PBA R Basics
Reference Slides

Fall 2026. Reference material: consult as needed alongside the weekly readings.

Jaewon (“Jay-one”) Yoo
National Tsing Hua University

What is in this file

1. **Getting R bearings**: operators, precedence, variables and names
2. **Data types**: numeric, character, factor, logical, date/time
3. **Vectors**: generate, compute, index, filter, convert
4. **From vectors to data frames**: matrix-style and list-style indexing, building and importing a data frame
5. **User-defined functions**: writing, defaults, nesting, scope

These slides are reference material for PBA. They are not lectured; the tools appear when an analysis needs them. Suggested readings: AAG Ch. 4–6 (types, structures, reading data), Ch. 7–9 (functions and control flow).

1/ Getting R Bearings

Getting R bearings: the operators you will type all semester.

Operators in R: Arithmetic Operators

- `Every journey has to start somewhere...`
- These operators are used to do basic arithmetic operations like addition, subtraction, multiplication, division, exponent, modulus, etc.

Operator	Operation	Examples
*	multiplication	a*b
/	division	a/b
+	addition	a+b
-	subtraction	a-b
^	exponentiation	a^b
%%	modulus	a%%b
%/%	integer division	a%/%b

Operators in R: Relational Operators

- These operators are used to compare (e.g., equals, larger than, smaller than, etc.) two objects.

Operator	Operation	Examples
==	equals	a==b
!=	not equal	a!=b
>	greater than	a>b
>=	greater than or equal to	a>=b
<	less than	a<b
<=	less than or equal to	a<=b

Operators in R: Logical Operators

- These operators are used to perform logical/Boolean operations like AND, OR, NOT, etc. on objects.

Operator	Operation	Description
!	not	negation operator, e.g., !a
|	or	elementwise **or** operator, e.g., a|b
||	or	**or** operator which evaluates only the first element in the objects, e.g., a||b
&	and	elementwise and operator, e.g., a&b
&&	and	**and** operator which evaluates only the first element in the objects, e.g., a&&b

Operators in R: Assignment Operators

- The following operators are used to assign/store values (or data) on to a variable/object.

Operator	Operation	Examples
->	Rightward assignment	a -> b
->>	Rightward super-assignment	a ->> b
=	Assignment (right to left)	a = b
<-	Assignment (right to left)	a <- b
<<-	Super-assignment (right to left)	a <<- b

  - Rightward assignment is possible, but it's most common to assign right to left using <- or =
  - Super-assignment operators to be discussed when we learn the scoping rules.

Operators in R: Indexing Operators

- The following operators can be used to select and return specific values in a variable/object.

Operator	Operation	Examples
`[ ]`	Indexing which returns elements of list class	`a[b]`
`[[ ]]`	Indexing which returns an object of the class contained in the list	`a[[b]]`
`$`	Indexing single elements by name	`a$b`

Operator Precedence

- `1+2*3`

  ```
  [1] 7
  ```

  # Following the precedence rules, the program runs the multiplication first then moves onto addition.

- `(1+2)*3`

  ```
  [1] 9
  ```

  # To have your program run the addition first, you enclose it with parenthesis.

Variables in R

- A variable is a placeholder in computer memory used to store data.
  - Unlike other languages (e.g., C, C++), R does not require variables to be declared for use. Any form of values can be assigned to a variable in R.
- When the variable appear elsewhere in the program, the latest value stored in that variable is used.
- Variable assignment in R:
  - `x <- 2 ## If we printed x now, R will print 2.`
  - `y = 5`
  - `x <- y ## What would R print out if we print(x) now?`
  - `3 -> z`
  - `a <- b <- 7`

Naming Variables

- Proper way to name a variable:
  - Any combination of alphanumeric characters (i.e., letters and numerals) along with periods (.) and underscores (_).
  - Cannot start with a number or an underscore.
  - Case sensitive (e.g., FOO, Foo, and foo are three different variables).

Good names	Names that cause errors
a	1trial
b	$
FOO	^mean
my_var	2nd
.day	!bad

- ⇝ Use variable names that clearly present the data stored in it (e.g., purchase.freq, price, quantity).

