Introduction to R, Season 2

March, 2024

Table of Contents

# 1 About this Course

## 1.1 Curriculum

The course covers fundamentals of R, a high-level programming language, and use it to wrangle data for analysis and visualization.

## 1.2 Target Audience

The course is intended for researchers who want to learn coding for the first time with a data science application, or have explored programming and want to focus on fundamentals.

# 2 Intro to Computing

## 2.1 Goals of the course

* Fundamental concepts in high-level programming languages (R, Python, Julia, WDL, etc.) that is transferable: *How do programs run, and how do we solve problems using functions and data structures?*
* Beginning of data science fundamentals: *How do you translate your scientific question to a data wrangling problem and answer it?*
* 
* Figure : Data science workflow
* Find a nice balance between the two throughout the course: we will try to reproduce a figure from a scientific publication using new data.

## 2.2 What is a computer program?

* A sequence of instructions to manipulate data for the computer to execute.
* A series of translations: English <-> Programming Code for Interpreter <-> Machine Code for Central Processing Unit (CPU)

We will focus on English <-> Programming Code for R Interpreter in this class.

More importantly: **How we organize ideas <-> Instructing a computer to do something**.

## 2.3 A programming language has following elements:

* Grammar structure to construct expressions
* Combining expressions to create more complex expressions
* Encapsulate complex expressions via functions to create modular and reusable tasks
* Encapsulate complex data via data structures to allow efficient manipulation of data

## 2.4 Posit Cloud Setup

Posit Cloud/RStudio is an Integrated Development Environment (IDE). Think about it as Microsoft Word to a plain text editor. It provides extra bells and whistles to using R that is easier for the user.

Today, we will pay close attention to:

* Script editor: where sequence of instructions are typed and saved as a text document as a R program. To run the program, the console will execute every single line of code in the document.
* Console (interpreter): Instead of giving a entire program in a text file, you could interact with the R Console line by line. You give it one line of instruction, and the console executes that single line. It is what R looks like without RStudio.
* Environment: Often, code will store information *in memory*, and it is shown in the environment. More on this later.

## 2.5 Using Quarto for your work

Why should we use Quarto for data science work?

* Encourages reproducible workflows
* Code, output from code, and prose combined together
* Extensions to Python, Julia, and more.

More options and guides can be found in [Introduction to Quarto](https://quarto.org/docs/get-started/hello/rstudio.html) .

## 2.6 Grammar Structure 1: Evaluation of Expressions

* **Expressions** are be built out of **operations** or **functions**.
* Operations and functions combine **data types** to return another data type.
* We can combine multiple expressions together to form more complex expressions: an expression can have other expressions nested inside it.

For instance, consider the following expressions entered to the R Console:

18 + 21

## [1] 39

max(18, 21)

## [1] 21

max(18 + 21, 65)

## [1] 65

18 + (21 + 65)

## [1] 104

nchar("ATCG")

## [1] 4

Here, our input **data types** to the operation are **numeric** in lines 1-4 and our input data type to the function is **character** in line 5.

Operations are just functions in hiding. We could have written:

sum(18, 21)

## [1] 39

sum(18, sum(21, 65))

## [1] 104

Remember the function machine from algebra class? We will use this schema to think about expressions.



Figure : Function machine from algebra class.

If an expression is made out of multiple, nested operations, what is the proper way of the R Console interpreting it? Being able to read nested operations and nested functions as a programmer is very important.

3 \* 4 + 2

## [1] 14

3 \* (4 + 2)

## [1] 18

Lastly, a note on the use of functions: a programmer should not need to know how the function is implemented in order to use it - this emphasizes [abstraction and modular thinking](#X4289ef2db5109e8791918a3c7b0cce6f2f89fd2), a foundation in any programming language.

### 2.6.1 Data types

Here are some data types that we will be using in this course:

* **Numeric**: 18, 21, 65, 1.25
* **Character**: “ATCG”, “Whatever”, “948-293-0000”
* **Logical**: TRUE, FALSE

## 2.7 Grammar Structure 2: Storing data types in the environment

To build up a computer program, we need to store our returned data type from our expression somewhere for downstream use. We can assign a variable to it as follows:

x = 18 + 21

If you enter this in the Console, you will see that in the Environment, the variable x has a value of 39.

### 2.7.1 Execution rule for variable assignment

Evaluate the expression to the right of =.

Bind variable to the left of = to the resulting value.

The variable is stored in the environment.

<- is okay too!

The environment is where all the variables are stored, and can be used for an expression anytime once it is defined. Only one unique variable name can be defined.

The variable is stored in the working memory of your computer, Random Access Memory (RAM). This is temporary memory storage on the computer that can be accessed quickly. Typically a personal computer has 8, 16, 32 Gigabytes of RAM. When we work with large datasets, if you assign a variable to a data type larger than the available RAM, it will not work. More on this later.

Look, now x can be reused downstream:

x - 2

## [1] 37

y = x \* 2

## 2.8 Grammar Structure 3: Evaluation of Functions

A function has a **function name**, **arguments**, and **returns** a data type.

### 2.8.1 Execution rule for functions:

Evaluate the function by its arguments, and if the arguments are functions or contains operations, evaluate those functions or operations first.

The output of functions is called the **returned value**.

sqrt(nchar("hello"))

## [1] 2.236068

(nchar("hello") + 4) \* 2

## [1] 18

## 2.9 Tips on writing your first code

Computer = powerful + stupid

Even the smallest spelling and formatting changes will cause unexpected output and errors!

* Write incrementally, test often
* Check your assumptions, especially using new functions, operations, and new data types.
* Live environments are great for testing, but not great for reproducibility.
* Ask for help!

# 3 Intro to Computing Exercises

If you need to clear your environment and start from the beginning, this code chunk will do that:

rm(list = ls())

## 3.1 Part 1: Expressions on numerics, characters, and logical.

Recall that we have looked at the first two data types, and you will look at the third data type briefly today:

* **Numeric**: Contains any numeric values, include decimals. For instance, -123.2.
* **Character**: Contains a series of letters and symbols, contained in a quote. For instance, "hello!"
* **Logical**: Contains only TRUE or FALSE values. It can be equivalently represented by T or F.

We will variables chr1 and chr2 to both have values of 2. This represents the number of chromosome copies you have in a cell for chromosomes 1 and 2, respectively.

chr1 = 2  
chr2 = 2

Use the function class() on either variables. What data type does it return, and what do you think it is telling you? The class() function is very useful to understand what data type your variable is. Also notice that the data type in the argument of the function does not have to be same data type as the returned value from the function!

Then, change the value of chr1 to be 3, representing 3 chromosome copies for chromosome 1.

Then, create variable chr3 to be assigned to chr1.

Change the value of chr1 back to 2. What happens to value of chr3?

Even though you have set chr3 to be equal to chr1 in the previous code chunk, chr3 does not change. This illustrates an important point: when you make an assignment via =, the R Console makes a copy of the data type and store its value independently in the environment (This is not always true in every programming language!).

Now, do the following: triple the value of chr1, and divide the value of chr3 by one half.

Tip: Is there a way of writing it by referencing the variable’s current value? For instance, instead of chr1 = 6, consider the syntax x = x \* 3. This way of referencing the variable’s current value is extremely important in writing code that is flexible to a variety of inputs. You do not have to know what the value of chr1 is to triple its value.

Write an expression that uses chr1, chr2, and chr3 in one line of code, and store the resulting value in a new variable of your own choice. Think carefully about the order of operations or function you use and check that the R Console is interpreting it correctly.

Take what you just wrote above, and now add some parenthesizes to change the order of operation. Does the resulting value change as you expect?

Here is a weird expression:

sqrt(chr1 \* abs(-chr2 + chr1)) + nchar("chromosome")

## [1] 10

Explain in words how the R Console parse out this expression to give the resulting value. The sqrt() function computes the square root of a number, and abs() function computes the absolute value of a number. Try ?sqrt and ?abs in the R console for more information.

In your own words:

## 3.2 Part 2: Brief introduction to vectors

Suppose you want to store information about the number of chromosome copies for chromosomes 1 to 22. Defining chr4, chr5, etc. seems repetitive. *A good rule of thumb in programming is Don’t Repeat Yourself (DRY). If you do, find a way to automate it!*

We have so far looked at **data types**, which so far includes **numeric** and **character**. Here, we introduce **data structures**, which store information about data types.

The **vector** is a data structure that stores many elements of the *same data type*. Each *element* of a vector contains a single data type, and there is no limit on how big a vector can be, as long the memory use of it is within the computer’s memory (RAM).

To create a vector, we use the combine c() function to combine individual data type elements together:

chrNum = c(2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2)

That was repetitive. Using DRY rule of thumb, we find a function to automate this:

chrNum = rep(2, 22)

Use the length() function on chrNum to see what the length of this vector is.

Now, we introduce a new operation: accessing elements of a vector using **brackets**. We will talk about this in much more detail - the point of the exercise is to get a taste of what is coming up next.

