Welcome & Syllabus

Lecture 01

Dr. Colin Rundel

Course Details

Course Team

Instrutor

- Dr. Colin Rundel
  - colin.rundel@duke.edu / cr173@duke.edu / rundel@gmail.com
  - Office hours: Wednesdays 1-2 pm (in person or Zoom) or by appointment

TAs

- Sam Rosen
- Lynn Kremers

Course website(s)

- GitHub pages - https://sta523-sp25.github.io
  - HTML, PDF, and qmds of Slides
  - Readings and other notes
- Canvas - https://canvas.duke.edu/courses/61210
  - Announcements
  - Gradebook

Labs

- Attendance is expected
- Opportunity to work on course assignments with TA support
- Labs will begin this week

Assessment

This course is graded 100% on your coursework (there are no exams).

We will be assessing you based on the following assignments,

Assignment	Type	Value	n	Assigned
Homeworks	Team	30%	5/6	~ Every other week
Midterms	Individual	40%	2	~ Week 6 and 14
Project	Team	10%	1	~ Week 10
Quizzes	Individual	20%	~15	

Teams

- Team assignments
  - Roughly biweekly homework assignments
  - Open ended, ~5 - 15 hours of work
  - Peer evaluation after completion
- Expectations and roles:
  - Everyone is expected to contribute equal *effort*
  - Everyone is expected to understand *all* code turned in
  - Individual contribution evaluated by peer evaluation, commits, etc.

Collaboration policy

- Only work that is clearly assigned as team work should be completed collaboratively (Homeworks + Project).
- Individual assignments (Midterms) must be completed individually, you may not directly share or discuss answers / code with anyone other than the myself and the TAs.
- On Homeworks you should not directly share answers / code with other teams, however you are welcome to discuss the problems in general and ask for advice.

Sharing / reusing code / AI policy

- We are aware that a huge volume of code is available on the web, and many tasks may have solutions posted.
- Unless explicitly stated otherwise, this course's policy is that you may make use of any online resources (e.g. Google, StackOverflow, etc.) but you must explicitly cite where you obtained any code you directly use or use as inspiration in your solution(s).
- Any recycled code that is discovered and is not explicitly cited will be treated as plagiarism, regardless of source.
- The same applies to the use of LLM like ChatGPT, Claude, or GitHub Copilot - you are welcome to make use of these tools as the basis for your solutions but you must cite the tool when using it for significant amounts of code generation.

Brief thoughts on AI tools

- AI tools are not a replacement for understanding the material, they can be a tool to help you understand the material.
- Reaading code and writing code are both skills that take time and practice to develop - both are still essential skills.
- Nature of the tools is changing rapidly - Autocomplete vs ChatBots vs Agentic

Academic integrity

> To uphold the Duke Community Standard:
>
> - I will not lie, cheat, or steal in my academic endeavors;
> - I will conduct myself honorably in all my endeavors; and
> - I will act if the Standard is compromised.

Course Tools

Accessing RStudio Workbench

To reduce friction, the preferred method is to use the department's RStudio server(s).

To access RStudio/Posit Workbench:

1. Navigate to https://rstudio.stat.duke.edu
2. Log-in with your Duke NetID and password.

If off campus, use the VPN to create a secure connection from your computer to Duke. If you are on campus, be sure you

DSS RStudio alternatives

If you cannot access RStudio via the DSS servers:

- Make sure you are on authenticated Duke network (e.g. DukeBlue or VPN if off campus)
- Make sure you are not using a custom DNS server
  - e.g. `1.1.1.1` or `8.8.8.8`
- Use a Docker container from Duke OIT
  1. Go to https://cmgr.oit.duke.edu/ and login
  2. Select `Reserve a Container` and find a container for Sta 313
  3. Click the link under my reservations to create your environment

Local R + RStudio

If working locally you should make sure that your environment meets the following requirements:

- latest R (4.5.1)
- latest RStudio (2025.05.1+513)
- working git installation
- ability to create ssh keys (for GitHub authentication)
- *All* R packages updated to their latest version from CRAN

Support policy for local installs - we will try to help you troubleshoot if we can but reserve the right to tell you to use the dept server.