2/ Data Types

Numeric and Integers in R

- **Two Main Types**
  - Numeric (with decimal; also known as double)
  - Integer (whole number)
- **Default Behavior**
  - Numeric values stored in a variable are automatically assumed to be of type numeric.
  - Append the value with an 'L' to specify as integer (e.g., 5L).
- **Examples:**

```
> i <- 5L
> is.integer(i)
> class(i)  # or use typeof() or mode()
> class(4L * 2.8)
> class(5L / 2L)
```

Characters/Strings in R

- **What is it?**
  - Sequences of texts enclosed in single (') or double quotes (").
  - Case-sensitive: "Data" ≠ "data" ≠ "DATA".
- **Manipulating Characters**
  - Concatenation: `paste()`, `paste0()`.
  - Substring: `substr()`.
  - String length: `nchar()`.
- **Examples:**

```
> x <- "data"
> class(x)
> nchar(x)
> paste("Hello", x)
> substr(x, 1, 2)
```

Factors in R

- **Factor Fundamentals**
  - Ideal for categorical data (e.g., gender).
  - Internally stored as integers, displayed as character labels.
- **Why Use Factors?**
  - Ensure correct treatment of categorical variables in statistical models.
  - Efficient storage and data manipulation.
- **Common Functions**
  - Get or set factor levels: `levels()`.
  - Convert to factor: `as.factor()`.
  - Create a factor: `factor()`.
- **Examples:**

```
> gender <- factor(c("Male", "Female", "Male"))
> levels(gender)
> as.integer(gender)
> ordered_gender <- factor(gender, ordered = TRUE)
```

Logical/Booleans in R

- **Logicals (Booleans)**: Derived from Boolean algebra by George Boole, represent data as TRUE or FALSE.
  - Result from comparisons of numbers or characters.
  - Numerically, non-zero is TRUE, zero is FALSE.

- **Examples**:

```
> TRUE * 5     # Output: 5
> FALSE * 5    # Output: 0
> 2 == 3       # Output: FALSE
> 2 != 3       # Output: TRUE
> 2 < 3        # Output: TRUE
> 2 >= 3       # Output: FALSE
> "data" == "stats" # Output: FALSE
> "data" < "stats"  # Output: TRUE
```

More on Logical Operations

- & (AND, Conjunction $\wedge$): TRUE if both are TRUE.
- | (OR, Disjunction $\vee$): TRUE if at least one is TRUE.
- ! (NOT, Negation $\neg$): Reverses the logical value.

- **Examples:**

```
> 3 & 0  ## FALSE
> 3 | 0  ## TRUE
> !0     ## TRUE
> 3 & 1  ## TRUE
> 3 | 2  ## TRUE
> !1     ## FALSE
```

Logical Operations in Practice

- Customer Segmentation
  - Logical operations can be used to define business rules.
  - Example: classifying customers as `VIP` or `AtRisk`.
- **Examples:**

```
> customers <- data.frame(
    id = 1:6,
    spend = c(500, 1200, 300, 750, 2000, 150),
    visits = c(12, 4, 30, 7, 20, 1)
  )

# Define "VIP" customers: high spend AND frequent visits
> customers$VIP <- (customers$spend > 1000) &
    (customers$visits > 10)

# Define "at-risk": low spend OR very few visits
> customers$AtRisk <- (customers$spend < 400) |
    (customers$visits < 5)

> customers
  id spend visits   VIP AtRisk
1  1   500     12 FALSE  FALSE
2  2  1200      4 FALSE   TRUE
3  3   300     30 FALSE   TRUE
4  4   750      7 FALSE  FALSE
5  5  2000     20  TRUE  FALSE
6  6   150      1 FALSE   TRUE
```

Another Practical Use Case

- Data Cleaning
  - Logical checks are powerful for identifying invalid or missing data.
  - Example: flagging rows with impossible ages or missing income.
- **Examples:**

```
require(tidyverse)
> df <- tibble(
    age = c(25,-3,45,200,30),
    income = c(50000,20000,NA,100000,30000)
  )

> invalid <- (df$age < 0) | (df$age > 120) | is.na(df$income); invalid
[1] FALSE  TRUE  TRUE  TRUE FALSE

> df[invalid, ]

# A tibble: 3 × 2
    age income
  <dbl>  <dbl>
1    -3  20000
2    45     NA
3   200 100000
```

Dates and Time in R

- R manages dates with **Date**, and date-time with **POSIXct** or **POSIXlt**.
  - Stored as days (Date) or seconds (POSIXct) since 1970-01-01 (Epoch).
  - `POSIXlt` is stored as a list of components (year, month, day, hour, minute, second, etc.).

