Functions

301113 Programming for Data Science

WESTERN SYDNEY
UNIVERSITY

School of Computer, Data and Mathematical Sciences

Week 5

Outline

Making our way to University

Earlier, we wrote an algorithm describing how we travel from home to University and discussed the difference between high level and low level programs.

A top down design of the algorithm could have been

```
walk_to(train)
travel_on_train_to_parramatta()
walk_to(shuttle_bus)
travel_on_the_WSU_shuttle_bus_to_campus()
walk_to(class)
```

- We have now broken the algorithm into smaller piece that we can work on separately, where each piece is a function.
- We have also used `walk_to()` repeatedly. We only have to define what `walk_to()` does once, but it can be used many times.

Motivation

We have mentioned that it is good programming practice to design programs using top down design.

Top down design requires that we first write the program at a high level, using abstract blocks, then we proceed to design the algorithm for each block (using top down design).

So far, we have written code using `for` and `if` constructs, variables and available functions.

To design code blocks for top down design, we must write our own functions.

Outline

What is a function?

A mathematical function is a construct that allows us to abstract a mathematical equation. For example, if we define:

$$f(x) = \frac{1}{2\pi} e^{-\lambda \|x-\mu\|}$$

then we can now use $f(x)$ instead of writing the equation.

A function used in programming languages is similar. It is a construct for abstracting a piece of code.

Functions should be defined to perform a certain task and have a name to reflect the task.

Pre-defined Functions

We have been using functions that were pre-defined for us. For example `plot`, `c`, `print`. There are many functions available in the R base and more in libraries.

Defining a function

We can create a function by using the function `function`, providing the arguments and the algorithm that uses the arguments. All functions can have multiple arguments, but have only one output.

```
timesTwo <- function(x) {
    return(2*x)
}
```

For the above function:

- It has one argument `x`
- The body of the function lies between the braces `{}` (containing `return(2*x)`).
- It returns the value `2*x`
- The function is assigned to the variable `timesTwo`

Defining and Calling Functions

We just showed how to define a function. Note that defining a function makes the function available to use, but it does not call the function.

We can check a function definition by printing the variable that holds the function.

```
print(timesTwo)

## function(x) {
##  return(2*x)
## }
```

To call the function, we provide it with parentheses and the required arguments.

```
timesTwo(14)
```

Prime Numbers

Example

Let's write a function to check if a number is prime.

- Make sure the function name represents the function.
- The algorithm should only depend on the function arguments.
- Think about the type of variable that should be returned.

Multiple Arguments

Functions can have as many arguments as needed.

```
outlierThreshold <- function(x1, x2, x3, x4, x5, x6, x7, x8, x9) {
    ## compute the outlier threshold
    Q3 = quantile(c(x1, x2, x3, x4, x5, x6, x7, x8, x9), probs = 0.75)
    iqr = IQR(c(x1, x2, x3, x4, x5, x6, x7, x8, x9))
    threshold = Q3 + 1.5*iqr
    return(threshold)
}
```

But if the variable list is too long, then it might be an indicator to split the function into two, or organise the data into better structures.

Multiple Arguments

Functions can have as many arguments as needed.

```
outlierThreshold <- function(x1, x2, x3, x4, x5, x6, x7, x8, x9) {
    ## compute the outlier threshold
    Q3 = quantile(c(x1, x2, x3, x4, x5, x6, x7, x8, x9), probs = 0.75)
    iqr = IQR(c(x1, x2, x3, x4, x5, x6, x7, x8, x9))
    threshold = Q3 + 1.5*iqr
    return(threshold)
}
```

But if the variable list is too long, then it might be an indicator to split the function into two, or organise the data into better structures.

Problem

Which data structures can we use to simplify the above function?

Named Arguments

When providing arguments to a function, we must provide them in the correct order, unless we use their names.

```
standardise <- function(x, mean, sd) {
    return((x - mean)/sd)
}

## arguments are not named, so order of arguments matters
standardise(1, 2, 3)
```

```
## [1] -0.3333333
```

```
## named arguments can be in any order
standardise(mean = 1, sd = 2, x = 3)
```

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

Using named arguments makes code more readable and so is good programming practice.

