Rintro

2025-11-18

R Introduction

Printing

to print, you can run the function print() in order to output anything. You can run it in the console, or in your R-Code chunk. Here's a classic example.

```
print("Hello World!")
```

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

I used print() to print a string, and the computer returned me the string.

Data Types

R Works with several data-types called "Higher data types." Numeric - Anything numeric, like -12.5 or 42. R sometimes refers to numbers as "integers" (if they are whole numbers) or "doubles" (if it is storing a number in a way that could handle decimals, since this (historically) takes up twice the amount of computer memory as an integer).

Character - Anything textual (strings), like letters, symbols, and the characters that represent numbers (if there is quotation marks surrounding them - used as strings essentially) Note: in other programming languages this datatype is sometimes known as a "character string" or just a "string".

Boolean - holds a value that is either TRUE or FALSE. (This is sometimes also referred to as the "logical" data type.)

Examples "This is a ___ Data type" <- Character 676941 <- Numeric TRUE <- Boolean.

typeof() func Regarding Data types, typeof() returns the data type of its input.

```
typeof("Hi")
```

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

Operators

Operators 'operate' on data, self explanatory. ### Numerical operators + | Addition operator.

```
1 + 1
```

```
## [1] 2
```

- Subtraction operator

```
2 - 3
```

```
## [1] -1
```

- Multiplication operator

```
5 * 3
```

```
## [1] 15
```

/ | Division operator

```
1/2
```

```
## [1] 0.5
```

^ | Exponential operator

```
2^3
```

```
## [1] 8
```

%% | Modulus operator

```
5 %% 2
```

```
## [1] 1
```

Order of Operations R follows GEMDAS (Mathematical Order of Operations)

```
3 + 5 / 5 * 3 ^ 2
```

```
## [1] 12
```

The order can be changed using parenthesis, just like normal math

```
(3+5)/(5*3)^2
```

```
## [1] 0.03555556
```

R also handles expressions line-by-line. You can do multiple separate calculations by putting each one on a separate line. When R reads your code, it treats everything on one line as a single expression that is separate from other lines:

```
#3+3 5*5
#Error: unexpected numeric constant in "3+3 5"
```

```
3+3
```

```
## [1] 6
```

```
5*5
```

```
## [1] 25
```

However, R discards results after working through an equation unless stored.

Variables

Variables are R's way of saving values for use later in the program, instead of copy-pasting the same expression or lines of code and altering it to fit specifically into your new use.

for example,

```
#use1
(3+3) * 7
```

```
## [1] 42
```

```
#use2
(3+3) * 6
```

```
## [1] 36
```

is much less efficient than

```
a <- 3+3
a*7
```

```
## [1] 42
```

```
a*6
```

```
## [1] 36
```

R Studio keeps track of your variables in the ENVIRONMENT tab. This is where any objects in your code are stored, to make it easy for you to keep track of them.

R uses '<-' as the assignment operator - similar to python's '=' assignment operator. the Variable goes to the left, and the expression/value goes to the right.

outputs Variable assignments have no real result- its just a value being stored.

```r
b <- "Hello, World!"
```

As you can see, there was no output. However, just typing in a variable's name will output.

```r
b
```

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

The variability of variables You can change a variable's stored value by just re-assigning it.

```r
b
```

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

```r
b <- 67

b
```

```
## [1] 67
```

As you can see, all it took was a simple re-assignment.

Assignment precedence The assignment operator has precedence, too, but its extremely low. It will only assign after all the expression has been iterated on the right side.

```r
c <- 6 + 7 * 8 * 3 ^ 2
c
```

```
## [1] 510
```

Variable naming conventions You can name a variable anything you choose... with certain restrictions. R does not allow variables to begin with a number, such as 100, or contain spaces within the variable, such as one hundred.

```r
#100 <- "Hello, World!"

#one hundred <- "Hello, World!"
#Error in 100 <- "Hello, World!" :
#invalid (do_set) left-hand side to assignment
```

Technically you can get around this restriction by putting an invalid variable name inside backticks, for example: `100` or `one hundred`.

```r
`100` <- "Hello, World!"
`one hundred` <- "Hello, World!"
```