If we want to access the 3rd element of this vector:

chrNum[3]

## [1] 2

Now, modify the third element of this chrNum vector to be doubled. Also, modify chrNum so that the 21th element of chrNum is the sum of chrNum[2], chrNum[3], and chrNum[4].

Last thing about vector for now: If we want to access multiple elements of chrNum, we first specify a numeric vector the elements we want to access. This is called the **indexing vector**. Then, we put the indexing vector within the brackets for chrNum to access it.

If we want the 2nd and 4th elements:

indexing = c(2, 4)  
chrNum[indexing]

## [1] 2 2

Often, we do it without storing the indexing vector as a variable.

chrNum[c(2, 4)]

## [1] 2 2

Your turn: access the 10th and 11th elements of chrNum using an indexing vector:

Now try the following: access chromosomes 1 to 10. Instead of writing out 1 to 10 for your indexing vector, use the sequence seq() function: try seq(1, 10) to create your indexing vector. You could also use the operation 1:10 instead of seq() as an alternative:

## 3.3 You are done!

Change the logical value of this variable to TRUE. That’s all you need to know about **logicals** for now!

finished\_with\_exercise1 = FALSE

## 3.4 Feedback!

How many hours did you spend on this exercise?

time\_spent = 0

If you worked with other peers, write their names down in the following character vector: each element is one person’s name.

peers = c("Myself")

# 4 Working with data structures

## 4.1 Vectors

In the first exercise, you started to explore **data structures**, which store information about data types. You played around with **vectors**, which is a ordered collection of a data type. Each *element* of a vector contains a data type, and there is no limit on how big a vector can be, as long the memory use of it is within the computer’s memory (RAM).

We can now store a vast amount of information in a vector, and assign it to a single variable. We can now use operations and functions on a vector, modifying many elements within the vector at once! This fits with the feature of “encapsulate complex data via data structures to allow efficient manipulation of data” described in the first lesson!

We often create vectors using the combine function, c() :

staff = c("chris", "shasta", "jeff")  
chrNum = c(2, 3, 1)

If we try to create a vector with mixed data types, R will try to make them be the same data type, or give an error:

staff = c("chris", "shasta", 123)  
staff

## [1] "chris" "shasta" "123"

Our numeric got converted to character so that the entire vector is all characters.

### 4.1.1 Using operations on vectors

Recall from the first class:

* Expressions are be built out of **operations** or **functions**.
* Operations and functions combine **data types** to return another data type.

Now that we are working with data structures, the same principle applies:

* Operations and functions combine **data structures** to return another data structure (or data type!).

What happens if we use some familiar operations we used for numerics on a numerical vector? If we multiply a numerical vector by a numeric, what do we get?

chrNum = chrNum \* 3  
chrNum

## [1] 6 9 3

All of chrNum’s elements tripled! Our multiplication operation, when used on a *numeric vector with a numeric*, has a *new* meaning: it multiplied all the elements by 3. Multiplication is an operation that can be used for multiple data types or data structures: we call this property **operator overloading**. Here’s another example: *numeric vector multiplied by another numeric vector*:

chrNum \* c(2, 2, 0)

## [1] 12 18 0

but there are also limits: a numeric vector added to a character vector creates an error:

#chrNum + staff

When we work with operations and functions, we must be mindful what inputs the operation or function takes in, and what outputs it gives, no matter how “intuitive” the operation or function name is.

### 4.1.2 Subsetting vectors explicitly

In the exercise this past week, you looked at a new operation to subset elements of a vector using brackets.

Inside the bracket is either a single numeric value or an a **numerical indexing vector** containing numerical values. They dictate which elements of the vector to return.

staff[2]

## [1] "shasta"

staff[c(1, 2)]

## [1] "chris" "shasta"

small\_staff = staff[c(1, 2)]

In the last line, we created a new vector small\_staff that is a subset of the staff given the indexing vector c(1, 2). We have three vectors referenced in one line of code. This is tricky and we need to always refer to our rules step-by-step: evaluate the expression right of the =, which contains a vector bracket. Follow the rule of the vector bracket. Then store the returning value to the variable left of =.

Alternatively, instead of using numerical indexing vectors, we can use a **logical indexing vector**. The logical indexing vector must be the *same length* as the vector to be subsetted, with TRUE indicating an element to keep, and FALSE indicating an element to drop. The following block of code gives the same value as before:

staff[c(TRUE, FALSE, FALSE)]

## [1] "chris"

staff[c(TRUE, TRUE, FALSE)]

## [1] "chris" "shasta"

small\_staff = staff[c(TRUE, TRUE, FALSE)]

### 4.1.3 Subsetting vectors implicitly

Here are two applications of subsetting on vectors that need distinction to write the correct code:

1. **Explicit subsetting**: Suppose someone approaches you a 100-length vector of people’s ages, and say that they want to subset to the first 10 elements.
2. **Implicit subsetting**: Suppose someone approaches you a 100-length vector of people’s ages, and say that they want to subset to elements < 18 age.

We already know how to explicitly subset:

set.seed(123) #don't worry about this function  
age = round(runif(100, 1, 100)) #don't worry about these functions  
first\_ten\_age = age[1:10]

For implicit subsetting, we don’t know which elements to select off the top of our head! If we know which elements have less than 18, then we can give the elements for an explicit subset. Therefore, we need to create a logical indexing vector using a **comparison operator**:

indexing\_vector = age < 18  
indexing\_vector

## [1] FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE  
## [13] FALSE FALSE TRUE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE  
## [25] FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE TRUE FALSE  
## [37] FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE TRUE TRUE FALSE FALSE  
## [49] FALSE FALSE TRUE FALSE FALSE TRUE FALSE FALSE TRUE FALSE FALSE FALSE  
## [61] FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  
## [73] FALSE TRUE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE  
## [85] TRUE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE  
## [97] FALSE TRUE FALSE FALSE

The comparison operator < compared the numeric value of age to see which elements of age is less than 18, and then returned a logical vector that has TRUE if age is less than 18 at that element and FALSE otherwise.

Then,

age\_young = age[indexing\_vector]  
age\_young

## [1] 6 11 5 16 3 15 16 15 6 13 14 10 1 12 11 14 10

We could have done this all in one line without storing the indexing vector as a variable in the environment:

age\_young = age[age < 18]

We have the following comparison operators in R:

< less than

<= less or equal than

== equal to

!= not equal to

> greater than

>= greater than or equal to

You can also put these comparison operators together to form more complex statements, which you will explore in this week’s exercise.

Another example:

age\_90 = age[age == 90]  
age\_90

## [1] 90 90 90

age\_not\_90 = age[age != 90]  
age\_not\_90

## [1] 29 79 41 88 94 6 53 89 56 46 96 46 68 58 11 25 5 33 95 89 70 64 99 66 71  
## [26] 55 60 30 16 96 69 80 3 48 76 22 32 24 15 42 42 38 16 15 24 47 27 86 6 45  
## [51] 80 13 57 21 14 76 38 67 10 39 28 82 45 81 81 80 45 76 63 71 1 48 23 39 62  
## [76] 36 12 25 67 42 79 11 44 99 89 89 18 14 66 35 66 33 20 78 10 47 52

For most of our subsetting tasks on vectors (and dataframes below), we will be encouraging implicit subsetting. The power of implicit subsetting is that you don’t need to know what your vector contains to do something with it! This technique is related to *abstraction* in programming mentioned in the first lesson: by using expressions to find the specific value you are interested instead of *hard-coding* the value explicitly, it generalizes your code to handle a wider variety of situations.

## 4.2 Dataframes

Before we dive into dataframes, check that the tidyverse package is properly installed by loading it in your R Console:

library(tidyverse)

## Warning: package 'tidyverse' was built under R version 4.0.3

## Warning: package 'purrr' was built under R version 4.0.5

## Warning: package 'stringr' was built under R version 4.0.3

Here is the data structure you have been waiting for: the **dataframe**. A dataframe is a spreadsheet such that each column must have the same data type. Think of a bunch of vectors organized as columns, and you get a dataframe.

For the most part, we load in dataframes from a file path (although they are sometimes created by combining several vectors of the same length, but we won’t be covering that here):

load(url("https://github.com/fhdsl/S1\_Intro\_to\_R/raw/main/classroom\_data/CCLE.RData"))

### 4.2.1 Using functions and operations on dataframes

We can run some useful functions on dataframes to get some useful properties, similar to how we used length() for vectors:

nrow(metadata)

## [1] 1864

ncol(metadata)

## [1] 30

dim(metadata)

## [1] 1864 30

colnames(metadata)

## [1] "ModelID" "PatientID" "CellLineName"   
## [4] "StrippedCellLineName" "Age" "SourceType"   
## [7] "SangerModelID" "RRID" "DepmapModelType"   
## [10] "AgeCategory" "GrowthPattern" "LegacyMolecularSubtype"  
## [13] "PrimaryOrMetastasis" "SampleCollectionSite" "Sex"   
## [16] "SourceDetail" "LegacySubSubtype" "CatalogNumber"   
## [19] "CCLEName" "COSMICID" "PublicComments"   
## [22] "WTSIMasterCellID" "EngineeredModel" "TreatmentStatus"   
## [25] "OnboardedMedia" "PlateCoating" "OncotreeCode"   
## [28] "OncotreeSubtype" "OncotreePrimaryDisease" "OncotreeLineage"

The last function, colnames() returns a character vector of the column names of the dataframe. This is an important property of dataframes that we will make use of to subset on it.