Default Arguments

Many algorithms have parameters that have a commonly used value, but are sometimes changed. R functions allow us to define default values for all arguments. If a default value is defined, then we have the option of providing a value when calling it, or using the default.

Default values can be provided when defining the function.

```
standardise <- function(x, mean = 0, sd = 1) {
    return((x - mean)/sd)
}

standardise(1, 2, 3)

## [1] -0.3333333

standardise(mean = 1, sd = 2, x = 3)

## [1] 1
```

Hello world, and you

Problem

Write a function that takes a name one argument and prints “Hello name!”, where name is the provided name, or prints “Hello world!” if no name is provided.

Returning multiple values

We stated that functions can take multiple arguments, but return only one variable.

If the variable is a compound variable (such as a vector, list or data frame), then we can store multiple values in it.

Returning multiple values

We stated that functions can take multiple arguments, but return only one variable.

If the variable is a compound variable (such as a vector, list or data frame), then we can store multiple values in it.

Problem

Write a function that returns the mean and standard deviation of a provided vector.

Benefits of using Functions

Writing functions may take extra thought, but it has many benefits:

- It allows us to break a program into tasks (top down design), which can then be assigned to different people to write.
- It reduces the duplicate code.
- Provides improved code readability by hiding the details.
- It allows us to locate the source of bugs more easily.

Variable Descriptions

It is good programming practice to include a short description of each function argument at the top of the function (including the units the variable is measured in, if required).

Variable Descriptions

It is good programming practice to include a short description of each function argument at the top of the function (including the units the variable is measured in, if required).

Problem

Write a function that compute BMI, when given height and weight, and include a variable description at the top of the function.

Checking Arguments

Many errors occur when running programs due to unexpected values being provided as arguments to a function.

It is common practice for a function to check its arguments at the start of the function before proceeding to the algorithm.

Checks can involve:

- the domain of the values (should the argument be positive?)
- the type of the variable (is a string being provided?)

Outline

Variable Scope

Variable scope refers to which variables can be seen at a given place in the algorithm.

Any variables created within the function are also removed when the function ends.

Think of the function as its own separate environment that is created and then destroyed when the functions ends. The only value that remains is the value that is returned by the function.

If we used the top down design paradigm, we can consider entering a function as dropping down a level; the upper layer still exists, but we have temporarily created a lower level to work in.

Values outside of the function are accessible, but it is poor programming practice to access variables outside of function.

Accessing Variables


```
f <- function(x) {
    x <- x + 1
    y <- y + 1
    return(x + y)
}

x = 5
y = 6
f(x)
```

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

```
print(x)
```

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

```
print(y)
```

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

Function Environments

We saw in the previous example that the function could read variable that were not part of the function, but it could not change them.

When a function is called a new environment is created and the argument variables are copied into that environment. The function can only change the variables within this environment, but it can read variables from its parent environment (from which it was called).

Once the function finishes, the environment is destroyed.

Function Environments

We saw in the previous example that the function could read variable that were not part of the function, but it could not change them.

When a function is called a new environment is created and the argument variables are copied into that environment. The function can only change the variables within this environment, but it can read variables from its parent environment (from which it was called).

Once the function finishes, the environment is destroyed.

Double arrow assignment

R does provide the double arrow assignment operator <<– which will change variables in parent environments. Using this a will lead to confusion, so it is best to avoid.

Function Environments

We saw in the previous example that the function could read variable that were not part of the function, but it could not change them.

When a function is called a new environment is created and the argument variables are copied into that environment. The function can only change the variables within this environment, but it can read variables from its parent environment (from which it was called).

Once the function finishes, the environment is destroyed.

Double arrow assignment

R does provide the double arrow assignment operator <<– which will change variables in parent environments. Using this a will lead to confusion, so it is best to avoid.

Example

Run the previous example using the double arrow assignment operator.

Pure Functions

A pure function is a function that always produces the same output for a given input and has no side effects (it does not modify the system state, except for modifying the variable they assign their value to).

```
y <- 2
pure <- function(x) {
    x <- x^2 # a copy of x is used
    return(x)
}

impure <- function(x) {
    y <- y^2 + x # a copy of x and y are used
    return(y)
}
```

R protects from side effects by copying the variables when entering the function (copy on modify).

Problem

Show that the function `impure` is impure.