- **Examples:**

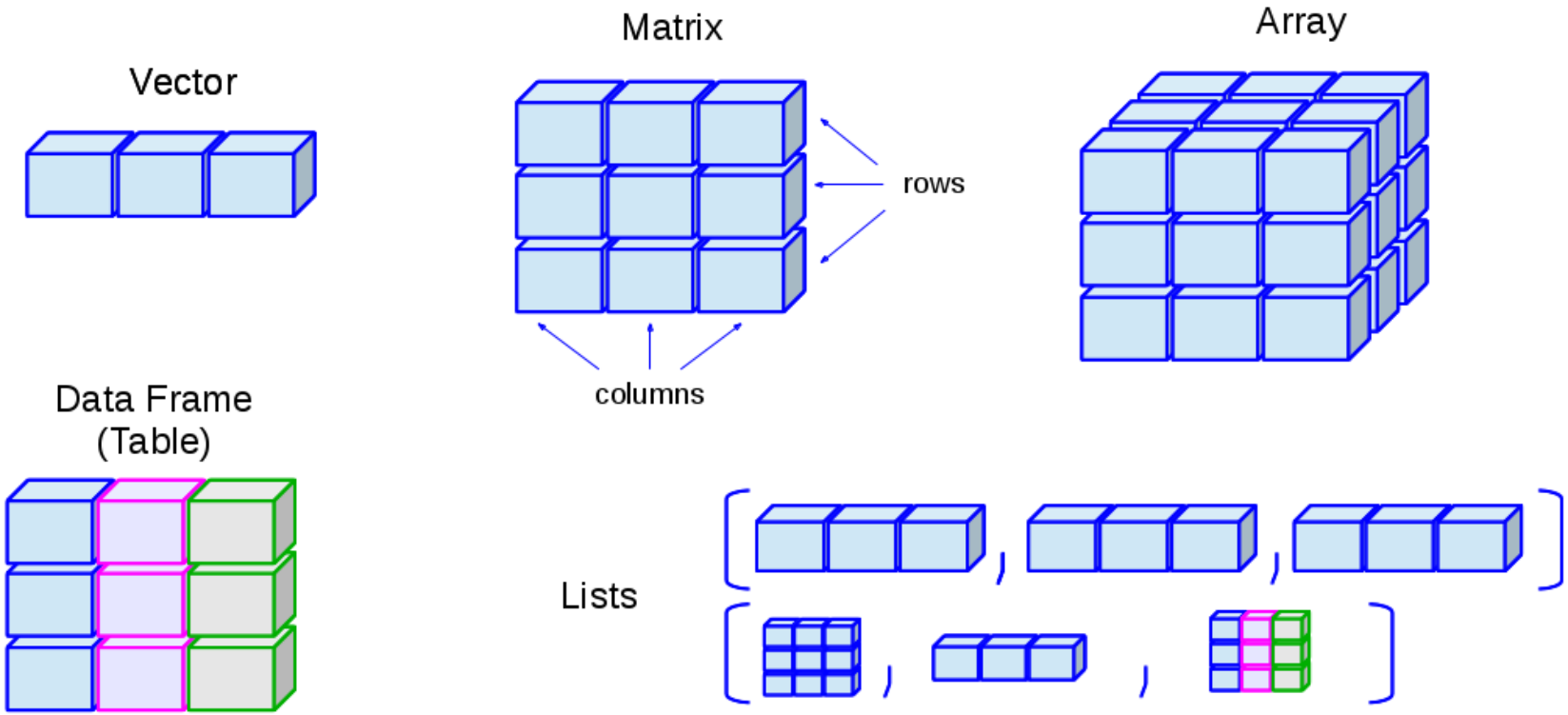
```
> date1 <- as.Date("2012-06-28")
> class(date1)
[1] "Date"
> as.numeric(date1)
[1] 15519

> date2 <- as.POSIXlt("2012-06-28 17:42")
> class(date2)
[1] "POSIXlt" "POSIXt"

> as.data.frame(unclass(date2))
  sec min hour mday mon year wday yday isdst zone gmtoff
1   0  42   17   28   5  112    4  179     0  KST     NA

> Sys.Date() - date1 # Sys.time() for system date-time!
Time difference of 4829 days
```

- For easier date/time manipulation, we'll use `lubridate` package.
  - ⇝ More on this later!