We introduce an operation for dataframes: the dataframe$column\_name operation selects for a column by its column name and returns the column as a vector. For instance:

metadata$OncotreeLineage[1:5]

## [1] "Ovary/Fallopian Tube" "Myeloid" "Bowel"   
## [4] "Myeloid" "Myeloid"

metadata$Age[1:5]

## [1] 60 36 72 30 30

We treat the resulting value as a vector, so we can perform implicit subsetting:

metadata$OncotreeLineage[metadata$OncotreeLineage == "Myeloid"]

## [1] "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid"  
## [8] "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid"  
## [15] "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid"  
## [22] "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid"  
## [29] "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid"  
## [36] "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid"  
## [43] "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid"  
## [50] "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid"  
## [57] "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid"  
## [64] "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid"  
## [71] "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid"

The bracket operation [ ] on a dataframe can also be used for subsetting. dataframe[row\_idx, col\_idx] subsets the dataframe by a row indexing vector row\_idx, and a column indexing vector col\_idx.

metadata[1:5, c(1, 3)]

## ModelID CellLineName  
## 1 ACH-000001 NIH:OVCAR-3  
## 2 ACH-000002 HL-60  
## 3 ACH-000003 CACO2  
## 4 ACH-000004 HEL  
## 5 ACH-000005 HEL 92.1.7

We can refer to the column names directly:

metadata[1:5, c("ModelID", "CellLineName")]

## ModelID CellLineName  
## 1 ACH-000001 NIH:OVCAR-3  
## 2 ACH-000002 HL-60  
## 3 ACH-000003 CACO2  
## 4 ACH-000004 HEL  
## 5 ACH-000005 HEL 92.1.7

We can leave the column index or row index empty to just subset columns or rows.

metadata[1:5, ]

## ModelID PatientID CellLineName StrippedCellLineName Age SourceType  
## 1 ACH-000001 PT-gj46wT NIH:OVCAR-3 NIHOVCAR3 60 Commercial  
## 2 ACH-000002 PT-5qa3uk HL-60 HL60 36 Commercial  
## 3 ACH-000003 PT-puKIyc CACO2 CACO2 72 Commercial  
## 4 ACH-000004 PT-q4K2cp HEL HEL 30 Commercial  
## 5 ACH-000005 PT-q4K2cp HEL 92.1.7 HEL9217 30 Commercial  
## SangerModelID RRID DepmapModelType AgeCategory GrowthPattern  
## 1 SIDM00105 CVCL\_0465 HGSOC Adult Adherent  
## 2 SIDM00829 CVCL\_0002 AML Adult Suspension  
## 3 SIDM00891 CVCL\_0025 COAD Adult Adherent  
## 4 SIDM00594 CVCL\_0001 AML Adult Suspension  
## 5 SIDM00593 CVCL\_2481 AML Adult Mixed  
## LegacyMolecularSubtype PrimaryOrMetastasis SampleCollectionSite  
## 1 Metastatic ascites  
## 2 Primary haematopoietic\_and\_lymphoid\_tissue  
## 3 Primary Colon  
## 4 Primary haematopoietic\_and\_lymphoid\_tissue  
## 5 bone\_marrow  
## Sex SourceDetail LegacySubSubtype CatalogNumber  
## 1 Female ATCC high\_grade\_serous HTB-71  
## 2 Female ATCC M3 CCL-240  
## 3 Male ATCC HTB-37  
## 4 Male DSMZ M6 ACC 11  
## 5 Male ATCC M6 HEL9217  
## CCLEName COSMICID PublicComments  
## 1 NIHOVCAR3\_OVARY 905933   
## 2 HL60\_HAEMATOPOIETIC\_AND\_LYMPHOID\_TISSUE 905938   
## 3 CACO2\_LARGE\_INTESTINE NA   
## 4 HEL\_HAEMATOPOIETIC\_AND\_LYMPHOID\_TISSUE 907053   
## 5 HEL9217\_HAEMATOPOIETIC\_AND\_LYMPHOID\_TISSUE NA   
## WTSIMasterCellID EngineeredModel TreatmentStatus OnboardedMedia PlateCoating  
## 1 2201 MF-001-041 None  
## 2 55 MF-005-001 None  
## 3 NA Unknown MF-015-009 None  
## 4 783 Post-treatment MF-001-001 None  
## 5 NA MF-001-001 None  
## OncotreeCode OncotreeSubtype OncotreePrimaryDisease  
## 1 HGSOC High-Grade Serous Ovarian Cancer Ovarian Epithelial Tumor  
## 2 AML Acute Myeloid Leukemia Acute Myeloid Leukemia  
## 3 COAD Colon Adenocarcinoma Colorectal Adenocarcinoma  
## 4 AML Acute Myeloid Leukemia Acute Myeloid Leukemia  
## 5 AML Acute Myeloid Leukemia Acute Myeloid Leukemia  
## OncotreeLineage  
## 1 Ovary/Fallopian Tube  
## 2 Myeloid  
## 3 Bowel  
## 4 Myeloid  
## 5 Myeloid

head(metadata[, c("ModelID", "CellLineName")])

## ModelID CellLineName  
## 1 ACH-000001 NIH:OVCAR-3  
## 2 ACH-000002 HL-60  
## 3 ACH-000003 CACO2  
## 4 ACH-000004 HEL  
## 5 ACH-000005 HEL 92.1.7  
## 6 ACH-000006 MONO-MAC-6

The bracket operation on a dataframe can be difficult to interpret because multiple expression for the row and column indicies is a lot of information for one line of code. You will see easier-to-read functions for dataframe subsetting in the next lesson.

Lastly, try running View(metadata) in RStudio Console…whew, a nice way to examine your dataframe like a spreadsheet program!

# 5 Working with Data Structures Exercises

If you need to clear your environment and start from the beginning, this code chunk will do that:

rm(list = ls())

## 5.1 Recap from class: subsetting explicitly and implicitly

In class, we looked at a numerical vector similar to this:

age = c(89, 70, 64, 99, 66, 71, 55, 60, 30, 16)

We could subset age **explicitly** two ways. Suppose we want to subset the 1st and 5th, and 9th elements. One can do it with numerical indexing vectors:

age[c(1, 5, 9)]

## [1] 89 66 30

or by **logical indexing vectors**:

age[c(TRUE, FALSE, FALSE, FALSE, TRUE, FALSE, FALSE, FALSE, TRUE, FALSE)]

## [1] 89 66 30

and you can do it in one step as we have done so, or two steps by storing the indexing vector as a variable. *Either ways is fine.*

num\_idx = c(1, 5, 9)  
age[num\_idx]

## [1] 89 66 30

logical\_idx = c(TRUE, FALSE, FALSE, FALSE, TRUE, FALSE, FALSE, FALSE, TRUE, FALSE)  
age[logical\_idx]

## [1] 89 66 30

Now, let’s subset this vector **implicitly**, in 3 steps:

1. Come up with a criteria for subsetting: “I want to subset to values greater than 50”.
2. We can use a **comparison operator** to create a **logical indexing vector** that fits this criteria.

age > 50

## [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE FALSE FALSE

1. Use this logical indexing vector to subset.

age[age > 50]

## [1] 89 70 64 99 66 71 55 60

And you are done.

Alternatively, you could have done this in two steps:

another\_idx = age > 50  
age[another\_idx]

## [1] 89 70 64 99 66 71 55 60

On to practicing yourself!

## 5.2 Part 1: Comparison operators on data types

In class, we looked at comparison operators on vectors to create logical indexing vectors. We are going to first take a step back and see how comparison operators work on simpler data types. Here are the comparison operators common in R:

< less than

<= less or equal than

== equal to

!= not equal to

> greater than

>= greater than or equal to

Consider the following variables:

chr1 = 2  
chr2 = 3  
chr3 = 1  
staff1 = "chris"  
staff2 = "shasta"

We saw how we used comparison operators on vectors, but they can be used for any singular data type:

Using comparison operators, write 3 expressions so that the returned value is TRUE. I’ll give one example:

staff1 != "jeff"

## [1] TRUE

Using comparison operators, write 3 expressions so that the returned value is FALSE:

chr1 > 2

## [1] FALSE

We can combine comparison operators together to form more complex statements:

The “and” & operator returns TRUE if both inputs have values of TRUE, and FALSE otherwise:

a = TRUE  
b = TRUE  
c = FALSE  
d = FALSE  
a & b

## [1] TRUE

c & d

## [1] FALSE

a & c

## [1] FALSE

b & d

## [1] FALSE

Therefore, we can create compound comparison operations like this:

chr1 > chr3 & chr1 < chr2

## [1] TRUE

Similarly, the “or” | operator returns TRUE if at least one of the inputs are TRUE. Otherwise it returns FALSE.

a | b

## [1] TRUE

c | d

## [1] FALSE

a | c

## [1] TRUE

b | d

## [1] TRUE

Create 2 compound comparison operations using | so that one returns TRUE and the other returns FALSE. I’ll give you one example:

chr1 > chr3 | chr1 < chr2

## [1] TRUE

You can extend these compound statements to contain multiple & and | operations mixed together, and parentheses should be used to clarify your order of operation.

