



Karolinska
Institutet

Functions, Loops, Conditionals and Classes

Reflect

What do you know about

- Functions
- Loops
- Conditionals

How are you currently using

- Functions
- Loops
- Conditionals

Part of Intended Learning Outcome

- Identify situations suitable to the use of functions, loops and conditionals
- Construct their own algorithms, incorporating self-created functions, loops and conditionals
- Evaluate the efficiency of code, selecting the best option of, for example functions, to solve a certain problem
- Use RMarkdown to create easier markup and navigation, as well as create PDF files and websites

Functions

The what, how and why

Definition

Functions "Encapsulate" a task, i.e., combine many instructions into a single line of code.

Most programming languages provide built-in functions that would otherwise require many steps to accomplish.

How Does it Work?

When a function is "called" the program "leaves" the current section of code and begins to execute the first line inside the function. Thus, the function "flow of control" is:

1. The program comes to a line of code containing a "function call".
2. The program enters the function (starts at the first line in the function code).
3. All instructions inside of the function are executed from top to bottom.
4. The program leaves the function and goes back to where it started from.
5. Any data computed and RETURNED by the function is used in place of the function in the original line of code.

Why are Functions Used?

1. Functions allow us to reduce our program into several sub-steps.
2. They allow reuse of code instead of rewriting it.
3. Functions allow a clean environment (local variables only "live" if the function does).
 - In other words, function_1 can use a variable called i, and function_2 can also use a variable called i and there is no confusion. Each variable i only exists when the computer is executing the given function.

Function Anatomy

```
my_func <- function(x = 0, y = 1) {  
  z <- x * y  
  return(z)  
}
```

```
my_func(x = 10, y = 5)
```


Function Anatomy

Function Name

- This is the identifier used to call the function. It should be chosen to describe the purpose or action of the function.

```
my_func <- function(x = 0, y = 1) {  
  z <- x * y  
  return(z)  
}
```

```
my_func(x = 10, y = 5)
```

Function Anatomy

Parameters (or Arguments)

- These are variables or values that the function accepts as input. Parameters are optional, and a function may have zero or more parameters. Parameters are enclosed in parentheses after the function name.

```
my_func <- function(x = 0, y = 1) {  
  z <- x * y  
  return(z)  
}
```

```
my_func(x = 10, y = 5)
```

Function Anatomy

Function Body

- This is the block of code enclosed within curly braces {} in languages like R, C++ and JavaScript or indented in Python. It contains the actual instructions and logic of the function.

```
my_func <- function(x = 0, y = 1) {  
  z <- x * y  
  return(z)  
}
```

```
my_func(x = 10, y = 5)
```

Function Anatomy

Return Statement

- Functions often produce a result, which is specified using the return statement. It sends a value back to the caller. A function can have multiple return statements, but when one is executed, it exits the function immediately.

```
my_func <- function(x = 0, y = 1) {  
  z <- x * y  
  return(z)  
}
```

```
my_func(x = 10, y = 5)
```

Function Anatomy

Function Call

- To execute a function and utilize its functionality, you need to call it by its name, passing the required arguments.

```
my_func <- function(x = 0, y = 1) {  
  z <- x * y  
  return(z)  
}
```


my_func(x = 10, y = 5)

Additional Features of Functions – Scope

Scope


- Functions can have their own local scope, meaning that variables declared within the function are not accessible from outside the function. This helps encapsulate functionality and prevent unintended variable clashes.

```
my_func <- function(x = 0, y = 1) {  
  z <- x * y  
}
```

A purple arrow points from the text "local" inside a purple box to the assignment operator "<-" in the line "z <- x * y".

local

```
my_func <- function(x = 0, y = 1) {  
  z <<- x * y  
}
```

A purple arrow points from the text "not local" inside a purple box to the double assignment operator "<<-" in the line "z <<- x * y".

not local

Additional Features of Functions – Default arguments

Default arguments

- Allow you to specify a default value for one or more function arguments. When a function is called, if a value is not provided for a parameter with a default value, the default value is used instead.

```
my_func <- function(x = 0, y = 1) {  
  z <- x * y  
}
```

```
my_func <- function(x, y) {  
  z <- x * y  
}
```

Additional Features of Functions – Closures

Closures

- Is a concept in programming languages, particularly in languages that support first-class functions or function pointers. A closure is a function that retains access to variables from the outer (enclosing) function's scope even after the outer function has finished executing.

```
outer_function <- function(x) {  
  inner_function <- function(y) {  
    return(x + y)  
  }  
  return(inner_function)  
}
```

```
closure <- outer_function(10)  
result <- closure(5)
```


When are Closures Used?

