Programming Basics I

Plan for the week

- M: Functions and Conditionals (`if` and `if else`)
- W: Loops (`for` and `while`)
- F: Section
    - Programming practice

Abstraction

- Copying and pasting code is something you *never* want to do!
- Functions are one form of abstraction that can be used in place of copy and paste. Loops are another.

Functions

What is a function?

In spreadsheets, we learned about formulas, like `AVERAGE`, `MAX`, `CONCATENATE`, and `VLOOKUP`.

In R, we learned about dplyr, which has functions, like `select`, `filter`, `arrange`, `mutate`, and `group_by`.

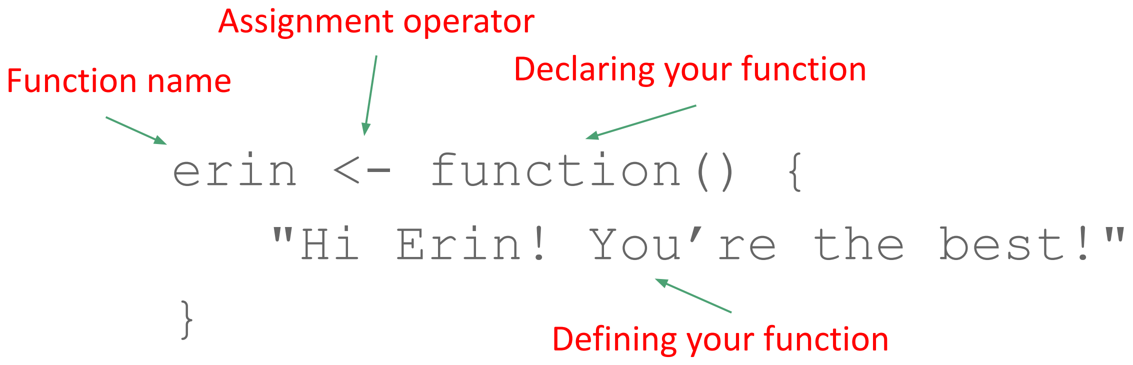
These are all "built-in" functions, meaning they have been written for you.

But you can also write your own functions, to perform tasks unique to your data.

Example: Function Definition

```
erin <- function() {
   "Hi Erin! You’re the best!"
}
```

Example: Function Definition



Assignment operator

Function name

Declaring your function

```
erin <- function() {
    "Hi Erin! You’re the best!"
}
```

Defining your function

You declare a function by writing `<- function()`

The code between the `{ }` brackets is called the body of the function.

The body is where you define your function.

Example: Function Definition

```
erin <- function() {
   "Hi Erin! You’re the best!"
}
```

Example: Function Application

```
> erin()
[1] "Hi Erin! You’re the best!"
```

Another example: Function Definition

```
anna <- function() {
   "Hi Anna! You’re REALLY the best!"
}
```

Abstraction

```
erin <- function() {
    "Hi Erin! You’re the best!"
}

anna <- function() {
    "Hi Anna! You’re REALLY the best!"
}
```

These functions are REALLY similar.

Can we generalize somehow?

Generalizing code for easy reuse is called abstraction!

Abstraction

parameter

```
TA <- function(name) {
   paste("Hi ", name, "! You’re the best!")
}
```

Built-in function

Abstraction is a really important concept in programming. You *never* want to copy-and-paste code. You want to write general code that can be reused.

`name` is a parameter, a.k.a. an argument, on which the function depends.

`TA` can be called with any string as an input:
e.g., `TA("Erin")` or `TA("Anna")`

Abstraction

```
paste("Hi ", name, "! You’re the best!")
```

Built-in function

Example: Function Application

```
> paste("Hi ", "Erin", "! You’re the best!")
[1] "Hi  Erin ! You’re the best!"

> paste("Hi ", "Anna", "! You’re the best!")
[1] "Hi  Anna ! You’re the best!"
```