**Now, let’s use comparison operators on vectors.**

We create the age vector from our lesson.

set.seed(123) #don't worry about this function  
age = round(runif(20, 1, 100)) #don't worry about these functions

Create a compound comparison operation that looks returns a logical vector so that the elements are TRUE if age is between 18 *and* 65.

## 5.3 Part 2: Subsetting vectors

Let’s practice explicit subsetting. Can you subset for the first 10 elements of age explicitly?

We will now look at implicit subsetting using with a criteria of interest: “We want to subset age to have values less than 21 *or* greater than 40”.

Then, the next step is to create the logical indexing vector via compound comparison operations. You should use the or | operation:

The final step is to use this logical indexing vector to subset:

Let’s do another practice: Can you subset for all age greater than 65?

## 5.4 Part 3: Dataframes

library(tidyverse)  
load(url("https://github.com/fhdsl/Intro\_to\_R/raw/main/classroom\_data/CCLE.RData"))

Use the colnames, dim, ncol, and nrow functions to explore the basic properties of the dataframe metadata:

Using the $ operator, subset OncotreeLineage column from the metadata dataframe and store the result as a character vector called lineage:

Then, subset for the first 10 elements of lineage explicitly:

Create a logical indexing vector that gives TRUE if an element of lineage is “Myeloid”, and FALSE otherwise. You will need to use the == comparison operation. Store it as the variable myeloid\_idx.

We can now use this logical indexing vector to subset on the *rows* of metadata. Take a look (you need to remove the # comments the code):

#myeloid\_metadata = metadata[myeloid\_idx ,]  
#View(myeloid\_metadata)

## 5.5 You are done!

Change the logical value of this variable to TRUE. That’s all you need to know about **logicals** for now!

finished\_with\_exercise1 = FALSE

## 5.6 Feedback!

How many hours did you spend on this exercise?

time\_spent = 0

If you worked with other peers, write their names down in the following character vector: each element is one person’s name.

peers = c("Myself")

# 6 Data Wrangling with Tidy Data, Part 1

## 6.1 Data Science Workflow



We are now equipped with enough fundamental programming skills to apply it to various steps in the data science workflow. We start with *Transform* and *Visualize* with the assumption that our data is in a nice, “Tidy format”. First, we need to understand what it means for a data to be “Tidy”.

## 6.2 Tidy Data

Here, we describe a standard of organizing data. It is important to have standards, as it facilitates a consistent way of thinking about data organization and building tools (functions) that make use of that standard. The principles of **tidy data**, developed by Hadley Wickham:

1. Each variable must have its own column.
2. Each observation must have its own row.
3. Each value must have its own cell.

If you want to be technical about what variables and observations are, Hadley Wickham describes:

A **variable** contains all values that measure the same underlying attribute (like height, temperature, duration) across units. An **observation** contains all values measured on the same unit (like a person, or a day, or a race) across attributes.



Figure : A tidy dataframe.

## 6.3 Examples and counter-examples of Tidy Data:

Consider the following three datasets, which all contain the exact same information:

table1

## # A tibble: 6 × 4  
## country year cases population  
## <chr> <int> <int> <int>  
## 1 Afghanistan 1999 745 19987071  
## 2 Afghanistan 2000 2666 20595360  
## 3 Brazil 1999 37737 172006362  
## 4 Brazil 2000 80488 174504898  
## 5 China 1999 212258 1272915272  
## 6 China 2000 213766 1280428583

This table1 satisfies the the definition of Tidy Data. The observation is a country’s year, and the variables are attributes of each country’s year.

head(table2)

## # A tibble: 6 × 4  
## country year type count  
## <chr> <int> <chr> <int>  
## 1 Afghanistan 1999 cases 745  
## 2 Afghanistan 1999 population 19987071  
## 3 Afghanistan 2000 cases 2666  
## 4 Afghanistan 2000 population 20595360  
## 5 Brazil 1999 cases 37737  
## 6 Brazil 1999 population 172006362

Something is strange able table2. The observation is still a country’s year, but “type” and “count” are not clear attributes of each country’s year.

table3

## # A tibble: 6 × 3  
## country year rate   
## \* <chr> <int> <chr>   
## 1 Afghanistan 1999 745/19987071   
## 2 Afghanistan 2000 2666/20595360   
## 3 Brazil 1999 37737/172006362   
## 4 Brazil 2000 80488/174504898   
## 5 China 1999 212258/1272915272  
## 6 China 2000 213766/1280428583

In table3, we have multiple values for each cell under the “rate” column.

## 6.4 Our working Tidy Data: DepMap Project

The [Dependency Map project](https://depmap.org/portal/) is a multi-omics profiling of cancer cell lines combined with functional assays such as CRISPR and drug sensitivity to help identify cancer vulnerabilities and drug targets. Here are some of the data that we have public access. We have been looking at the metadata since last session.

* Metadata
* Somatic mutations
* Gene expression
* Drug sensitivity
* CRISPR knockout
* and more…

Let’s see how these datasets fit the definition of Tidy data:

| Dataframe | The observation is | Some variables are | Some values are |
| --- | --- | --- | --- |
| metadata | Cell line | ModelID, Age, OncotreeLineage | “ACH-000001”, 60, “Myeloid” |
| expression |  |  |  |
| mutation |  |  |  |

## 6.5 Transform: “What do you want to do with this dataframe”?

Remember that a major theme of the course is about: **How we organize ideas <-> Instructing a computer to do something.**

Until now, we haven’t focused too much on how we organize our scientific ideas to interact with what we can do with code. Let’s pivot to write our code driven by our scientific curiosity. After we are sure that we are working with Tidy data, we can ponder how we want to transform our data that satisfies our scientific question. We will look at several ways we can transform tidy data, starting with subsetting columns and rows.

Here’s a starting prompt:

In the metadata dataframe, which rows would you filter for and columns would you select that relate to a scientific question?

We should use the implicit subsetting mindset here: ie. “I want to filter for rows such that the Subtype is breast cancer and look at the Age and Sex.” and *not* “I want to filter for rows 20-50 and select columns 2 and 8”.

*Notice that when we filter for rows in an implicit way, we often formulate our criteria about the columns.*

(This is because we are guaranteed to have column names in dataframes, but not usually row names. Some dataframes have row names, but because the data types are not guaranteed to have the same data type across rows, it makes describing by row properties difficult.)

Let’s convert our implicit subsetting criteria into code!

metadata\_filtered = filter(metadata, OncotreeLineage == "Breast")  
breast\_metadata = select(metadata\_filtered, ModelID, Age, Sex)  
  
head(breast\_metadata)

## ModelID Age Sex  
## 1 ACH-000017 43 Female  
## 2 ACH-000019 69 Female  
## 3 ACH-000028 69 Female  
## 4 ACH-000044 47 Female  
## 5 ACH-000097 63 Female  
## 6 ACH-000111 41 Female

Here, filter() and select() are functions from the tidyverse package, which we have to install and load in via library(tidyverse) before using these functions.

### 6.5.1 Filter rows

Let’s carefully a look what how the R Console is interpreting the filter() function:

* We evaluate the expression right of =.
* The first argument of filter() is a dataframe, which we give metadata.
* The second argument is strange: the expression we give it looks like a logical indexing vector built from a comparison operator, but the variable OncotreeLineage does not exist in our environment! Rather, OncotreeLineage is a column from metadata, and we are referring to it as a **data variable** in the context of the dataframe metadata. So, we make a comparison operation on the column OncotreeLineage from metadata and its resulting logical indexing vector is the input to the second argument.
  + How do we know when a variable being used is a variable from the environment, or a data variable from a dataframe? It’s not clear cut, but here’s a rule of thumb: most functions from the tidyverse package allows you to use data variables to refer to columns of a dataframe. We refer to documentation when we are not sure.
  + This encourages more *readable* code at the expense of consistency of referring to variables in the environment. The authors of this package [describes this trade-off](https://dplyr.tidyverse.org/articles/programming.html#data--and-env-variables).
* Putting it together, filter() takes in a dataframe, and an logical indexing vector described by data variables as arguments, and returns a data frame with rows that match condition described by the logical indexing vector.
* Store this in metadata\_filtered variable.