GitHub

- We will be using an organization specifically to this course github.com/sta523-sp25
- All assignments will be distributed and collected via GitHub
- All of your work and your membership (enrollment) in the organization is private
- We will be distributing a survey this weekend to collection your GitHub account names
  - Before lab you will be invited to the course organization.
- All course related repositories will be created for you

Before Friday

- Create a GitHub account if you don't have one
- Complete the course survey
- Make sure you can login in to the Department's RStudio server https://rstudio.stat.duke.edu
- Setup ssh key authentication with GitHub, see https://github.com/DukeStatSci/github_auth_guide

In R (almost) everything is a vector

Vectors

The fundamental building block of data in R are vectors (collections of related values, objects, etc).

R has two types of vectors (that everything is built on):

- atomic vectors (*vectors*)
  - homogeneous collections of the *same* type (e.g. all `true`/`false` values, all numbers, or all character strings).
- generic vectors (*lists*)
  - heterogeneous collections of *any* type of R object, even other lists (meaning they can have a hierarchical/tree-like structure).

Atomic Vectors

Atomic Vectors

R has six atomic vector types, we can check the type of any object in R using the `typeof()` function. `mode()` is a higher level abstraction used to group similar types together.

`typeof()`	`mode()`
logical	logical
double	numeric
integer	numeric
character	character
complex	complex
raw	raw

There are additional types in R, e.g. `list`, `closure`, `environment`, etc. We will see these in the next couple of lectures. See

`logical` - boolean values (`TRUE` and `FALSE`)

```
1 typeof(TRUE)
```

```
[1] "logical"
```

```
1 mode(TRUE)
```

```
[1] "logical"
```

```
1 typeof(FALSE)
```

```
[1] "logical"
```

```
1 mode(FALSE)
```

```
[1] "logical"
```

R will let you use `T` and `F` as shortcuts to `TRUE` and `FALSE`, this is a bad practice as these values are actually **global variables** that can be overwritten.

```
1 T
```

```
[1] TRUE
```

```
1 T = "FALSE"
2 T
```

```
[1] "FALSE"
```

`character` - text strings

Either single or double quotes are fine, the opening and closing quote must match.

```
1 typeof("hello")
```

```
[1] "character"
```

```
1 mode("hello")
```

```
[1] "character"
```

```
1 typeof('world')
```

```
[1] "character"
```

```
1 mode('world')
```

```
[1] "character"
```

Quote characters can be included by escaping or using a non-matching quote.

```
1 "abc'123"
```

```
[1] "abc'123"
```

```
1 "abc\"123"
```

```
[1] "abc\"123"
```

```
1 'abc"123'
```

```
[1] "abc\"123"
```

```
1 'abc\'123'
```

```
[1] "abc'123"
```

Numeric types

`double` - floating point values (these are the default numerical type)

```
1 typeof(1.33)
```

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

```
1 mode(1.33)
```

```
[1] "numeric"
```

```
1 typeof(7)
```

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

```
1 mode(7)
```

```
[1] "numeric"
```

`integer` - integer values (literals are indicated by an `L` suffix)

```
1 typeof( 7L )
```

```
[1] "integer"
```

```
1 mode( 7L )
```

```
[1] "numeric"
```

```
1 typeof( 1:3 )
```

```
[1] "integer"
```

```
1 mode( 1:3 )
```

```
[1] "numeric"
```

Combining / Concatenation

Atomic vectors can be constructed using the combine `c()` function.

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

```
[1] 1 2 3
```

```
1 c("Hello", "World!")
```

```
[1] "Hello"  "World!"
```

```
1 c(1, 1:10)
```

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

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

```
[1] 1 2 3
```

Note - atomic vectors are inherently flat / 1d, so nesting `c()` calls does not create a nested structure (when working with

Inspecting types

- `typeof(x)` - returns a character vector (length 1) of the *type* of object `x`.
- `mode(x)` - returns a character vector (length 1) of the *mode* of object `x`.

```
1 typeof(1)
```