Outline

Functions of Functions

Functions can be called at any point of the program and so can also be called within functions.

```
outlierThreshold <- function(x) {
    return(quantile(x, 0.75) + 1.5*IQR(x))
}

detectOutlier <- function(x) {
    pos <- which(x > outlierThreshold(x))
    return(x[pos])
}

finishTimes <- c(0.9, 0.8, 0.9, 1.2, 1.1, 1.0, 12.1)
detectOutlier(finishTimes)

## [1] 12.1
```

Passing arguments to internal functions

There will be times when we want to pass arguments to a function within a function that we have written. R allows us to use the ellipsis keyword to accomplish this.

```
plotBirths <- function(births, ...) {
    n <- length(births)
    months <- month.abb[1:n]
    ## using ... to pass additional arguments to plot()
    plot(1:n, births, xaxt="n", xlab = "Months", ylab = "Births", ...)
    axis(1, at=1:n, label=months)
}

births <- c(12, 30, 23, 14, 15)
plotBirths(births, type = "b", col = 2, main = "Births per Month")
```

Example

Let's change the additional parameters to see the effect on the plot.

Recursive Functions

Recursive functions call themselves. This would cause infinite function calls unless there is a statement to stop the recursion.

A commonly used example is computing the factorial of a number.

```
factorial <- function(x) {
    if (x == 1) {
        return(1)
    }
    return(x * factorial(x-1))
}

factorial(6)
```

```
## [1] 720
```

Recursive Functions

Recursive functions call themselves. This would cause infinite function calls unless there is a statement to stop the recursion.

A commonly used example is computing the factorial of a number.

```
factorial <- function(x) {
    if (x == 1) {
        return(1)
    }
    return(x * factorial(x-1))
}

factorial(6)

## [1] 720
```

Problem

Which values of the argument x cause problems with this function?

Functions as Arguments

Functions are assigned to variables, and so can be passed as arguments to functions. Passing functions as arguments allows us to change the function algorithm as the code is running. Functions that take functions as arguments are called functionals.

```
prettyPrintStat <- function(x, statName, statFunc) {
    stat <- statFunc(x)
    xText <- paste(x, collapse = " ")
    text <- paste("The", statName, "of", xText, "is", stat)
    return(text)
}

runningTimes = c(10, 9, 2, 12, 8)
prettyPrintStat(runningTimes, "mean", mean)

## [1] "The mean of 10 9 2 12 8 is 8.2"

prettyPrintStat(runningTimes, "standard deviation", sd)

## [1] "The standard deviation of 10 9 2 12 8 is 3.76828873628335"
```

Returning Functions

Functions can return (usually more refined) functions called closures.

```
add <- function(x) {
    f <- function(y) {
        return(x + y)
    }
    return(f)
}

add2 <- add(2)
add2(5)
```

```
## [1] 7
```

```
add2(4)
```

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

Lazy Evaluation

The arguments to a functions in R are only evaluated if they are needed. This help to speed up evaluation, but can also be used strategically.

```
abuse <- function() {
    print("You stink!")
}

avoidAbuse <- function(x, f) {
    return(x^2)
}

# the function 'abuse' is not called, due to it not being used in the function
avoidAbuse(4, abuse)

## [1] 16
```

Outline

All actions are functions

Every action that we perform in R is calling a function. Some might not look like function calls due to the difference of syntax.

So all actions can be used for functional programming.

Problem

What do these functions commonly look like?

```
`+`(2,3)
`for`(a, 1:5, print(a^2))
`[`(v, 2)
```

Functional Programming

Functional programming is a type of programming where programs composed of applying and composing functions.

Functional programs can be written in R by organising data into appropriate structures and applying functions over the data.

Functions can be applied to:

- lists using `lapply` and `sapply`
- matrices using `apply`
- to tables using `tapply` and `mapply`

First class functions

R functions are known as first class functions, meaning they are treated just like variables.

Applying a function to elements of a list

Rather than using a `for` loop to apply a function to the elements of a list one by one, the function `lapply` (list apply) can be used. `lapply` takes the list as its first argument and a function as its second argument. The function is applied to each elements of the list

```
people <- list(
    list(name = "Fred", age = 43),
    list(name = "Lucy", age = 51),
    list(name = "Alex", age = 33)
    )

extractName <- function(x) {
    return(x$name)
}

## Extract the variable "name" from each element of a list
names <- lapply(people, extractName)
```

In this example, each element of the list is used as an argument of `extractName`.