### 6.5.2 Select columns

Let’s carefully a look what how the R Console is interpreting the select() function:

* We evaluate the expression right of =.
* The first argument of filter() is a dataframe, which we give metadata.
* The second and third arguments are **data variables** referring the columns of metadata.
  + For certain functions like filter(), there is no limit on the number of arguments you provide. You can keep adding data variables to select for more column names.
* Putting it together, select() takes in a dataframe, and as many data variables you like to select columns, and returns a dataframe with the columns you described by data variables.
* Store this in breast\_metadata variable.

## 6.6 Summary Statistics

Now that your dataframe has be transformed based on your scientific question, you can start doing some analysis on it! A common data science task is to examine summary statistics of a dataset, which summarizes the observations of a variable in a numeric summary.

If the columns of interest are numeric, then you can try functions such as mean(), median(), mode(), or summary() to get summary statistics of the column. If the columns of interest is character or logical, then you can try the table() function.

All of these functions take in a vector as input and not a dataframe, so you have to access the column as a vector via the $ operation.

mean(breast\_metadata$Age, na.rm = TRUE)

## [1] 50.96104

table(breast\_metadata$Sex)

##   
## Female Unknown   
## 91 1

## 6.7 Pipes

Often, in data analysis, we want to transform our dataframe in multiple steps via different functions. This leads to nested function calls, like this:

breast\_metadata = select(filter(metadata, OncotreeLineage == "Breast"), ModelID, Age, Sex)

This is a bit hard to read. A computer doesn’t care how difficult it is to read this line of code, but there is a lot of instructions going on in one line of code. This multi-step function composition will lead to an unreadable pattern such as:

result = function3(function2(function1(dataframe, df\_col4, df\_col2), arg2), df\_col5, arg1)

To untangle this, you have to look into the middle of this code, and slowly step out of it.

To make this more readable, programmers came up with an alternative syntax for function composition via the **pipe** metaphor. The ideas is that we push data through a chain of connected pipes, in which the output of a pipe becomes the input of the subsequent pipe.

Instead of a syntax like result2 = function3(function2(function1(dataframe))),

we linearize it with the %>% symbol: result2 = dataframe %>% function1 %>% function2 %>% function3.

In the previous example,

result = dataframe %>% function1(df\_col4, df\_col2) %>%  
 function2(arg2) %>%  
 function3(df\_col5, arg1)

This looks much easier to read. Notice that we have broken up one expression in to three lines of code for readability. If a line of code is incomplete (the first line of code is piping to somewhere unfinished), the R will treat the next line of code as part of the current line of code.

Try to rewrite the select() and filter() function composition example above using the pipe metaphor and syntax.

# 7 Data Wrangling with Tidy Data, Part 1 Exercises

If you need to clear your environment and start from the beginning, this code chunk will do that:

rm(list = ls())

## 7.1 Part 1: Warm-up/a remark on subsetting vectors

Consider the following vector:

age = c(10, 35, 24, 70, 84)

Subset age to be greater than 65 and store it as age\_over\_65.

Another use of subsetting is to modify part of a vector that satisfy a particular criteria. Instead of having the bracket [ ] notation on the right hand side of the equation, if it is on the left hand side of the equation, then we can modify a subset of the vector.

To demonstrate it, let’s create a new variable, medicaid\_eligible so that it is identical as age.

medicaid\_eligible = age

We can modify medicaid\_eligible so that all elements of medicaid\_eligible that is equal or lesss than 65 is assigned to a value of 0.

medicaid\_eligible[medicaid\_eligible <= 65] = 0  
medicaid\_eligible

## [1] 0 0 0 70 84

Now, modify medicaid\_eligible so that all elements of medicaid\_eligible that is greater than 65 is assigned to a value of 1.

Just to be aware of: Here is a new data type that we should be aware of: missing values, which are seen when there is missing data in a spreadsheet. It has the value of NA, and you can check whether a variable has NA values using the is.na() function. Try using is.na() function on the following variables:

missing\_val = NA  
vec\_with\_NA = c(2, 4, NA, NA, 3, NA)

## 7.2 Part 2: Subsetting dataframes

library(tidyverse)  
load(url("https://github.com/fhdsl/Intro\_to\_R/raw/main/classroom\_data/CCLE.RData"))

Using select(), subset expression dataframe to keep only the following columns: “KRAS\_Exp”, “NRAS\_Exp”, and “HRAS\_Exp”.

Then, using filter(), subset the result of your previous dataframe to keep rows that have “KRAS\_Exp” values between 2 and 6.

Now, formulate a new scientific question for the metadata dataset: which rows would you filter for and columns would you select that relate to a scientific question?

For instance: “I want to filter for rows such that the subtype is breast cancer, and select for columns age, sex, and the subtype.”

Your turn now:

Then, use filter() and select() to subset your dataframe so that it address your scientific question.

## 7.3 Part 3: Summary statistics

Let’s take a short break from looking at cancer cell lines for the moment. Let’s consider a sample of penguins.

Let’s load it in, and take a look.

library(tidyverse)  
library(palmerpenguins)  
head(penguins)

## # A tibble: 6 × 8  
## species island bill\_length\_mm bill\_depth\_mm flipper\_length\_mm body\_mass\_g  
## <fct> <fct> <dbl> <dbl> <int> <int>  
## 1 Adelie Torgersen 39.1 18.7 181 3750  
## 2 Adelie Torgersen 39.5 17.4 186 3800  
## 3 Adelie Torgersen 40.3 18 195 3250  
## 4 Adelie Torgersen NA NA NA NA  
## 5 Adelie Torgersen 36.7 19.3 193 3450  
## 6 Adelie Torgersen 39.3 20.6 190 3650  
## # ℹ 2 more variables: sex <fct>, year <int>

Let’s examine some summary statistics of this dataset: **pick a few columns of interest, and summarize the column via the following functions**:

If the columns of interest are numeric, then you can try functions such as mean(), median(), min(), or max().

If the columns of interest is character or logical, then you can try the table() function.

Within the column, if there are NA (a special value indicating “Not Available”) elements present, some of these functions will return with a value of NA. To remove the NA value before computing, some of these function have a na.rm argument that you can set to TRUE.

All of these functions take in a vector as input and not a dataframe, so you have to access the column as a vector via the $ operation.

mean(penguins$body\_mass\_g, na.rm=TRUE) #example

## [1] 4201.754

#compute the summary statistics of a few columns below:

Then, **compare the mean body mass between the three penguin species** (Adelie, Chinstrap, Gentoo).

In order to do so, you have to subset the penguins dataframe three times, so that each dataframe contains one of the three species (the code below subsets the first species for you). Then, you will analyze each species’ dataframe separately using the mean() function. Do you see any difference of mean penguin mass across species?

adelie = filter(penguins, species == "Adelie")  
#your code here

## 7.4 Feedback!

How many hours did you spend on this exercise?

time\_spent = 0

If you worked with other peers, write their names down in the following character vector: each element is one person’s name.

peers = c("myself")

# 8 Data Wrangling with Tidy Data, Part 2

Today, we will continue learning about common functions from the Tidyverse that is useful for Tidy data manipulations.

## 8.1 Modifying and creating new columns in dataframes

The mutate() function takes in the following arguments: the first argument is the dataframe of interest, and the second argument is a *new or existing data variable* that is defined in terms of *other data variables*.

We create a new column olderAge that is 10 years older than the original Age column.

metadata$Age[1:10]

## [1] 60 36 72 30 30 64 63 56 72 53

metadata2 = mutate(metadata, olderAge = Age + 10)  
metadata2$olderAge[1:10]

## [1] 70 46 82 40 40 74 73 66 82 63

Here, we used an operation on a column of metadata. Here’s another example with a function:

expression$KRAS\_Exp[1:10]

## [1] 4.634012 4.638653 4.032101 5.503031 3.713696 3.972693 3.235727 4.135042  
## [9] 9.017365 3.940167

expression2 = mutate(expression, log\_KRAS\_Exp = log(KRAS\_Exp))  
expression2$log\_KRAS\_Exp[1:10]

## [1] 1.533423 1.534424 1.394288 1.705299 1.312028 1.379444 1.174254 1.419498  
## [9] 2.199152 1.371223

### 8.1.1 Alternative: Creating and modifying columns via $

Instead of mutate() function, we can also create a new or modify a column via the $ symbol:

expression2 = expression  
expression2$log\_KRAS\_Exp = log(expression2$KRAS\_Exp)

## 8.2 Merging two dataframes together

Suppose we have the following dataframes:

expression

| ModelID | PIK3CA\_Exp | log\_PIK3CA\_Exp |
| --- | --- | --- |
| “ACH-001113” | 5.138733 | 1.636806 |
| “ACH-001289” | 3.184280 | 1.158226 |
| “ACH-001339” | 3.165108 | 1.152187 |

metadata

| ModelID | OncotreeLineage | Age |
| --- | --- | --- |
| “ACH-001113” | “Lung” | 69 |
| “ACH-001289” | “CNS/Brain” | NA |
| “ACH-001339” | “Skin” | 14 |

Suppose that I want to compare the relationship between OncotreeLineage and PIK3CA\_Exp, but they are columns in different dataframes. We want a new dataframe that looks like this:

| ModelID | PIK3CA\_Exp | log\_PIK3CA\_Exp | OncotreeLineage | Age |
| --- | --- | --- | --- | --- |
| “ACH-001113” | 5.138733 | 1.636806 | “Lung” | 69 |
| “ACH-001289” | 3.184280 | 1.158226 | “CNS/Brain” | NA |
| “ACH-001339” | 3.165108 | 1.152187 | “Skin” | 14 |

We see that in both dataframes, the rows (observations) represent cell lines with a common column ModelID, so let’s merge these two dataframes together, using full\_join():

merged = full\_join(metadata, expression, by = "ModelID")

The number of rows and columns of metadata:

dim(metadata)

## [1] 1864 30

The number of rows and columns of expression:

dim(expression)

## [1] 1450 536

The number of rows and columns of merged:

dim(merged)

## [1] 1864 565

We see that the number of *columns* in merged combines the number of columns in metadata and expression, while the number of *rows* in merged is the larger of the number of rows in metadata and expression : full\_join() keeps all observations common to both dataframes based on the common column defined via the by argument.