Abstraction

```
paste("Hi ", name, "! You’re the best!", sep = "")
```

Built-in function

Example: Function Application

```
> paste("Hi ", "Erin", "! You’re the best!")
[1] "Hi Erin! You’re the best!"

> paste("Hi ", "Anna", "! You’re the best!")
[1] "Hi Anna! You’re the best!"
```

Abstraction

```
TA <- function(name) {
   paste("Hi ", name, "! You’re the best!", sep = "")
}
```

Built-in function

Example: Function Application

```
> TA("Erin")
[1] "Hi Erin! You’re the best!"
> TA("Anna")
[1] "Hi Anna! You’re the best!"
```

Input from the Console

Input from the console

```
readline(prompt = "Enter Name: ")
```

`readline` is a built-in function that gets input from the console and displays it.

`prompt = "Enter Name: "` is a named parameter of `readline`.

Input from the console

```
enterName <- function() {
  readline(prompt = "Enter Name: ")
}
```

```
> enterName()
Enter Name: Juho
[1] "Juho"
```

```
> enterName
function {
    ...
}
```

Input from the console

```
enterName <- function() {
  name <- readline(prompt = "Enter Name: ")
  return(name)
}
```

Run the following in an R Script or R Markdown

```
> paste("Hello, ", enterName(), "!")
```

Enter Sarah from the console, and R will display `Hello, Sarah!`

Function Composition

```
> TA(enterName())
```

What do you think this composition of function calls will do?

Pro-Tip: Always use informative names for functions and variables!

Function Composition

```
> TA(enterName())
Enter Name: Will
[1] Hi Will! You’re the best!
```

Scope

Local and Global Variables

```
printNumber <- function() {
    var1 <- 1
    print(var1)
}
```

`var1` is a local variable, a variable declared inside a function.
Because it is "local" to the function, it does not exist outside the function.

```
> print(var1)
[1] Error: object 'var1' not found
> printNumber()
[1] 1
```

Local and Global Variables

```
var2 <- 2
printNumber <- function() {
   print(var2)
}
```

`var2` is a global variable, a variable declared outside a function.
Because it is "global," it can be accessed both inside and outside the function.

```
> print(var2)
[1] 2
> printNumber()
[1] 2
```

Local and Global Variables

```
xSquared <- function(num) {
  num * num
}
```

`num` is a local variable, so it cannot be accessed outside `xSquared`

```
> print(num)
[1] Error: object 'num' not found
> xSquared(5)
[1] 25
```

Local and Global Variables

```
num <- 10
xSquared <- function(num) {
  num * num
}
```

`num` is both a local and a global variable, so it *can* be accessed outside `xSquared`

```
> print(num)
[1] 10
```

```
> xSquared(num)
[1] 100
```

Local and Global Variables

```
num <- 10
xSquared <- function(num) {
  print(num)
  num * num
}
```

`num` is both a local and a global variable, so it *can* be accessed inside `xSquared`

```
> xSquared(num)
[1] 10
[1] 100
```

Conditionals

Up next: Conditionals

So far, you learned about writing your own functions, a form of abstraction that allows you to perform the same task on different inputs in a clear and concise manner.

Now, suppose you want to perform different tasks under different conditions.

For example, what if you want to do something different for different ranges of numbers? E.g., one thing for those between -1 and 1, and another, for those between 1 and 100.

How can you write functions that make decisions based on these predicates?

Introducing, conditionals!

Logicals

Logicals are one of the basic R data types.

They are either `TRUE` or `FALSE`.

In other programming languages, logicals are called booleans.

Relational Operators

When we compare data, we uncover a relationship between them.

A predicate evaluates a relation, as a logical, so either `TRUE` or `FALSE`.

`3 > 2` is an example of a predicate
`“David” < “Shivani”` is as well

Relational operators build predicates.

The most common relational operators are:
`== != < > <= >=`

Operator	Meaning
==	equal to
!=	not equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

Logical Operators

We use logical operators to link predicates.

```
> x <- TRUE
> y <- FALSE

> !x
FALSE
> x && y
FALSE
> x || y
TRUE
```

Operator	Meaning
!	NOT
&&	AND
||	OR

Logical Operators

Element-wise AND and OR can be applied to logical vectors.

```
> x <- c(TRUE, FALSE, 0, 10)
> y <- c(FALSE, TRUE, FALSE, TRUE)

> x & y
FALSE FALSE FALSE TRUE

> x | y
TRUE TRUE FALSE TRUE
```

Operator	Meaning
&	Element-wise AND
|	Element-wise OR

NOTE: 0 is `FALSE`, and all non-zero numeric values are `TRUE`.