However, it is generally advisable to avoid naming variables with invalid names, since it will make your life a lot easier if you don't constantly have to include extra backticks. For example, we might replace spaces with underscores: one_hundred.

```
one_hundred <- 100
```

Boolean Data

Boolean operators Boolean operators are operators that can be used to return boolean values in expressions in R. examples:

```
4 < 5
```

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

```
4 > 5
```

```
## [1] FALSE
```

```
4 == 5
```

```
## [1] FALSE
```

Remember that assignment always takes place last. The expressions can be stored in variables,

```
boolop <- 4 < 5
boolop
```

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

The <> operators measure numerical values against one another, like the less-than and greater-than mathematical operators.

```
10 < 9
```

```
## [1] FALSE
```

```
10 > 9
```

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

The == operator measures values of each datatype for exact similarity, like the = mathematical operator.

```
10 == 9
```

```
## [1] FALSE
```

```
10 == 10
```

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

```
"Hello" == "Hello"
```

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

```
TRUE == TRUE
```

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

The <= and >= operators also function similarly to the less-than or equal-to and greater-than or equal-to operators in mathematics.

```
10 <= 100
```

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

```
10 <= 10
```

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

As they both fit the less-than or equal-to criteria, they both return the same boolean value.

Vectors (Arrays)

Vectors are just that- arrays. R has several types of 'Container' which hold multiple pieces of data. That container can be referred to as a single variable, allowing use of longer data-sets in more complex programs. For example, instead of:

```
  q <- 1
  r <- 2
  s <- 3
  q
```

```
## [1] 1
```

```
  r
```

```
## [1] 2
```

```
  s
```

```
## [1] 3
```

You can create a vector with c(1,2,3), putting the objects we want to combine inside the parentheses (and separated by commas).

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

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

An important caveat to this - All the data in a vector must be the same type of data. For example, a vector could contain all numerics, or all characters, but not a mix of the two.

Another vector example:

```r
c("This", "is a", "vector!")
```

```
## [1] "This"    "is a"    "vector!"
```

Vector Operations Operators[see operators segment for list] can be used on vectors. For example, here's me adding two vectors together.

```r
v1 <- c(1,2,3)
v2 <- c(4,5,6)
v1 + v2
```

```
## [1] 5 7 9
```

Vectors of different lengths undergo unique processes when they are added together. For example:

```r
v3 <- c(10, 20, 30, 40, 50)
v4 <- c(1, 2)
v3 + v4
```

```
## Warning in v3 + v4: longer object length is not a multiple of shorter object
## length
```

```
## [1] 11 22 31 42 51
```

After returning a warning, reminding you of the differing lengths of the two vectors, R does something to the smaller vector to make the lengths the same- Recycling. Recycling repeats the shorter vector over and over until it is the same length as the longer vector. I.e. v4 will be repeated 2.5 times to become (1,2,1,2,1) before adding it to v3.

Dataframes

Data is usually handled in the form of a TABLE. Usually, each column is some type of measurement (variable). IMPORTANT NOTE - note that the variable represented by a column is different to the R variables we learned about.

Just like R has a vectors to store a single series of values, it also contains a structure called a dataframe to hold a table of data. In fact, R uses vectors to create dataframes: behind the scenes, each column of a dataframe is stored within a vector.

```r
vi <- c('S1', 'S2')
va <- c(5.2, 8.1)
vb <- c(6.4, 7.9)
data.frame(ID = vi, MeasurementA = va, MeasurementB = vb)
```

```
##   ID MeasurementA MeasurementB
## 1 S1          5.2          6.4
## 2 S2          8.1          7.9
```

Dataframes can also be assigned to R variables so that we can store and retrieve them. For the purposes of this section, suppose that the R variable df contains a dataframe of the table above.

```
df <- data.frame(ID = vi, `Measurement A` = va, `Measurement B` = vb)
```