Therefore, we expect to see NA values in merged, as there are some cell lines that are not in expression dataframe.

There are variations of this function depending on your application:



Given xxx\_join(x, y, by = "common\_col"),

* full\_join() keeps all observations.
* left\_join() keeps all observations in x.
* right\_join() keeps all observations in y.
* inner\_join() keeps observations common to both x and y.

## 8.3 Grouping and summarizing dataframes

Also known as: “The rows I want is described by a column. The columns I want need to be summarized from other columns.”

In a dataset, there may be multiple levels of observations, and which level of observation we examine depends on our scientific question. For instance, in metadata, the observation is cell lines. However, perhaps we want to understand properties of metadata in which the observation is the cancer type, OncotreeLineage. Suppose we want the mean age of each cancer type, and the number of cell lines that we have for each cancer type.

This is a scenario in which the *desired rows are described by a column*, OncotreeLineage, and the columns, such as mean age, need to be *summarized from other columns.*

As an example, this dataframe is transformed from:

| ModelID | OncotreeLineage | Age |
| --- | --- | --- |
| “ACH-001113” | “Lung” | 69 |
| “ACH-001289” | “Lung” | 23 |
| “ACH-001339” | “Skin” | 14 |
| “ACH-002342” | “Brain” | 23 |
| “ACH-004854” | “Brain” | 56 |
| “ACH-002921” | “Brain” | 67 |

into:

| OncotreeLineage | MeanAge | Count |
| --- | --- | --- |
| “Lung” | 46 | 2 |
| “Skin” | 14 | 1 |
| “Brain” | 48.67 | 3 |

We use the functions group\_by() and summarise() :

metadata\_by\_type = metadata %>%   
 group\_by(OncotreeLineage) %>%   
 summarise(MeanAge = mean(Age, rm.na=TRUE), Count = n())

## `summarise()` ungrouping output (override with `.groups` argument)

Or, without pipes:

metadata\_by\_type\_temp = group\_by(metadata, OncotreeLineage)  
metadata\_by\_type = summarise(metadata\_by\_type\_temp, MeanAge = mean(Age, rm.na=TRUE), Count = n())

## `summarise()` ungrouping output (override with `.groups` argument)

The group\_by() function returns the identical input dataframe but remembers which variable(s) have been marked as grouped:

head(group\_by(metadata, OncotreeLineage))

## # A tibble: 6 × 30  
## # Groups: OncotreeLineage [3]  
## ModelID PatientID CellLineName StrippedCellLineName Age SourceType  
## <chr> <chr> <chr> <chr> <dbl> <chr>   
## 1 ACH-000001 PT-gj46wT NIH:OVCAR-3 NIHOVCAR3 60 Commercial  
## 2 ACH-000002 PT-5qa3uk HL-60 HL60 36 Commercial  
## 3 ACH-000003 PT-puKIyc CACO2 CACO2 72 Commercial  
## 4 ACH-000004 PT-q4K2cp HEL HEL 30 Commercial  
## 5 ACH-000005 PT-q4K2cp HEL 92.1.7 HEL9217 30 Commercial  
## 6 ACH-000006 PT-ej13Dz MONO-MAC-6 MONOMAC6 64 Commercial  
## # ℹ 24 more variables: SangerModelID <chr>, RRID <chr>, DepmapModelType <chr>,  
## # AgeCategory <chr>, GrowthPattern <chr>, LegacyMolecularSubtype <chr>,  
## # PrimaryOrMetastasis <chr>, SampleCollectionSite <chr>, Sex <chr>,  
## # SourceDetail <chr>, LegacySubSubtype <chr>, CatalogNumber <chr>,  
## # CCLEName <chr>, COSMICID <dbl>, PublicComments <chr>,  
## # WTSIMasterCellID <dbl>, EngineeredModel <chr>, TreatmentStatus <chr>,  
## # OnboardedMedia <chr>, PlateCoating <chr>, OncotreeCode <chr>, …

The summarise() returns one row for each combination of grouping variables, and one column for each of the summary statistics that you have specified.

Functions you can use for summarise() must take in a vector and return a simple data type, such as any of our summary statistics functions: mean(), median(), min(), max(), etc.

The exception is n(), which returns the number of entries for each grouping variable’s value.

You can combine group\_by() with other functions. See this [guide](https://dplyr.tidyverse.org/articles/grouping.html).

## 8.4 Appendix: How functions are built

As you become more independent R programmers, you will spend time learning about new functions on your own. We have gone over the basic anatomy of a function call back in the first lesson, but now let’s go a bit deeper to understand how a function is built and how to call them.

Recall that a function has a **function name**, **input arguments**, and a **return value**.

*Function definition consists of assigning a* ***function name*** *with a “function” statement that has a comma-separated list of named* ***function arguments****, and a* ***return expression****. The function name is stored as a variable in the global environment.*

In order to use the function, one defines or import it, then one calls it.

Example:

addFunction = function(num1, num2) {  
 result = num1 + num2   
 return(result)  
}  
result = addFunction(3, 4)

With function definitions, not all code runs from top to bottom. The first four lines defines the function, but the function is never run. It is called on line 5, and the lines within the function are executed.

When the function is called in line 5, the variables for the arguments are reassigned to function arguments to be used within the function and helps with the modular form.

To see why we need the variables of the arguments to be reassigned, consider the following function that is *not* modular:

x = 3  
y = 4  
addFunction = function(num1, num2) {  
 result = x + y   
 return(result)  
}  
result = addFunction(10, -10)

Some syntax equivalents on calling the function:

addFunction(3, 4)  
addFunction(num1 = 3, num2 = 4)  
addFunction(num2 = 4, num1 = 3)

but this *could* be different:

addFunction(4, 3)

With a deeper knowledge of how functions are built, when you encounter a foreign function, you can look up its help page to understand how to use it. For example, let’s look at mean():

?mean  
  
Arithmetic Mean  
  
Description:  
  
 Generic function for the (trimmed) arithmetic mean.  
  
Usage:  
  
 mean(x, ...)  
   
 ## Default S3 method:  
 mean(x, trim = 0, na.rm = FALSE, ...)  
   
Arguments:  
  
 x: An R object. Currently there are methods for numeric/logical  
 vectors and date, date-time and time interval objects.  
 Complex vectors are allowed for ‘trim = 0’, only.  
  
 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.  
  
 na.rm: a logical evaluating to ‘TRUE’ or ‘FALSE’ indicating whether  
 ‘NA’ values should be stripped before the computation  
 proceeds.  
  
 ...: further arguments passed to or from other methods.

Notice that the arguments trim = 0, na.rm = FALSE have default values. This means that these arguments are *optional* - you should provide it only if you want to. With this understanding, you can use mean() in a new way:

numbers = c(1, 2, NA, 4)  
mean(x = numbers, na.rm = TRUE)

## [1] 2.333333

# 9 Data Wrangling with Tidy Data, Part 2 Exercises

To erase your environment and start from scratch:

rm(list = ls())

library(tidyverse)

## Warning: package 'tidyverse' was built under R version 4.0.3

## Warning: package 'purrr' was built under R version 4.0.5

## Warning: package 'stringr' was built under R version 4.0.3

library(palmerpenguins)  
load(url("https://github.com/fhdsl/Intro\_to\_R/raw/main/classroom\_data/CCLE.RData"))

## 9.1 Part 1: Creating new columns

Consider the following dataframe:

simple\_df = data.frame(id = c("AAA", "BBB", "CCC", "DDD", "EEE"),  
 case\_control = c("case", "case", "control", "control", "control"),  
 measurement1 = c(2.5, 3.5, 9, .1, 2.2),  
 measurement2 = c(0, 0, .5, .24, .003),  
 measurement3 = c(80, 2, 1, 1, 2))  
  
simple\_df

## id case\_control measurement1 measurement2 measurement3  
## 1 AAA case 2.5 0.000 80  
## 2 BBB case 3.5 0.000 2  
## 3 CCC control 9.0 0.500 1  
## 4 DDD control 0.1 0.240 1  
## 5 EEE control 2.2 0.003 2

Create a new column, named sum\_measurements, via mutate or $, which is the sum of measurement1, measurement2, and measurement3 columns.