Missing Values

Sometimes, a vector might be missing values.

How can we find out if we have such faulty data?

We use `is.na()`, which returns a logical vector of the same length as its input. The entry in the output vector is `FALSE` wherever there is a missing value.

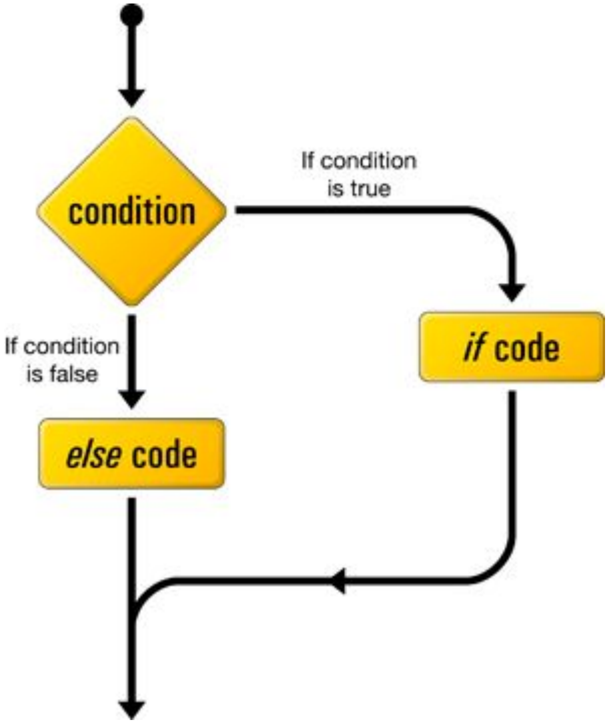
```
> is.na(c(1, NA, 2))
FALSE TRUE FALSE
```

If Statement

We use conditionals to make decisions based on predicate values.

In R, the syntax for conditionals is an if-else statement:

```
if (predicate) {
   # Something happens
} else {
   # Something else happens
}
```



Example

The clause in an if statement is executed only when the predicate is true.

```
x <- 1

if (x < 0) {
   # This code is not executed
   print(“x is negative”)
}

if (x > 0) {
   # This code IS executed
   print(“x is positive”)
}
```

If-Else Example

The else clause is executed when the predicate is false.

```
x <- 1

if (x < 0) {
  print(“x is negative”)
} else {
  print(“x is greater than or equal to 0”)
}
```

If-Else Example

You can include as many else clauses as you like.

```
if (x > 0) {
  print(“x is greater than 0”)
} else if (x == 0) {
  print(“x is equal to 0”)
} else {
  print(“x is less than 0”)
}
```

Another Example

You can combine predicates in conditionals using logical operators.

```
x <- 1
if (x > 0 && !x == 0) {
   print(“x is positive, and x is not 0”)
}
```

Nested If-Else Statements

You can also nest if-else statements inside one another:

```
if (x > 0) {
   if (x < 10) {
     print("x is greater than 0 and less than 10”)
   } else {
     print("x is greater than or equal to 10”)
   }
}
```

Switch Statement

If you have multiple predicates to test, it might be too much work to write if and else-if statements for each one. In this case, you can use a switch statement:

```
a <- 10
b <- 10
symbol <- readline(prompt = "Enter an ARITHMETIC OPERATOR: ")
switch(symbol,
       "+" = print(a + b),
       "-" = print(a - b),
       "*" = print(a * b),
       "/" = print(a / b))
```

Next time: Repetitive tasks

Today, you saw examples of conditionals.

You can use them to write code to perform a task when a condition is met.

What if you want to execute a task many many times, so long as a condition is met?

You will learn to write loops next time!