3/ Vectors and Data Frames

R provides data structures as a flexible foundation to work with data.

Data Structures



- `Vectors (1D)`, `matrices (2D)`, `arrays (nD)` can store only single data type whereas `lists (1D)` and `data frames (2D)` can store multiple data types.

  ⇝ nD = n Dimensional.

Vectors: Simplest Form of Data Structure

Generating Vectors in R

- Generating vectors is a fundamental step in data manipulation in `R`.
- Various functions and operators are available for this purpose:
  1. `c()`: combine or concatenate values
  2. Colon operator (`:`) or `seq()`: generate regular sequences
  3. `rep()`: replicates the values in x
- **Examples:**

```
> x <- c(1,2,4)
> 5:8
> seq(5,8)
> rep(1:3,2)
> i <- 2 # Generate vector of length 1

> 1:(i-1)
> 1:i-1 # Why is the output different? percedence!
```

Arithmetic Operations on Vectors

- Operations are applied to vectors **element-wise**!
  - Basic Arithmetic Operations:

```
> x <- c(1,2,4); y <- c(5,0,-1) # Generate vectors x and y
> x + y  # Addition: 6 2 3
> x - y  # Subtraction: -4 2 5
> x * y  # Multiplication: 5 0 -4
> x / y  # Division: 0.2 Inf -4
> x^y    # Exponentiation: 1.00 1.00 0.25
```

  - Modular Operations:

```
> x %% y   # Modulus (x mod y): 1 NaN 0
> x %/% y  # Integer division: 0 Inf -4
```

- **Matrix Operations:**

```
> x %*% y  # Matrix multiplication: 1 (as matrix)
```

⇝ Matrices are essentially multidimensional vectors!

Logical Operations on Vectors

- **Element-wise comparison:**

```
> x <- c(1,2,4); y <- c(5,0,-1) # Generate vectors x and y

x > y  # Greater than: FALSE TRUE TRUE
x <= y # Less than or equal to: TRUE FALSE FALSE
x == y # Equal to: FALSE FALSE FALSE
x != y # Not equal to: TRUE TRUE TRUE
```

- **Logical operations:**

```
!y              # Negation: FALSE TRUE FALSE
x >= 2 | y < 3  # OR operation: FALSE TRUE TRUE
x < 2 & y >= 3  # AND operation: TRUE FALSE FALSE

x <- 1:100
sum(x > 50)     # Q: What will R return and why?
```

Functions in R are Vectorized

- Suppose we have a function `f()` that we wish to apply to all elements of a vector `x`.
  - In many cases, we can accomplish this by simply calling `f()` on `x` itself.
- Simply put: functions are vectorized! meaning they are applied on each element of the input vector directly.
- **Examples:**

```
> sqrt(1:3)                  # Square root: 1.000000 1.414214 1.732051
> round(c(1.2, 3.9, 0.4))    # Round up|down: 1 4 0

> y <- c(12, 5, 13)
> y + 4                      # Addition: 16 9 7
> for (i in 1:length(y)) { # If fn.s were NOT vectorized.. -> loop
    print(y[i] + 4)
  }

> '+'(y, 4)                  # Addition..?
```

⇝ The operator + is really a function!

Recycling Vectors in R

- **Recycling Rule:** When vectors of unequal length are involved in an operation, the shorter vector is recycled to match the length of the longer one.
  - `c(1,2,4) + c(6,0,9)` is valid.
  - `c(1,2,4) + c(6,0,9,20,22)` generates a warning.
- **Warning Case:** You get a warning if the length of the longer vector is not a multiple of the shorter one.
- **Example:**

```
> c(1,2,4) + c(6,0,9,20,22)
[1] 7 2 13 21 24

Warning message:
In c(1, 2, 4) + c(6, 0, 9, 20, 22) :
  longer object length is not a multiple of shorter object length
```

Vector Indexing

- **Different Indexing Techniques:**

```
> y <- c(1.2, 3.9, 0.4, 0.12)
> y[1]      # Single indexing
[1] 1.2
> y[2:3]    # Range indexing
[1] 3.9 0.4
```

⇝ Single indexing retrieves one element while range indexing retrieves a sequence.

```
> v <- 3:4  # Variable indexing
> y[v]
[1] 0.4 0.12
> z <- c(5, 12, 13)
> z[-1]     # Excluding elements
[1] 12 13
```

⇝ Variable indexing allows dynamic element retrieval and indexing with negation operator `z[-1]` removes elements.