Filter simple\_df to samples that have “case” in case\_control column, and look at the **mean** of the sum\_measurements column. Do the same for samples that have the “control” in case\_control column. How do they compare?

Let’s return back to the metadata dataframe. Here is a custom function medicaid\_eligible() that takes a numeric vector age and makes some transformations: it returns 0 if age is less than 65 and 1 if age is equal or greater than 65. You don’t need to understand how the function is defined, but you should understand the example usages of the medicaid\_eligible() function on a numeric vector.

medicaid\_eligible = function(age) {  
 age[age < 65] = 0  
 age[age >= 65] = 1  
 return(age)  
}  
  
#example usage:  
medicaid\_eligible(c(24, 95))

## [1] 0 1

#another example:  
my\_family\_ages = c(3, 64, 32, 35, 76)  
medicaid\_eligible(my\_family\_ages)

## [1] 0 0 0 0 1

Create a new column named medicaid that is the output of the medicaid\_eligible function with input argument of the Age column.

## 9.2 Part 2: Joining multiple dataframes

Consider another dataframe related to simple\_df, called simple2\_df.

simple2\_df = data.frame(id = c("AAA", "EEE", "FFF", "GGG", "HHH"),  
 measurement4 = c(25, 23, 56, 23, 9),  
 measurement5 = c(TRUE, TRUE, TRUE, FALSE, FALSE))

We want to join simple\_df and simple2\_df together using a common identifier column id. The function full\_join() will keeps all observations common to both dataframes based on the common column defined via the by argument via a character.

dim(simple\_df)

## [1] 5 5

dim(simple2\_df)

## [1] 5 3

simple\_joined\_df = full\_join(simple\_df, simple2\_df, by = "id")  
dim(simple\_joined\_df)

## [1] 8 7

simple\_joined\_df

## id case\_control measurement1 measurement2 measurement3 measurement4  
## 1 AAA case 2.5 0.000 80 25  
## 2 BBB case 3.5 0.000 2 NA  
## 3 CCC control 9.0 0.500 1 NA  
## 4 DDD control 0.1 0.240 1 NA  
## 5 EEE control 2.2 0.003 2 23  
## 6 FFF <NA> NA NA NA 56  
## 7 GGG <NA> NA NA NA 23  
## 8 HHH <NA> NA NA NA 9  
## measurement5  
## 1 TRUE  
## 2 NA  
## 3 NA  
## 4 NA  
## 5 TRUE  
## 6 TRUE  
## 7 FALSE  
## 8 FALSE

Notice that it is by = "id", not by = id when referring to the id column common to both dataframes. In my opinion, this function’s design was not great, as it runs against the form we are used to.

We see NA in some of our data, because not every entry in simple\_df is in simple2\_df, and vice versa.

There are other variations of full\_join():

* full\_join(x, y, by = "common\_col") keeps all observations.
* left\_join(x, y, by = "common\_col") keeps all observations in x.
* right\_join(x, y, by = "common\_col") keeps all observations in y.
* inner\_join(x, y, by = "common\_col") keeps observations common to both x and y.

Try some of the other join functions out yourself:

Now, let’s apply what we learned here to our genomics dataframes.

Using full\_join twice, create a dataframe that has columns from metadata, expression, and mutation. You should use column ModelID that is common to all three dataframes as your identifier. How will you check your work to know that you did the join correctly?

## 9.3 Part 3: Pipes

Often, in data analysis, we want to transform our dataframe in multiple steps via different functions. This leads to nested function calls, like this:

breast\_metadata = select(filter(metadata, OncotreeLineage == "Breast"), ModelID, Age, Sex)

This is a bit hard to read. A computer doesn’t care how difficult it is to read this line of code, but there is a lot of instructions going on in one line of code. This multi-step function composition will lead to an unreadable pattern such as:

result = function3(function2(function1(dataframe)))

To untangle this, you have to look into the middle of this code, and slowly step out of it.

To make this more readable, programmers came up with an alternative syntax for function composition via the **pipe** metaphor. The ideas is that we push data through a chain of connected pipes, in which the output of a pipe becomes the input of the subsequent pipe.

Instead of a syntax like result = function3(function2(function1(dataframe))),

we linearize it with the %>% symbol: result2 = dataframe %>% function1 %>% function2 %>% function3.

In the previous example,

breast\_metadata = metadata %>%   
 filter(OncotreeLineage == "Breast") %>%  
 select(ModelID, Age, Sex)

This looks much easier to read. Notice that we have broken up one expression in to three lines of code for readability. If a line of code is incomplete (the first line of code is piping to somewhere unfinished), the R will treat the next line of code as part of the current line of code.

Your turn: write the following the code in the pipe form and check that it gets the same result:

analysis = select(filter(metadata, Age > 18 & Age < 45), Age, OncotreeLineage, PrimaryOrMetastasis)

Challenge!!! Make this 3-nested function call into the pipe format.

analysis = mutate(select(filter(metadata, Age > 18 & Age < 45), Age, OncotreeLineage, PrimaryOrMetastasis), medicaid = medicaid\_eligible(Age))

In the world of R data analysis, you will see some people analyzing their data using the %>% symbol, and some will just use nested functions. You should decide what you prefer, but just want you to be aware of people’s styles.

## 9.4 Feedback!

How many hours did you spend on this exercise?

time\_spent = 0

If you worked with other peers, write their names down in the following character vector: each element is one person’s name.

peers = c("myself")

# 10 Data Visualization

## 10.1 Common Plots

### 10.1.1 Univariate

* Numeric: histogram
* Character: bar plots

### 10.1.2 Bivariate

* Numeric vs. Numeric: Scatterplot, line plot
* Numeric vs. Character: Box plot

Why do we focus on these common plots? Our eyes are better at distinguishing certain visual features more than others. All of these plots are focused on their position to depict data, which gives us the most effective visual scale.



Figure : Source: <https://www.oreilly.com/library/view/visualization-analysis-and/9781466508910/K14708_C005.xhtml>

## 10.2 Grammar of Graphics

The syntax of the grammar of graphics breaks down into 4 sections.

Data

Mapping to data

Geometry

Additional settings

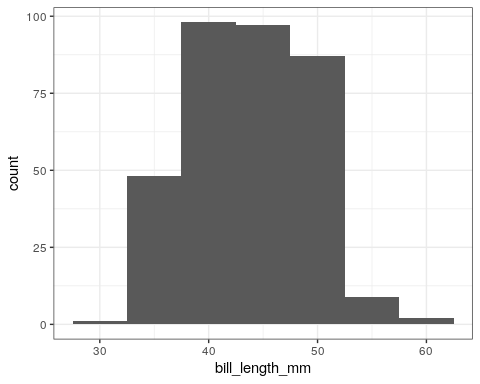
### 10.2.1 Histogram

ggplot(penguins) + aes(x = bill\_length\_mm) + geom\_histogram() + theme\_bw()



With options:

ggplot(penguins) + aes(x = bill\_length\_mm) + geom\_histogram(binwidth = 5) + theme\_bw()



### 10.2.2 Bar plots

ggplot(penguins) + aes(x = species) + geom\_bar()



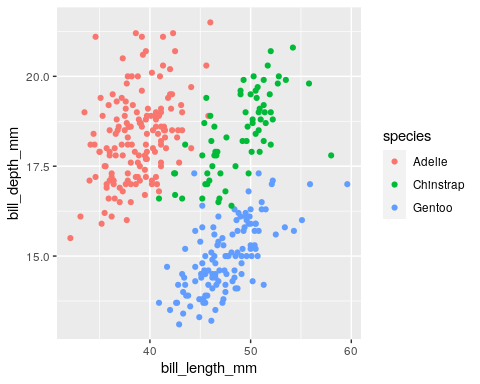
### 10.2.3 Scatterplot

ggplot(penguins) + aes(x = bill\_length\_mm, y = bill\_depth\_mm) + geom\_point()



### 10.2.4 Multivaraite Scatterplot

ggplot(penguins) + aes(x = bill\_length\_mm, y = bill\_depth\_mm, color = species) + geom\_point()



### 10.2.5 Multivaraite Scatterplot

ggplot(penguins) + aes(x = bill\_length\_mm, y = bill\_depth\_mm) + geom\_point() + facet\_wrap(~species)



### 10.2.6 Line plot?

ggplot(penguins) + aes(x = bill\_length\_mm, y = bill\_depth\_mm) + geom\_line()



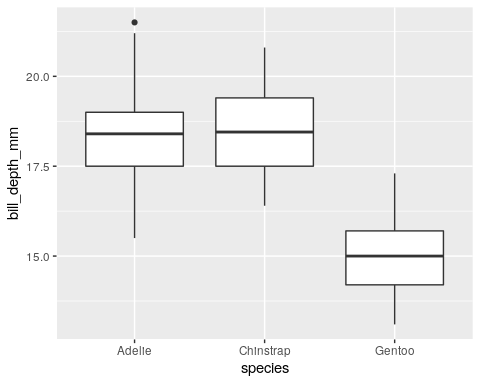
### 10.2.7 Grouped Line plot?