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

```
1 mode(1)
```

```
[1] "numeric"
```

```
1 typeof(1L)
```

```
[1] "integer"
```

```
1 mode(1L)
```

```
[1] "numeric"
```

```
1 typeof("A")
```

```
[1] "character"
```

```
1 mode("A")
```

```
[1] "character"
```

```
1 typeof(TRUE)
```

```
[1] "logical"
```

```
1 mode(TRUE)
```

```
[1] "logical"
```

Type predicates

- `is.logical(x)` - returns `TRUE` if `x` has *type* `logical`.
- `is.character(x)` - returns `TRUE` if `x` has *type* `character`.
- `is.double(x)` - returns `TRUE` if `x` has *type* `double`.
- `is.integer(x)` - returns `TRUE` if `x` has *type* `integer`.
- `is.numeric(x)` - returns `TRUE` if `x` has *mode* `numeric`.

```
1 is.integer(1)
```

```
[1] FALSE
```

```
1 is.double(1)
```

```
[1] TRUE
```

```
1 is.numeric(1)
```

```
[1] TRUE
```

```
1 is.integer(1L)
```

```
[1] TRUE
```

```
1 is.double(1L)
```

```
[1] FALSE
```

```
1 is.numeric(1L)
```

```
[1] TRUE
```

```
1 is.integer(3:7)
```

```
[1] TRUE
```

```
1 is.double(3:8)
```

```
[1] FALSE
```

```
1 is.numeric(3:7)
```

```
[1] TRUE
```

Other useful predicates

- `is.atomic(x)` - returns `TRUE` if `x` is an *atomic vector*.
- `is.list(x)` - returns `TRUE` if `x` is a *list* (generic vector).
- `is.vector(x)` - returns `TRUE` if `x` is either an *atomic* or *generic* vector.

```
1 is.atomic(c(1,2,3))
```

```
[1] TRUE
```

```
1 is.atomic(list(1,2,3))
```

```
[1] FALSE
```

```
1 is.list(c(1,2,3))
```

```
[1] FALSE
```

```
1 is.list(list(1,2,3))
```

```
[1] TRUE
```

```
1 is.vector(c(1,2,3))
```

```
[1] TRUE
```

```
1 is.vector(list(1,2,3))
```

```
[1] TRUE
```

Type Coercion

R is a dynamically typed language – it will automatically convert between most types without raising warnings or errors. Keep in mind that atomic vectors must always contain values of the same type.

```
1 c(1, "Hello")
```

```
[1] "1"     "Hello"
```

```
1 c(FALSE, 3L)
```

```
[1] 0 3
```

```
1 c(1.2, 3L)
```

```
[1] 1.2 3.0
```

```
1 c(FALSE, "Hello")
```

```
[1] "FALSE" "Hello"
```

Operator coercion

Builtin operators and functions (e.g. `+`, `&`, `log()`, etc.) will generally attempt to coerce values to an appropriate type for the given operation (numeric for math, logical for logical, etc.)

```
1 3.1+1L
```

```
[1] 4.1
```

```
1 5 + FALSE
```

```
[1] 5
```

```
1 TRUE & FALSE
```

```
[1] FALSE
```

```
1 TRUE & 7
```

```
[1] TRUE
```

```
1 log(1)
```

```
[1] 0
```

```
1 log(TRUE)
```

```
[1] 0
```

```
1 TRUE | FALSE
```

```
[1] TRUE
```

```
1 FALSE | !5
```

```
[1] FALSE
```

Explicit Coercion

Most of the `is` functions we just saw have an `as` variant which can be used for *explicit* coercion.

```
1 as.logical(5.2)
```

```
[1] TRUE
```

```
1 as.character(TRUE)
```

```
[1] "TRUE"
```

```
1 as.integer(pi)
```

```
[1] 3
```

```
1 as.numeric(FALSE)
```

```
[1] 0
```

```
1 as.double("7.2")
```

```
[1] 7.2
```

```
1 as.double("one")
```

```
Warning: NAs introduced by coerc
[1] NA
```

Missing Values

Missing Values

R uses `NA` to represent missing values in its data structures, what may not be obvious is that there are different `NA`s for the different atomic types.

```
1 typeof(NA)
```

```
[1] "logical"
```

```
1 typeof(NA_character_)
```

```
[1] "character"
```

```
1 typeof(NA+1)
```