Filtering Vectors in R

- **Indexing vs. Filtering:**
  - Filtering allows you to access, change, or remove elements based on conditions rather than fixed index positions.

```
> z <- c(5,2,-3,8)
> z[c(TRUE, FALSE, TRUE, TRUE)]  # Logical filtering
[1] 5 -3 8
> which(z*z > 8)  # Identifies indices
[1] 1 3 4

# Is z[which(z*z > 8)] filtering or indexing?
```

⇝ Logical filtering: directly use logical vectors.

```
> w <- z[z*z > 8]  # Conditional filtering
> w
[1] 5 -3 8
> x <- c(1,3,8,2)
> x[x > 3] <- 0; x    # Replacing via conditions
[1] 1 3 0 2
```

⇝ Conditional filtering: use conditions that generate logical vectors.

Vectors: Mixed Element Types

- If we create a vector with mixed elements (e.g., character and numeric), the resulting vector will be a character vector.
  - Take away: Vectors must have *homogeneous* elements.
    ⇝ If not? R converts less flexible to more flexible type.

```
> num.char.vec <- c(1, 3, "five", 7)
```

- Combining vectors results in a new, larger vector. If any element is a character, the entire combined vector becomes a character vector.
  - Caveat: Be cautious of type coercion.

```
> num.vec <- c(1, 3, 5, 7)
> comb.vec <- c(num.vec, num.char.vec)
```

Vectors: Type Conversion

- **Explicit Conversion Functions:**
  - Functions like ‘as.character’ and ‘as.numeric’ enable explicit type conversion.

```
> numbers.vec <- c(-3, -2, -1, 0, 1, 2, 3)
> num2char <- as.character(numbers.vec)

> as.logical(numbers.vec) # Q: What will happen?
```

- **Be Cautious of ‘NA’:**
  - Incompatible conversions will result in ‘NA’.
    ⇝ Best Practice? Always validate the results after conversion.

```
> char.vec <- c("1", "3", "five", "7")
> char2num <- as.numeric(char.vec); char2num # Print char2num
[1]  1  3 NA  7
```

Data Frames: *'Flexible'* 2D Vectors

Or... think of an Excel sheet containing data where each column represents a `variable/feature` and each row represents an `observation`.

Generating and Indexing Matrices

- Creating matrices with `matrix()`
  - Define dimensions using `nrow` and `ncol`, or let R decide.

```
> y <- matrix(c(1,2,3,4), nrow = 2, ncol = 2); y
     [,1] [,2]
[1,]    1    3
[2,]    2    4

> z <- matrix(c(1,2,3,4), nrow = 2, byrow = T); z # Given nrow=2 & input vec. of lenght 4: ncol=?
     [,1] [,2]
[1,]    1    2
[2,]    3    4

> matrix(c(1,2,3,4), nrow = 3) # Refer to this example for the question below!
```

⇝ Q: What happens when dimensions don't align? and why?

- Accessing matrix elements using the [ ] operator: specify row and column indices to retrieve values or sub-matrices.

```
> z[1,2] # Accessing a single element: row 1, column 2
> z[1,]  # Accessing all elements in row 1
> z[,2]  # Accessing all elements in column 2
```

Introduction to Lists in R

- **Definition**: A vector that can house multiple data types, e.g., numeric, logical, strings.
  - Contains multiple components indexed with `$` or `[[ ]]` operators.
  - E.g., an employee database with `name`, `salary`, and `union membership`.
- Creating lists with `list()`

```
> j <- list(name="Joe", salary=55000, union=T); j # creating a list with component names
$name
[1] "Joe"
$salary
[1] 55000
$union
[1] TRUE

> jalt <- list("Joe", 55000, T); jalt #
[[1]]
[1] "Joe"
[[2]]
[1] 55000
[[3]]
[1] TRUE

> z <- vector(mode="list")  # since a list in R is a vector!
```

Accessing List Components

- We can point to specific component(s) of a list using the `$` or `[[ ]]` operators:

```
> j <- list(name="Joe", salary=55000, union=T)
> j$salary; j[["salary"]]; j[[2]]  # All return 55000
[1] 55000
[1] 55000
[1] 55000

> class(j[[2]])  # Identifies the data type as 'numeric'
[1] "numeric"
```