If we run the variable by itself, we will print out the dataframe:

```
  df
```

```
##   ID Measurement.A Measurement.B
## 1 S1           5.2           6.4
## 2 S2           8.1           7.9
```

example df

ID Measurement A Measurement B 1 S1 5.2 8.1 2 S2 6.4 7.9

Each column's data can be extracted easily with the $ operator. This is a rudiamentary data wrangling operator, and will be better replaced later on.

```
df$ID
```

```
## [1] "S1" "S2"
```

Each column can only contain one datatype, because R uses vectors in those columns. While columns are similar to variables, they are technically not variables. One such similarity is how we have to write the column name in code. We run into the same restrictions on column names that apply to R variable names, e.g. they cannot contain spaces, or start with a number. However, there is no restriction on creating dataframes with column names that violate these rules, so you will often encounter dataframes with column names that cannot be referenced as R variables.

Functions

R Allows the coder to create blocks of code that saves time with repetitive tasks- functions. Technically, a function is a variable that saves code.

For example, you could create a function to calculate the sum of two numbers:

```
adder <- function(number1, number2) {
    result <- number1 + number2
    return(result)
}
```

To use this function, you just reference it along with values for the parameters (the variables in the parenthesis)

```
adder(6, 7)
```

```
## [1] 13
```

Function Components Line 1 - adder <- function(number1, number2) { Recalling that functions are essentially variables that contain code, we understand that adder is the function's name, with number1 and number2 acting as parameters to be filled in by the caller. We could make them 1 and 2, 3 and 3, or 4 and 100000, just as long as they're numbers. The curly bracket at the end indicates that code on the following lines are to be included as parts of the function.

Line 2 - result <- number1 + number2 Essentially the function's job - it adds your two provided parameters and stores them in a temporary variable. While not necessary, the indentation is good programming practice as it improves function readability.

Line 3 - return(result) This line takes your temporary variable 'result' and prints it out to the reader. Same indentation rules apply.

Line 4 - } Signals an end to the function itself. Allows further coding without the computer mixing it up for part of the function.

Complex functions Adder is a very simple function, and some may say simply using the '+' operator would be easier. However, functions are designed to handle even more complex code.

```
hypotenuse <- function(a, b) {
  c <- (a^2 + b^2)^0.5
  return(c)
}
```

This function calculates the hypotentuse. Line one creates the function name 'hypotenuse' and its parameters, a and b. Line two carries out the entire pythagorean theorem - sqrt(a^2 + b ^2) - and stores the result into c. line 3 prints out c, and line 4 closes the function. instead of re-writing all of this code everytime a hypothesis is needed, you only write it once, then call the name and parameters. Here's how this is useful.

```
hypotenuse(3, 4)
```

```
## [1] 5
```

```
hypotenuse(5, 12)
```

```
## [1] 13
```

```
hypotenuse(7, 36)
```

```
## [1] 36.67424
```

See how fast I was able to get 3 hypotenuses of different numbers instead of re-writing all the code for each?

Function review

- the 'function' keyword tells R that you're creating a re-usable function.
- function's following parenthesis contain parameters - temporary variable names for the function to replace.
  - You can have as many or few parameters as you want, just seperate them with commas
- Function code goes within two curly-brackets.
- Functions can end however you'd like - return(...) with the ... being your desired value, a new variable created, etc.
- Store the function into a name, like a variable with the assignment operator
- Run the code anytime by writing the function name, parenthesis, and defined parameters inside. hypotenuse(6,7)

Packages

Packages are collections of pre-written functions or other neat tools, such as datasets or programs, that can be imported into your own code.

Installing packages R comes with many pre-installed packages, visible by going into RStudio's *packages* tab in the bottom right pane.

To install a package, either - Go to the Packages tab, click install, and search the name of your desired package. - Enter 'install.packages("some_package_name")' into the console.

Package installation can take some time, so its worth checking to see if you have a package installed before you install one. To check, Go to the packages tab, search up the package name.

Loading packages Packages only need to be installed TO YOUR COMPUTER once. Each project requires you to re-load those packages to access them, though.

To reload, use the function library("package_name").

Conclusion

Thank you for reading my introduction to R.