Closures are useful for a variety of programming tasks, including:

- Data Encapsulation – Closures allow you to encapsulate data within a function, making it inaccessible from outside the function.
- Function Factories – Closures allow creation of functions dynamically with specific behavior, depending on the values captured from the outer scope.
- Callback Functions – Closures are used to pass functions as arguments to other functions, such as in event handlers or asynchronous programming.

Additional Features of Functions – Anonymous Functions

Anonymous function (lambda expression)

- Anonymous functions are functions that are defined without a name. They are typically used for short, simple operations that can be defined in a single line of code.

```
function(x) {  
  x * 5  
}
```

```
function(x) x * 5
```

! Only works in other function !

Additional Features of Functions – Ellipses

Ellipses (dots)

- "ellipses" typically refer to three dots (...) used in a function's argument list. The use of ellipses in function definitions provides a mechanism for functions to accept a variable number of arguments or to pass additional arguments to other functions.

```
my_func <- function(x, y, ...) {  
  z <- x + y  
  
  if (length(list(...)) > 0) {  
    z <- z + sum(...)  
  }  
  
  return(z)  
}
```

Additional Features of Functions – Recursion

Recursion

- A function can call itself, which is known as recursion. Recursive functions are useful for solving problems that can be broken down into smaller, similar subproblems.

Example – factorial 10!

```
factorial <- function(n){  
  if (n == 0 | n == 1) {  
    return(1)  
  } else {  
    return(n * factorial(n - 1))  
  }  
}
```

```
factorial(10)
```

Additional Features of Functions – Optional Documentation

Optional Documentation (Docstrings)

- It is a good practice to include comments or documentation strings (docstrings) that describe the purpose, usage, and behavior of the function. This helps other developers understand how to use the function correctly.

```
#' Multiply two values  
#' @param x A numeric value  
#' @param y A numeric value  
#' @return Returns multiple of x  
and y
```

```
my_func <- function(x = 0, y = 1) {  
  z <- x * y  
}
```

Additional Features of Functions – Higher-order

Higher-order Functions

- Higher-order functions are a key concept in functional programming and are used to write more concise and expressive code by abstracting and encapsulating common patterns of computation.

```
numbers <- list(1, 4, 9, 16)
```

```
square_roots <- lapply(numbers,  
  sqrt)
```

Additional Features of Functions – Overloading


Function Overloading (in some languages): In languages like C++ and Java, you can define multiple functions with the same name but different number of arguments. This is known as function overloading.

! Not available in R S3 !

```
my_func <- function(x){  
  x * 10  
}
```

```
my_func <- function(x, y){  
  x * y  
}
```

```
my_func(10)
```



With only one argument
in the function call the
first function is used

Function examples

<examples “Functions, loops and conditionals” in R>

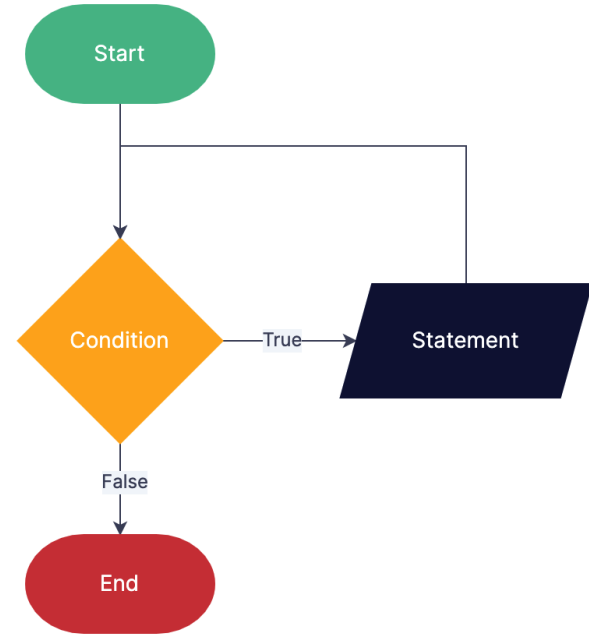
Discuss

Loops

For and while

Definition

A loop is a control structure that allows you to execute a block of code repeatedly based on a certain condition or for a specified number of times.