- The `[ ]` operator can also be used, but this will return another list!:

```
> j["salary"]; j[2]  # Both return a sub-list
$salary
[1] 55000

$salary
[1] 55000

> class(j[2])  # Identifies the output as a 'list'
[1] "list"
```

Data Frame: Generation & Indexing

- As we did so far, we can manually generate a data frame:

```
> kids <- c("Jack","Jill")
> ages <- c(2,10)

> data <- data.frame(kids, ages, stringsAsFactors=F); data
   kids ages
1  Jack    2
2  Jill   10
```

- And, point to specific values using the indexing operators, but..:

```
> data[[1]]; data$kids # list-like indexing
[1] "Jack" "Jill"
[1] "Jack" "Jill"

> data[,1] # matrix-like indexing
[1] "Jack" "Jill"
```

- It's much more common to import data from external sources rather than manually creating one!

Data Frame: Importing Data

- Importing data in R using `read.table()` or `read.csv()`
  - Your choice of import method may vary depending on the file format of your data, such as text or CSV.

```
> exam1 <- read.csv(url("https://bit.ly/3ZqTQLY")); exam1
  ID Exam1 Exam2 Quiz
1  1   3.3   2.0  3.7
2  2   4.0   4.0  4.0
3  3   2.3   0.0  3.3
4  4   2.2   1.0  3.3
5  5   3.1   1.2  3.9
> exam1 <- read.table(url("https://bit.ly/3ZqTQLY"), sep = ",", header = T)
```

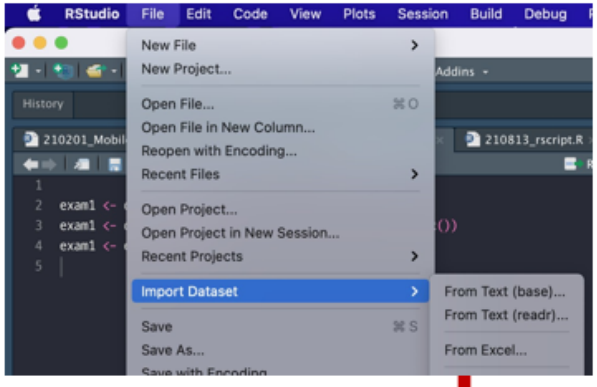
- If importing data from a local file:

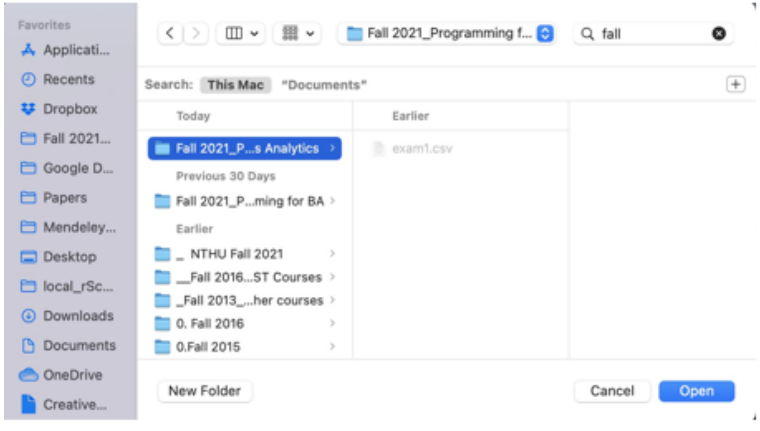
```
# Use full file path:
> exam1 <- read.csv(file = "/Users/J1Yoo/Documents/Fall_2022_PBA/import_data.csv"); exam1

# Or..
> setwd("/Users/J1Yoo/Documents/Fall_2022_PBA") # set up the working directory,
> exam1 <- read.csv(file = "./import_data.csv") # then use a relative file path.
```