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

```
1 typeof(NA_real_)
```

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

```
1 typeof(NA+1L)
```

```
[1] "integer"
```

```
1 typeof(NA_integer_)
```

```
[1] "integer"
```

```
1 typeof(c(NA,""))
```

```
[1] "character"
```

```
1 typeof(NA_complex_)
```

```
[1] "complex"
```

NA stickiness

Because NAs represent missing values it makes sense that most calculations using them will also be missing.

```
1 + NA
```

```
[1] NA
```

```
1 / NA
```

```
[1] NA
```

```
NA * 5
```

```
[1] NA
```

```
sqrt(NA)
```

```
[1] NA
```

```
3^NA
```

```
[1] NA
```

```
sum(c(1, 2, 3, NA))
```

```
[1] NA
```

Aggregation / summarization functions (e.g. `sum()`, `mean()`, `sd()`, etc.) will often have a `na.rm` argument which *drops* the missing values from the calculation.

```
sum(c(1, 2, 3, NA), na.rm = TRUE)
```

```
[1] 6
```

```
mean(c(1, 2, 3, NA), na.rm = TRUE)
```

```
[1] 2
```

NAs are not always sticky

A useful mental model for `NA`s is to consider them as a unknown value that could take any of the possible values for a type.

For numbers or characters this isn't very helpful, but for a logical value we know that the value must either be `TRUE` or `FALSE` and we can use that when deciding what value to return.

```
1 TRUE & NA
```

```
[1] NA
```

```
1 TRUE | NA
```

```
[1] TRUE
```

```
1 FALSE & NA
```

```
[1] FALSE
```

```
1 FALSE | NA
```

```
[1] NA
```

Other Special values (double)

These are defined as part of the IEEE floating point standard (not unique to R)

- `NaN` - Not a number
- `Inf` - Positive infinity
- `-Inf` - Negative infinity

```
1 pi / 0
```

```
[1] Inf
```

```
1 0 / 0
```

```
[1] NaN
```

```
1 1/0 + 1/0
```

```
[1] Inf
```

```
1 Inf - Inf
```

```
[1] NaN
```

```
1 NaN / NA
```

```
[1] NA
```

```
1 NaN * NA
```

```
[1] NA
```

Testing for `Inf` and `NaN`

`NaN` and `Inf` there are convenience functions for testing for these types of values

```
1 is.finite(Inf)
```

```
[1] FALSE
```

```
1 is.finite(NaN)
```

```
[1] FALSE
```

```
1 is.infinite(-Inf)
```

```
[1] TRUE
```

```
1 is.infinite(NaN)
```

```
[1] FALSE
```

```
1 is.nan(Inf)
```

```
[1] FALSE
```

```
1 is.nan(NaN)
```

```
[1] TRUE
```

```
1 Inf > 1
```

```
[1] TRUE
```

```
1 -Inf > 1
```

```
[1] FALSE
```

```
1 is.finite(NA)
```

```
[1] FALSE
```

```
1 is.infinite(NA)
```

```
[1] FALSE
```

```
1 is.nan(NA)
```

```
[1] FALSE
```

Coercion for infinity and NaN

First remember that `Inf`, `-Inf`, and `NaN` are doubles, however their coercion behavior is not the same as other doubles

```
1 as.integer(Inf)
```

```
Warning: NAs introduced by coercion to integer range
[1] NA
```

```
1 as.integer(NaN)
```

```
[1] NA
```

```
1 as.logical(Inf)
```

```
[1] TRUE
```

```
1 as.logical(-Inf)
```

```
[1] TRUE
```

```
1 as.logical(NaN)
```

```
[1] NA
```

```
1 as.character(Inf)
```

```
[1] "Inf"
```

```
1 as.character(-Inf)
```

```
[1] "-Inf"
```

```
1 as.character(NaN)
```

```
[1] "NaN"
```

Exercise 1

Part 1

What is the type of the following vectors? Explain why they have that type.

```
c(1, NA+1L, "C")
c(1L / 0, NA)
c(1:3, 5)
c(3L, NaN+1L)
c(NA, TRUE)
```

Part 2

Considering only the four (common) data types, what is R's implicit type conversion hierarchy (from highest priority to lowest priority)?