How Does it Work?

There are typically two main types of loops in programming:

- Conditional Loops (or "while" loops): These loops continue to execute a block of code as long as a specified condition remains true. The loop checks the condition before each iteration, and if it evaluates to true, the loop body is executed.
- Counted Loops (or "for" loops): These loops are used when you know beforehand how many times you want to execute a block of code. They typically iterate over a range or a collection of items, such as an array or a list.

Why are Loops Used?

Loops are fundamental for automating repetitive tasks and iterating over data structures, such as searching and sorting. They enable you to write efficient and concise code by reducing the need for redundant statements.

For Loop Anatomy – Variable

Variable

- The for loop is a control variable that keeps track of the current position in the sequence during each iteration.

```
for (i in 1:10) {  
  print(i)  
}
```

```
# Alternative  
for (i in my_vector) {  
  print(i)  
}
```

For Loop Anatomy – Sequence

Sequence

- The sequence represents the collection of values or elements over which the loop iterates. This could be a range of numbers, a list, an array, or any iterable data structure.

```
for (i in 1:10) {  
    print(i)  
}
```

```
# Alternative  
for (i in my_vector) {  
    print(i)  
}
```

For Loop Anatomy – Loop Body

Loop Body

- The loop body is a block of code that gets executed during each iteration of the loop. It contains the specific actions or operations you want to perform on each item in the sequence.

```
for (i in 1:10) {  
  print(i)  
}
```

```
# Alternative  
for (i in my_vector) {  
  print(i)  
}
```


For Loop Anatomy – Initialization

Initialization

- In a for loop, you initialize a loop variable to an initial value. This is usually done as part of the loop header.

```
for (i in 1:10) {  
  print(i)  
}
```

```
# Alternative  
for (i in my_vector) {  
  print(i)  
}
```

For Loop Anatomy – Termination

Termination Condition

- The loop continues to execute as long as the termination condition is met. The condition is checked before each iteration. If the condition evaluates to False, the loop terminates.

```
for (i in 1:10) {  
  print(i)  
}
```

```
# Alternative  
for (i in my_vector) {  
  print(i)  
}
```

While Loop Anatomy

```
i = 1  
while(i < 10) {  
  print(i)  
  i <- i + 1  
}
```

While Loop Anatomy – Initialization

Initialization

- This is where you initialize the variables or conditions needed for the loop to operate. You set the initial state that the loop will start with.

```
i = 1
while(i < 10) {
  print(i)
  i <- i + 1
}
```

While Loop Anatomy – Condition

Condition

- The loop's condition is an expression that is evaluated before each iteration of the loop. If the condition is true, the loop body is executed; if it's false, the loop terminates, and the program continues with the next statement after the loop.

```
i = 1
while(i < 10) {
  print(i)
  i <- i + 1
}
```

While Loop Anatomy – Loop Body

Loop Body

- This is the block of code that gets executed repeatedly as long as the condition remains true. It contains the statements or instructions you want to repeat.

```
i = 1
while(i < 10) {
  print(i)
  i <- i + 1
}
```

Control Statements

Break

- Terminates the loop prematurely, often based on a certain condition.

```
data <- list(1, 2, 3)

for (i in data) {
  if (is.null(dim(i))) {
    break
  }
  # do something
}
```

Control Statements

Return

- In some languages, you can use return to exit a function (and, by extension, any loops within that function).

```
my_func <- function(x, y) {  
  for (i in x) {  
    if (i == y) {  
      return(y)  
    }  
  }  
  return(-1)  
}
```


Control Statements

Next

- Skips the current iteration of the loop and proceeds to the next iteration.

```
data <- list
```

```
for (i in data) {  
  if (class(i) == "character") {  
    next  
  }  
  # do something  
}
```

Loop examples

<examples “Functions, loops and conditionals” in R>

Discuss

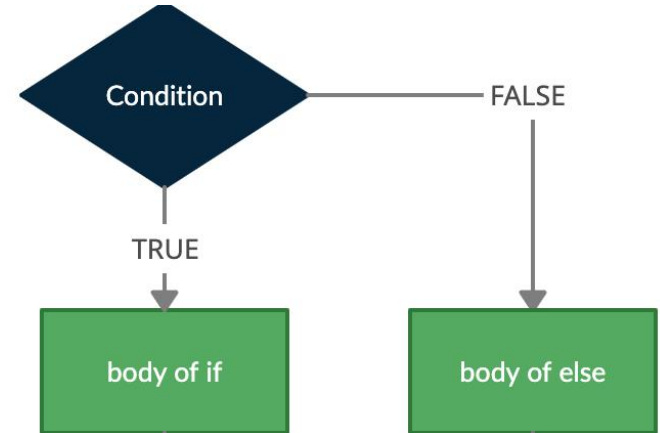
Conditionals

If... else...