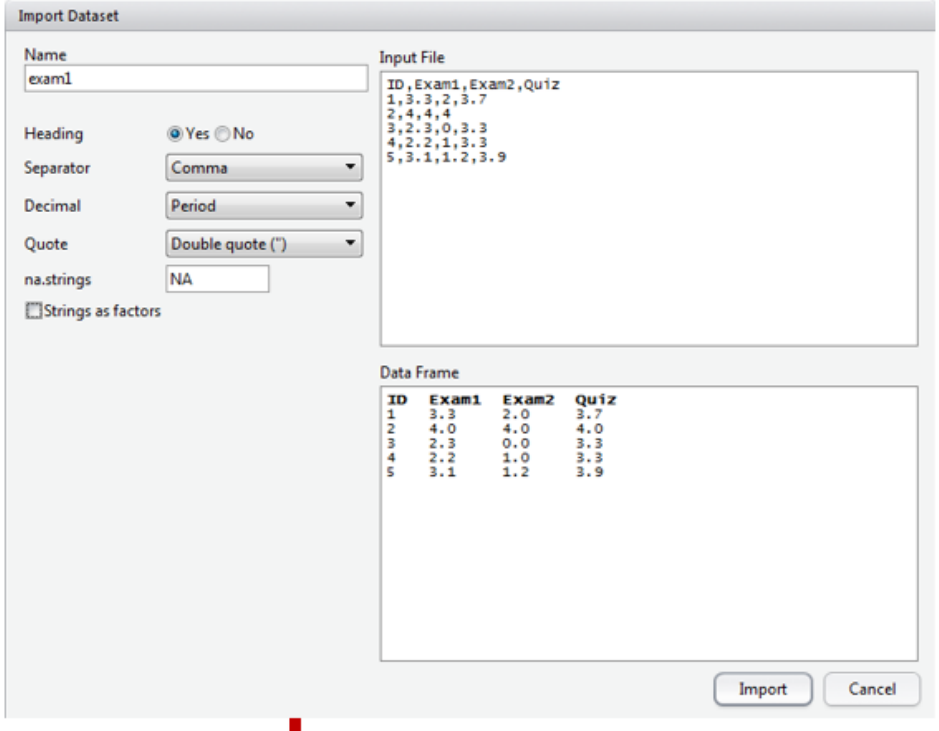
  - *Note:* use your file/directory paths when importing data from local files!

Importing Data via RStudio Menu

- For those who prefer GUI, RStudio offers menu-based data import.
  - **Advantage of Code-Based Import**: It creates an R script that serves as a reusable record, allowing for easy data re-importation in future sessions.







Auto. executes the func. below

```
> exam1 <- read.csv("~/Documents/Fall 2021_Programming for Business Analytics/
exam1.csv", header=FALSE)
>   View(exam1)
```

4/ User-Defined Functions

User-defined functions in *R* enables us to work with data more efficiently, offering customized solutions for business challenges.

Functions in R

- Recall that functions in R are objects which takes in input arguments then performs a specific task (i.e., `function(arglist) expr`).
- **Examples:**

```r
# Fahrenheit to Celsius function
f <- function(x) (x-32)/1.8
f <- function(x) {(x-32)/1.8}
f <- function(x) return((x-32)/1.8)
f(100)  # Output: [1] 37.77778

# Celsius to Fahrenheit function
f <- function(x) x*1.8+32
f(37.7778)  # Output: [1] 100
```

Setting Default Values in Functions

- Default values in function arguments simplify function calls and improve code reusability.

```
> f <- function(a, b = 1, c = 2, d = NULL) {
    cat(a,b,c,d)  # cat() concatenates values then prints it on screen
}

> f(1)          # Uses default values for b, c, and d
1 1 2

> f(1,2,3,4)    # Overrides default values
1 2 3 4

> f(1,2,3)      # Partially overrides default values
1 2 3

> f(3, c=5)     # Positional and named argument matching
3 1 5

> f(d=7)        # Missing required argument 'a' leads to a runtime error
```

- **Tip:** Default values are particularly useful for:
  - Making functions more user-friendly (convenience).
  - Allowing optional customization (flexibility).
  - Avoiding missing argument errors (error prevention).

Nesting Functions in R

- Functions can be nested within other functions to create more complex operations.

```
# Define function f that calls a nested function, g
f <- function(x,y){
  print(x)
  g <- function(y) { print(y) }
  g(y)
}

f(1, 2)
[1] 1
[1] 2
```

- **Tip:** Use `args()` and `body()` to inspect a function's input arguments and code body.
    - Especially useful for understanding how complex functions work.

