)
> y
[1] 10 12 14 16 18 20
```

seq(to=20,by=2,from=10)
seq(by=2,to=20,from=10)
seq(from=10,by=2,to=20)

The three statements are equivalent!

Compute the mean value of this set of numbers

```
> y.bar = sum(y)/length(y)
> y.bar
[1] 15
# = (10+12+14+16+18+20)/6
```

Subtract the mean value from each element of the vector

```
> y.dev=y-y.bar
> y.dev
[1] -5 -3 -1  1  3  5
```

Square each element of y.dev and sum up all the elements

```
> y.dev.sq = y.dev^2

> y.dev.sq
[1] 25  9  1  1  9 25

> y.dev.sq.sum = sum(y.dev.sq)

> y.dev.sq.sum
[1] 70
```

Divide y.dev.sq.sum by the length of the vector minus one and take the square root

```
> y.var =y.dev.sq.sum/(length(y.dev.sq)-1)
> y.var
[1] 14
> y.sd = sqrt(y.var)
> y.sd
[1] 3.741657
```

Collect all that we have done

```
> y = seq(10,20,2)
> y.bar = sum(y)/length(y)
> y.dev=y-y.bar
> y.dev.sq = y.dev^2
> y.dev.sq.sum = sum(y.dev.sq)
> y.var =y.dev.sq.sum/(length(y)-1)
> y.sd = sqrt(y.var)
> y.sd
[1] 3.741657
```

Easier way

```
> sd(y)
[1] 3.741657>
```

Functions used to compute the mean

c() : vector

sum() : sum its arguments

length() : argument is a vector; return the length of its argument

Information about **sd**

```
>?sd
```

Accessing Help

- Let us seek help on the function **sd**
- **?sd** or **help(sd)**
- **??sd**
  - find all the packages that contain a function called sd

List of commands starting with “sum”

```
?sum <TAB> <Up/Down>
```

Help with Functions

- Consider the function read.csv()
  - read data from comma-delimited file
  - ?read.csv (to get help)
  - ??read.csv (packages that contain read.csv)
  - args(read.csv) (arguments to function)
  - example(read.csv) (run some examples)

Help

- help(read.csv) <=====> ?read.csv
- help.search(read.csv) <=====> ??read.csv
- help.start()

Quitting R()

- q() or quit()

Help on the Web

- RSiteSearch("read.csv")
- http://rseek.org
  - find links to R-specific websites
- http://stats.stackexchange.com/
  - more information on statistics

Mailing Lists

- When in a bind, search google, which will lead you to forums, mailing lists, etc.
- Or ask a question (after reading existing information)
- Email lists are a last resort when stuck

Missing Values

```
> mean(c(NA,3,6))
[1] NA
```

NA , "Not Available", is a missing value

mean(NA,3,6) = (1/3) * (NA+3+6)
Since NA is not defined, neither is the sum!

```
> mean(c(NA,3,6), na.rm=TRUE)[1] 4.5
```

na.rm : remove missing values. The mean can now be computed

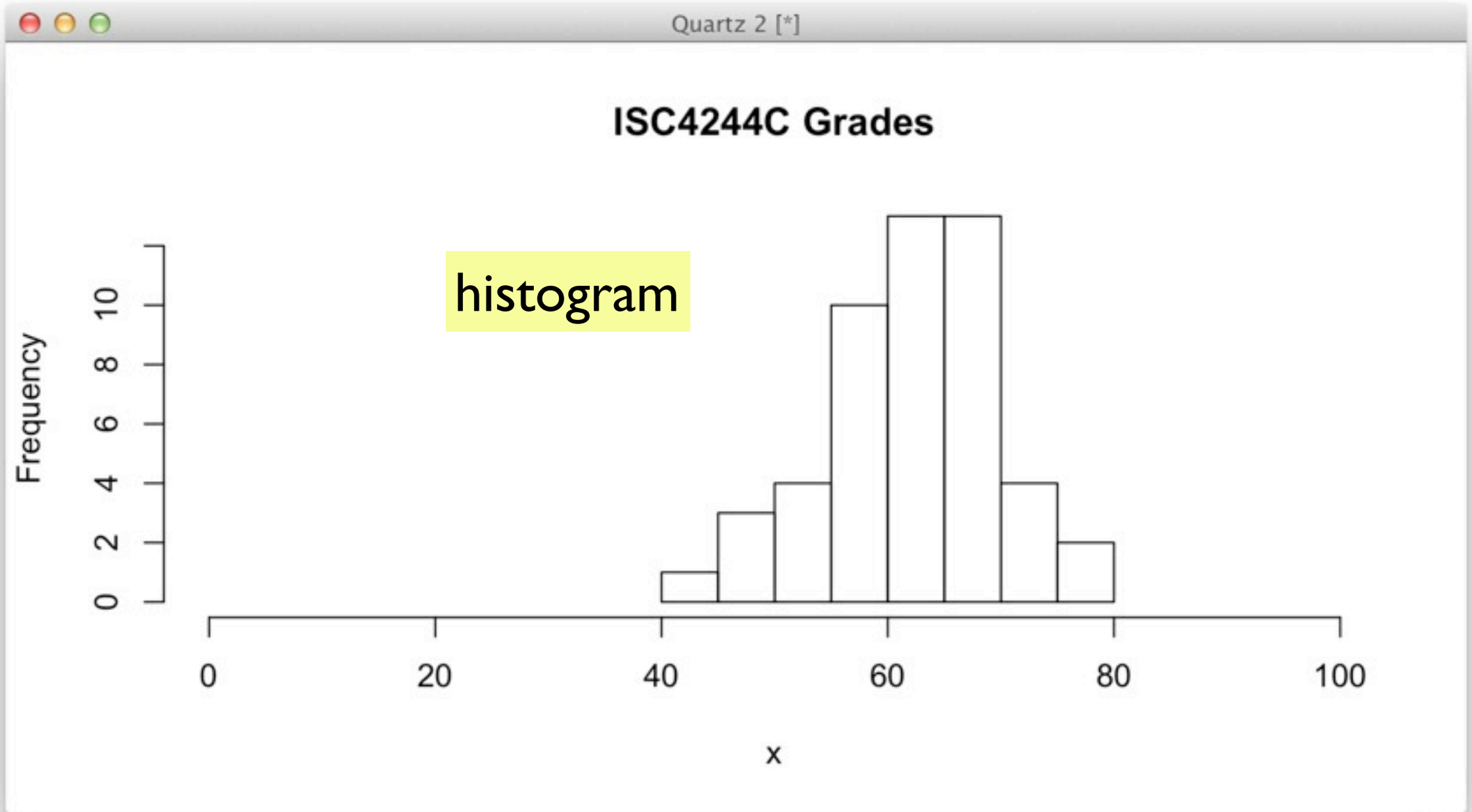
Descriptive Statistics

The quantitative study of a sample

- Mean – statistic of location
- Variance – statistic of dispersion
- Standard deviation - sqrt(variance)

Descriptive Statistics

http://en.wikipedia.org/wiki/



Descriptive Statistics