05:00

Logical & Comparison operators

Logical (boolean) operators

Operator	Operation	Vectorized?
`x | y`	or	Yes
`x & y`	and	Yes
`!x`	not	Yes
`x || y`	or	No
`x && y`	and	No
`xor(x, y)`	exclusive or	Yes

Vectorized?

```
1 x = c(TRUE,FALSE,TRUE)
2 y = c(FALSE,TRUE,TRUE)
```

```
1 x | y
```

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

```
1 x & y
```

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

```
1 x || y
```

```
Error in x || y: 'length = 3' in co
```

```
1 x && y
```

```
Error in x && y: 'length = 3' in co
```

```
1 TRUE && FALSE
```

```
[1] FALSE
```

`&` and `|` are almost always going to be the right choice, the only time we use `&&` or `||` is when you need to take advantage of short-circuit evaluation.

Note previously (before R 4.3) both `||` and `&&` only used the *first* value in the vector, all other values are ignored and

Vectorization and math

Almost all of the basic mathematical operations (and many other functions) in R are vectorized.

```
1 c(1, 2, 3) + c(3, 2, 1)
```

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

```
1 c(1, 2, 3) / c(3, 2, 1)
```

```
[1] 0.3333333 1.0000000 3.0000000
```

```
1 log(c(1, 3, 0))
```

```
[1] 0.000000 1.098612     -Inf
```

```
1 sin(c(1, 2, 3))
```

```
[1] 0.8414710 0.9092974 0.1411200
```

Length coercion (aka recycling)

If the lengths of the vector do not match, then the shorter vector has its values recycled to match the length of the longer vector.

```
x = c(TRUE, FALSE, TRUE)
y = c(TRUE)
z = c(FALSE, TRUE)
```

```
x | y
```

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

```
y | z
```

```
[1] TRUE TRUE
```

```
x & y
```

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

```
y & z
```

```
[1] FALSE  TRUE
```

```
x | z
```

```
Warning in x | z: longer object length is not a multiple of shorter object
length
```

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

Length coercion and math

The same length coercion rules apply for most basic mathematical operators,

```
x = c(1, 2, 3)
y = c(5, 4)
z = 10L
```

```
x + x
```

```
[1] 2 4 6
```

```
y / z
```

```
[1] 0.5 0.4
```

```
x + z
```

```
[1] 11 12 13
```

```
log(x)+z
```

```
[1] 10.00000 10.69315 11.09861
```

```
x %% y
```

```
Warning in x%%y: longer object length is not a multiple of shorter object
length
```

```
[1] 1 2 3
```

Comparison operators

Operator	Comparison	Vectorized?
`x < y`	less than	Yes
`x > y`	greater than	Yes
`x <= y`	less than or equal to	Yes
`x >= y`	greater than or equal to	Yes
`x != y`	not equal to	Yes
`x == y`	equal to	Yes
`x %in% y`	contains	Yes (over `x`)*

Comparisons

```
1 x = c("A","B","C")
2 y = c("A")
```

```
1 x == y
```

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

```
1 x != y
```

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

```
1 x %in% y
```

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

```
1 y %in% x
```

```
[1] TRUE
```

Type coercion also applies for comparison opperators which can result in *interesting* behavior

```
1 TRUE == "TRUE"
```

```
[1] TRUE
```

```
1 FALSE == 1
```

```
[1] FALSE
```

```
1 TRUE == 1
```

```
[1] TRUE
```

```
1 TRUE == 5
```

```
[1] FALSE
```

> & < with characters

While maybe somewhat unexpected, these comparison operators can be used character values.

```
1 "A" < "B"
```

```
[1] TRUE
```

```
1 "A" > "B"
```

```
[1] FALSE
```

```
1 "A" < "a"
```

```
[1] FALSE
```

```
1 "a" > "!"
```

```
[1] TRUE
```

```
1 "Good" < "Goodbye"
```

```
[1] TRUE
```

```
1 c("Alice", "Bob", "Carol") <
```

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

Note - to better understand how this works, i.e. the ordering used, see ASCII code and https://stackoverflow.com/questions/7229408/what-are-the-r-sorting-rules-of-character-vectors