ggplot(penguins) + aes(x = bill\_length\_mm, y = bill\_depth\_mm, group = species) + geom\_line()



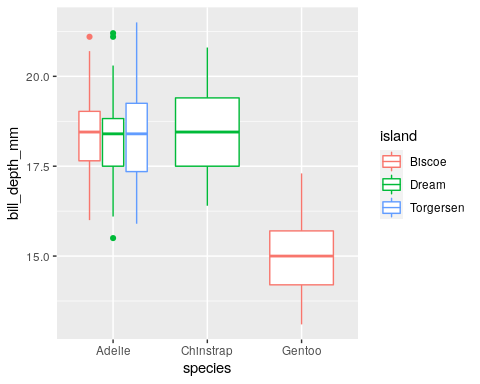
### 10.2.8 Boxplot

ggplot(penguins) + aes(x = species, y = bill\_depth\_mm) + geom\_boxplot()



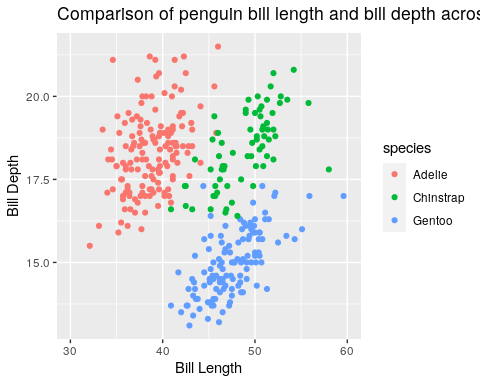
### 10.2.9 Grouped Boxplot

ggplot(penguins) + aes(x = species, y = bill\_depth\_mm, color = island) + geom\_boxplot()



### 10.2.10 Some additional options

ggplot(data = penguins) + aes(x = bill\_length\_mm, y = bill\_depth\_mm, color = species) + geom\_point() + labs(x = “Bill Length”, y = “Bill Depth”, title = “Comparison of penguin bill length and bill depth across species”) + scale\_x\_continuous(limits = c(30, 60))



## 10.3 Summary of options

data

geom\_point: x, y, color, shape

geom\_line: x, y, group, color

geom\_histogram: x, y, fill

geom\_bar: x, fill

geom\_boxplot: x, y, fill, color

facet\_wrap

labs

scale\_x\_continuous

scale\_y\_continuous

scale\_x\_discrete

scale\_y\_discrete

Consider the esquisse package to help generate your ggplot code via drag and drop.

# About the Authors

These credits are based on our [course contributors table guidelines](https://www.ottrproject.org/more_features.html#giving-credits-to-contributors).

| Credits | Names |
| --- | --- |
| **Pedagogy** |  |
| Lead Content Instructor(s) | Chris Lo |
| Lecturer | Chris Lo |
| Content Author(s) (include chapter name/link in parentheses if only for specific chapters) - make new line if more than one chapter involved | If any other authors besides lead instructor |
| Content Contributor(s) (include section name/link in parentheses) - make new line if more than one section involved | Wrote less than a chapter |
| Content Editor(s)/Reviewer(s) | Checked your content |
| Content Director(s) | Helped guide the content direction |
| Content Consultants (include chapter name/link in parentheses or word “General”) - make new line if more than one chapter involved | Gave high level advice on content |
| Acknowledgments | Gave small assistance to content but not to the level of consulting |
| **Production** |  |
| Content Publisher(s) | Helped with publishing platform |
| Content Publishing Reviewer(s) | Reviewed overall content and aesthetics on publishing platform |
| **Technical** |  |
| Course Publishing Engineer(s) | Helped with the code for the technical aspects related to the specific course generation |
| Template Publishing Engineers | [Candace Savonen](https://www.cansavvy.com/), [Carrie Wright](https://carriewright11.github.io/), [Ava Hoffman](https://www.avahoffman.com/) |
| Publishing Maintenance Engineer | [Candace Savonen](https://www.cansavvy.com/) |
| Technical Publishing Stylists | [Carrie Wright](https://carriewright11.github.io/), [Ava Hoffman](https://www.avahoffman.com/), [Candace Savonen](https://www.cansavvy.com/) |
| Package Developers ([ottrpal](https://github.com/jhudsl/ottrpal)) [Candace Savonen](https://www.cansavvy.com/), [John Muschelli](https://johnmuschelli.com/), [Carrie Wright](https://carriewright11.github.io/) |  |
| **Art and Design** |  |
| Illustrator(s) | Created graphics for the course |
| Figure Artist(s) | Created figures/plots for course |
| Videographer(s) | Filmed videos |
| Videography Editor(s) | Edited film |
| Audiographer(s) | Recorded audio |
| Audiography Editor(s) | Edited audio recordings |
| **Funding** |  |
| Funder(s) | Institution/individual who funded course including grant number |
| Funding Staff | Staff members who help with funding |

## ─ Session info ───────────────────────────────────────────────────────────────  
## setting value   
## version R version 4.0.2 (2020-06-22)  
## os Ubuntu 20.04.5 LTS   
## system x86\_64, linux-gnu   
## ui X11   
## language (EN)   
## collate en\_US.UTF-8   
## ctype en\_US.UTF-8   
## tz Etc/UTC   
## date 2024-03-19   
##   
## ─ Packages ───────────────────────────────────────────────────────────────────  
## package \* version date lib source   
## assertthat 0.2.1 2019-03-21 [1] RSPM (R 4.0.5)   
## bookdown 0.24 2024-03-13 [1] Github (rstudio/bookdown@88bc4ea)   
## cachem 1.0.8 2023-05-01 [1] CRAN (R 4.0.2)   
## callr 3.5.0 2020-10-08 [1] RSPM (R 4.0.2)   
## cli 3.6.2 2023-12-11 [1] CRAN (R 4.0.2)   
## crayon 1.3.4 2017-09-16 [1] RSPM (R 4.0.0)   
## desc 1.2.0 2018-05-01 [1] RSPM (R 4.0.3)   
## devtools 2.3.2 2020-09-18 [1] RSPM (R 4.0.3)   
## digest 0.6.25 2020-02-23 [1] RSPM (R 4.0.0)   
## ellipsis 0.3.1 2020-05-15 [1] RSPM (R 4.0.3)   
## evaluate 0.23 2023-11-01 [1] CRAN (R 4.0.2)   
## fastmap 1.1.1 2023-02-24 [1] CRAN (R 4.0.2)   
## fs 1.5.0 2020-07-31 [1] RSPM (R 4.0.3)   
## glue 1.4.2 2020-08-27 [1] RSPM (R 4.0.5)   
## htmltools 0.5.7 2023-11-03 [1] CRAN (R 4.0.2)   
## knitr 1.33 2024-03-13 [1] Github (yihui/knitr@a1052d1)   
## magrittr 2.0.3 2022-03-30 [1] CRAN (R 4.0.2)   
## memoise 2.0.1 2021-11-26 [1] CRAN (R 4.0.2)   
## pkgbuild 1.1.0 2020-07-13 [1] RSPM (R 4.0.2)   
## pkgload 1.1.0 2020-05-29 [1] RSPM (R 4.0.3)   
## prettyunits 1.1.1 2020-01-24 [1] RSPM (R 4.0.3)   
## processx 3.4.4 2020-09-03 [1] RSPM (R 4.0.2)   
## ps 1.4.0 2020-10-07 [1] RSPM (R 4.0.2)   
## R6 2.4.1 2019-11-12 [1] RSPM (R 4.0.0)   
## remotes 2.2.0 2020-07-21 [1] RSPM (R 4.0.3)   
## rlang 1.1.3 2024-01-10 [1] CRAN (R 4.0.2)   
## rmarkdown 2.10 2024-03-13 [1] Github (rstudio/rmarkdown@02d3c25)  
## rprojroot 2.0.4 2023-11-05 [1] CRAN (R 4.0.2)   
## sessioninfo 1.1.1 2018-11-05 [1] RSPM (R 4.0.3)   
## stringi 1.5.3 2020-09-09 [1] RSPM (R 4.0.3)   
## stringr 1.4.0 2019-02-10 [1] RSPM (R 4.0.3)   
## testthat 3.0.1 2024-03-13 [1] Github (R-lib/testthat@e99155a)   
## usethis 1.6.3 2020-09-17 [1] RSPM (R 4.0.2)   
## withr 2.3.0 2020-09-22 [1] RSPM (R 4.0.2)   
## xfun 0.26 2024-03-13 [1] Github (yihui/xfun@74c2a66)   
## yaml 2.2.1 2020-02-01 [1] RSPM (R 4.0.3)   
##   
## [1] /usr/local/lib/R/site-library  
## [2] /usr/local/lib/R/library

# 11 References