Passing on Arguments to Other Functions

- The `...` argument allows you to pass a set of arguments to another function.

```r
# Define function f
f <- function(x, y) {
  print(x)
  print(y)
}

# Define function g that calls f
g <- function(z, ...) {
  print(z)
  f(...)
}

# Call function g
g(1, 2, 3)  # Output: g prints 1, f prints 2 and 3
[1] 1
[1] 2
[1] 3
```

- **Key Takeaway:**
  - The `...` parameter allows you to pass arguments through to other functions without explicitly naming each one.

Understanding Variable Scope in R

- A **scope** refers to the region of code where a variable can be accessed or modified.
- **Types of Scopes:**
  - **Global scope**: accessible throughout the code.

```
global_var <- "I am global"
```

⇝ Global variable is accessible from any part of the code, both inside and outside of functions.

  - **Local scope**: the region within a function.

```
my_function <- function() {
  local_var <- "I am local"
}
```

⇝ Variables declared here are accessible only within that function. Each function call creates a new local scope.

Example: Why Understanding Scope Matters

```r
# Student wants to keep track of a total
total <- 0

add_to_total <- function(value) {
  total <- total + value  # Creates LOCAL total!
  print(paste("Total is now:", total))
}

add_to_total(10)  # Output: Total is now: 10
add_to_total(5)   # Output: Total is now: 5 (not 15!)

print(total)      # Output: [1] 0 (Still zero!)
```

- What happend?
  - Without understanding scope, the student expects `total` to be 15, but it remains 0. Each function call creates a new local `total` instead of modifying the global one.
- Key takeaway:
  - Understanding scope prevents logical errors (bugs) caused by variables with the same name in different scopes.

Function: Scoping Rules (1-2)

- **Rule #1. Global Variables:** Variables declared in console can be accessed anywhere.

```r
n <- 1
f <- function () print (n)
f()  # Output: [1] 1

n <- 2
f()  # Output: [1] 2
```

- **Rule #2. Local Variables:** Variables declared inside a function are prioritized (shadowing/masking global variables with the same name).

```r
n <- 100
f <- function () {n <- 1; print (n)}

f()  # Output: [1] 1
```

Function: Scoping Rules (3-4)

- **Rule 3. Local Variable Accessibility:** Variables declared inside a function cannot be accessed from outside.

```r
rm(list=ls()) # Warning: this will remove everything in your R Session!

f <- function () x <- 1
f()
x  # Error: object 'x' not found
```

- **Rule 4. Argument Priority:** Input arguments are prioritized over global variables. (Note: Arguments ARE local variables, thus Rule 4 demonstrates another angle on the shadowing rule.)

```r
n <- 100
f <- function (n) print (n)
f(1)  # Output: [1] 1
```

Function: Scoping Rules (5)

- **Rule 5. Nested Functions:** The same scoping rules apply at each level. Variables are looked up by searching outward through enclosing scopes (`nested scope lookup/chain`).

```
1  f <- function (x) {
2    a <- 2
3    g <- function (y) {
4      print (y + a)
5    }
6    g(x)
7  }
8
9  a <- 100
10
11 f(1)  # Q: what would R return?
```

- **Question:**
  - If we commented out `a <- 2` (line 2) from function f, what would happen?

Function: Scoping Rules (6)

- **Rule 6. Super Assignment Operator:** Use `<<-` to modify or create a variable in the nearest enclosing (parent) environment where it exists.

```r
b <- 0
f <- function() {
  a <- 1
  g <- function(a, b) {
    a <<- 2
    b <<- 2
    print(a)
    print(b)
  }
  g(3,3)
  print(a)
  print(b)
}
f()  # Output: [1] 3, [1] 3, [1] 2, [1] 2
```

- **Question:**
  - What if we were to use `<-` in place for `<<-`?
- **Caveat:**
  - Allows modifying variables in outer scopes. This is an advanced feature used sparingly (e.g., package development). For typical data analysis, have functions return values and explicitly assign them to variables.

Writing Your Own Binary Operator

- **Introduction:** Operators in R are functions. You can create your own binary operator.

```r
"%a2b%" <- function(a,b)
  return(a+2*b)
3 %a2b% 5  # Output: [1] 13
```

- **Another Example:**

```r
"%sdf%" <- function (a,b) {
  sdfxy <- setdiff(a,b)
  sdfyx <- setdiff(b,a)
  return(union(sdfxy, sdfyx))
}
x <- c(1,2,5)
y <- c(5,1,8,9)
x %sdf% y  # Q: What would happen?
```