Definition

A "conditional statement" or "conditional expression," is a fundamental construct that allows you to make decisions based on certain conditions.

It enables your program to execute different sets of instructions or take different paths of execution depending on whether a specified condition is true or false



How Does it Work?

Conditional statements typically take the form of "if-else" or "if-else if-else" structures and are used to control the flow of a program.

Common conditional operators used in conditionals include:

- Equality (== or !=)
- Greater than (>) and less than (<)
- Greater than or equal to (>=) and less than or equal to (<=)
- Logical operators (e.g., & for "and" and | for "or")

Why are Conditionals Used?

Conditional statements are a fundamental building block for creating logic and decision-making capabilities in software, enabling programs to respond dynamically to different situations and user input.

Conditional Anatomy – Condition

Condition

- The condition is a Boolean expression or a logical test that evaluates to true or false. It is the basis for deciding what code to execute.

```
if (x < 5) {  
  y <- "less than"  
} else if (x > 5) {  
  y <- "more than"  
} else {  
  y <- "equal"  
}
```


Conditional Anatomy – if Keyword

if Keyword

- The if keyword is the starting point of a conditional statement. It is followed by the condition enclosed in parentheses. If the condition evaluates to true, the code block following the if statement is executed.

```
if (x < 5) {  
  y <- "less than"  
} else if (x > 5) {  
  y <- "more than"  
} else {  
  y <- "equal"  
}
```

Conditional Anatomy – True Block

True Block

- The true block, also known as the "if block," contains the code that is executed if the condition specified in the if statement is true. This block is enclosed in curly braces ({}) to indicate the scope of the code that belongs to the if statement.

```
if (x < 5) {  
  y <- "less than"  
} else if (x > 5) {  
  y <- "more than"  
} else {  
  y <- "equal"  
}
```

Conditional Anatomy – else Keyword

else Keyword (optional)

- The else keyword allows you to specify an alternative block of code to execute when the condition in the if statement evaluates to false. If you don't provide an else block, the program continues execution after the if statement.

```
if (x < 5) {  
  y <- "less than"  
} else if (x > 5) {  
  y <- "more than"  
} else {  
  y <- "equal"  
}
```

Conditional Anatomy – False Block

False Block (optional)

- The false block, also known as the "else block," contains the code that is executed if the condition in the if statement is false. Like the true block, it is indented or enclosed in curly braces ({}).

```
if (x < 5) {  
  y <- "less than"  
} else if (x > 5) {  
  y <- "more than"  
} else {  
  y <- "equal"  
}
```

Additional Features of Conditionals

Multiple Conditions

- You can combine multiple conditions using logical operators (&&, ||) and parentheses to create more complex conditional expressions.

```
if (BMI < 30 && BMI > 18.5) {  
  x <- "no health check"  
} else {  
  x <- "health check"  
}
```

Additional Features of Conditionals

Guard Clauses

- You can use guard clauses or early returns to handle special cases at the beginning of a function or block of code before entering the main logic.

```
my_func <- function (x) {  
  if (x == specialCase) {  
    return(something)  
  }  
  # Main logic  
}
```

Additional Features of Conditionals

Default Values

- You can often provide default values in conjunction with conditional statements. If a condition is not met, the default value is used instead

```
if (x == regularCase) {  
    x <- y  
} else {  
    x <- -1  
}
```

Additional Features of Conditionals

Error Handling

- In addition to conditional statements for regular program flow, error-handling constructs like try-catch or try-except are used to handle exceptional

```
tryCatch(  
  {# Try part},  
  error = function(cond)  
    {#Error part},  
  warning = function(cond)  
    {#Warning part},  
  finally = {  
  }  
)
```


Nesting

Functions, loops and conditionals allow nesting

- An example of nesting is when you put a function inside of another function

Conditionals examples

<examples “Functions, loops and conditionals” in R>

Discuss

Classes

Informal (S3) classes and formal (S4) classes

Definition

A class is a blueprint or template for creating objects (instances).
A class defines the structure and behavior of objects of a particular type.