Applying a function to a data frame

Data frames are a type of list, so `lapply` can be used on a data frames columns.

```
standardise <- function(x) {
    m <- mean(x)
    s <- sd(x)
    return((x - m)/s)
}

X <- iris[,1:4]
X[] <- lapply(X, standardise)
```

Subsetting to preserve type

The function `lapply` returns a list, but if it is assigned to a subset of a data frame, it will be coerced to the data frame type, even if replacing the whole data frame using `[]`.

Applying functions with multiple arguments

`mapply` is similar to `lapply` except that it can take multiple lists to apply to a multi-argument function.

```
people <- data.frame(
    weight = c(92.5, 67.8, 72.1, 79.1), # measured in kg
    height = c(1.64, 1.72, 1.77, 1.81) # measured in m
)

bmi <- function(weight, height) {
    return(weight/height^2)
}

people$bmi <- mapply(bmi, people$weight, people$height)
```

Applying functions to matrices

Matrices have rows and columns. The function `apply` allows us to apply a function to either the rows or the columns. The direction is given by the argument `MARGIN` (1 for rows, 2 for columns).

```
A <- matrix(1:6, 2, 3)
print(A)
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
norm <- function(x) {
    return(sqrt(sum(x^2)))
}
apply(A, 1, norm)
```

```
## [1] 5.916080 7.483315
```

```
apply(A, 2, norm)
```

```
## [1] 2.236068 5.000000 7.810250
```

Changing BMI Units

Problem

We are given the data in kilograms and centimetres units instead of kilograms and metres. How do we change the previous code to handle the new data?

```
people <- data.frame(
    weight = c(92.5, 67.8, 72.1, 79.1), # measured in kg
    height = c(164, 172, 177, 181) # measured in cm
)
```

Applying a function based on an index

`tapply` applies the provided function to subsets of a provided vector, where the subsets are defined by an index vector.

```
results <- data.frame(
    unit = c(301110, 301110, 301111, 301111, 301112, 301112, 301112),
    mark = c(78, 67, 87, 55, 65, 78, 92)
)
tapply(results$mark, results$unit, mean)
```

```
##   301110   301111   301112
## 72.50000 71.00000 78.33333
```

We used the predefined function `mean`, but we could use any function to summarise the categories.

More useful functional programming functions

R provides a set of functions that are commonly used for functional programming.

- `Map`: apply a function to the elements.
- `Reduce`: reduce a set of objects to one using a binary function.
- `Filter`: extract the elements that satisfies a logical function.
- `Find`: return the first object that satisfies a logical function.
- `Position`: return the position of the first object that satisfies a logical function.
- `Negate`: returns the logical negation of the function.

Map-Reduce

Google's Map-Reduce framework is named after the Map and Reduce functions.

More useful functional programming functions

R provides a set of functions that are commonly used for functional programming.

- `Map`: apply a function to the elements.
- `Reduce`: reduce a set of objects to one using a binary function.
- `Filter`: extract the elements that satisfies a logical function.
- `Find`: return the first object that satisfies a logical function.
- `Position`: return the position of the first object that satisfies a logical function.
- `Negate`: returns the logical negation of the function.

Map-Reduce

Google's Map-Reduce framework is named after the Map and Reduce functions.

Exercise

Let's compute the sum of squares of `c(1,2,3,4,5)` using `Map` and `Reduce`

Anonymous Functions

We have seen that functions are assigned to variables and then called to run. The variable contains the function for later use, but there might be functions that we want to run but not preserve in a variable. These are called anonymous functions.

Anonymous functions are useful when applying a one off function.

```
people <- list(
    list(name = "Fred", age = 43),
    list(name = "Lucy", age = 51),
    list(name = "Alex", age = 33)
    )

## Extract the variable "name" from each element of a list
names <- lapply(people, function(x) { return(x$name) })
```

Summary

- Functions allow code abstraction, making code more readable, and likely leading to few errors.
- Functions can take many inputs called arguments, but provide one output.
- Functions can only modify their own environment, but can access parent environments.
- R treats functions as variables, so they can be passed to and returned from functions.
- R programs can be written using a functional programming style.