How Does it Work?

Classes serve as a fundamental building block of object-oriented programming (OOP), a popular programming paradigm that emphasizes the modeling of real-world entities and their interactions.

A class defines a set of attributes (properties or fields) and methods (functions or procedures) that the objects created from the class will have.

Why are Classes Used?

Abstraction

Encapsulation

Reusability

Modularity

Inheritance

Polymorphism

Organization

Complexity management

Classes provide a way to abstract or model real-world entities, concepts, or components in a program. They allow you to define the essential characteristics (attributes) and behaviors (methods) of these entities, while hiding unnecessary details. Abstraction makes code more understandable and manageable.

Why are Classes Used?

Abstraction

Encapsulation

Reusability

Modularity

Inheritance

Polymorphism

Organization

Complexity management

Classes support encapsulation, which means bundling data (attributes) and the functions that operate on that data (methods) into a single unit. Encapsulation helps hide the internal state and implementation details of an object, exposing only what is necessary for interaction. This enhances code security and reduces the likelihood of unintended data modification.

Why are Classes Used?

Abstraction

Encapsulation

Reusability

Modularity

Inheritance

Polymorphism

Organization

Complexity management

Classes enable code reuse. Once you define a class, you can create multiple objects (instances) from it. This means you can use the same class definition in different parts of your program or in different programs altogether. Reusing classes promotes consistency and reduces the need to duplicate code.

Why are Classes Used?

Abstraction

Encapsulation

Reusability

Modularity

Inheritance

Polymorphism

Organization

Complexity management

Classes encourage modularity by organizing code into discrete, self-contained units. Each class represents a specific functionality or component of the program. This makes it easier to develop, maintain, and debug complex software systems.

Why are Classes Used?

Abstraction

Encapsulation

Reusability

Modularity

Inheritance

Polymorphism

Organization

Complexity management

Classes can inherit attributes and methods from other classes, forming a hierarchy of classes. Inheritance allows you to create new classes based on existing ones, inheriting their characteristics and behaviors. This promotes code reuse and facilitates the modeling of relationships between objects.

Why are Classes Used?

Abstraction

Encapsulation

Reusability

Modularity

Inheritance

Polymorphism

Organization

Complexity management

Polymorphism enables objects of different classes to be treated as instances of a common superclass. This flexibility allows you to write code that can work with various objects without knowing their specific types.

Polymorphism simplifies code and makes it adaptable to different scenarios.

Why are Classes Used?

Abstraction

Encapsulation

Reusability

Modularity

Inheritance

Polymorphism

Organization

Complexity management

Classes provide a structured way to organize code. They group related attributes and methods together, making it easier to locate and manage code that pertains to a specific part of the program.

Why are Classes Used?

Abstraction

Encapsulation

Reusability

Modularity

Inheritance

Polymorphism

Organization

Complexity management

Classes help manage the complexity of large software projects. By breaking down a system into smaller, manageable classes, developers can focus on implementing and testing individual components independently.

S3 Class Anatomy

Define class

- In a S3 class an object is determined by a character vector attribute named "class" or "setClass".

```
obj <- c(x = 3, y = 4)  
class(obj) <- "Point2D"
```

OR

```
setClass("Point2D", slots = c(x =  
"numeric", y = "numeric"))  
point <- new("Point2D", x = 3, y = 4)
```

S3 Class Anatomy

Instantiate S3 class

- In S3 you “instantiate” an object which is then converted to a class.
or instantiate a class after its creation.

```
obj <- c(x = 3, y = 4)  
class(obj) <- "Point2D"
```

OR

```
setClass("Point2D", slots = c(x =  
"numeric", y = "numeric"))  
point <- new("Point2D", x = 3, y = 4)
```


S3 Class Anatomy

Extend class

- You can extend the functionality of existing classes by defining methods for those classes.

```
distance.Point2D <- function(p1, p2) {  
  sqrt((p1$x - p2$x)^2 + (p1$y -  
    p2$y)^2)  
}
```

```
print.Point2D <- function(p) {  
  cat("Point(", p$x, ",", p$y, ")\n")  
}
```

More on classes

There is much more to learn regarding classes, such as:

- S4 classes
- R6 classes
- Method overloading
- Interfaces and abstract classes
- Class documentation
- Method inheritance
- etc...

Discuss

Recap

Q&A Session



**Karolinska
Institutet**