# Week 2 Module 1: Basic Arithmetic

CSCI E-5a: Programming in R

Let's clear the global computing environment:

```
rm( list = ls() )
```

## Module Overview and Learning Outcomes

In this module, we'll focus on the basics of arithmetic in R.

- In Section 1, we'll define the concept of a number in R.
- In Section 2, we'll learn about fundamental arithmetic operations in R.
- In Section 3, we'll review the concept of operator precedence.
- In Section 4, we'll encounter some special values.

When you've completed this module, you'll be able to:

- Define the concepts of integers and floating point numbers.
- Define scientific notation in R.
- Perform all basic arithmetic operations in R.
- Explain the concept of operator precedence.
- Explain the four special values `Inf`, `-Inf`, `NaN`, and `NULL`.

We won't meet any new built-in R functions in this module, but we will encounter all the standard arithmetic operations.

All right! Let's get started by defining the concept of a number in R.

## Section 1: Numbers

> **Main Idea:** *We can define the concept of a number in R*

In this section, we'll define the concept of a number in R.

First, let's consider a very basic question – what is a numeric value in R?

That is, what is a number?

In general, in programming we usually work with two types of numbers:

- Integers, which are whole numbers such as 5, 0, or -17.
- Floating point numbers, which are numbers that have a decimal point and an unlimited number of decimal places after the decimal point, such as 3.1, 0.00, and -17.148372.

In many programming languages, the programmer must exercise care to distinguish between these two types of numbers, and at the physical level of the central processing unit (CPU) they are represented in different ways.

In R, we usually don't have to worry about this distinction, and essentially we're always working with floating point numbers.

If you have prior programming experience, this might seem a little unusual, and this is one way in which R is different from many other languages.

There's actually another way to represent floating point numbers, called *scientific notation*.

In scientific notation, a number is written as a floating point decimal number between 1 and 10, and then the value of 10 raised to some whole number power.

For instance, we can write the number 27.836 in this form:

$$27.836 \;=\; 2.7836 \times 10^{1}$$

In R, we write this using a special notation using the character `e`:

```
2.7836e1
```

```
[1] 27.836
```

The 'e' character represents the value of 10 raised to a power, which must be an integer (i.e. a whole number) but can be 0:

```
2.7863e0
```

```
[1] 2.7863
```

In fact, the exponent can even be a negative number:

```
2.7863e-1
```

```
[1] 0.27863
```

Don't get this "e" that's used in scientific notation confused with the number $e = 2.718282$:

```
exp( 1 )
```

```
[1] 2.718282
```

Scientific notation is very useful when we have very large or very small numbers.

For instance, the mean distance from the sun to the planet Neptune is:

$$4,500,000,000,000 \text{ m} \;=\; 4.5 \times 10^{12}\text{m}$$

We can represent this number by explicitly writing out all the 0s:

```
4500000000000
```

```
[1] 4.5e+12
```

However, this is awkward and easy to get wrong, and the scientific notation is more compact and simpler:

```
4.5e12
```

```
[1] 4.5e+12
```

Notice that R will generate an error if you try to help yourself by using commas:

```
4,500,000,000,000
```

```
Error: unexpected ',' in "4,"
```

By the way, notice that expression `error = TRUE`.

That's called a "chunk option", and it tells RStudio that if an error is generated while knitting that RStudio should continue to knit.

If we didn't have the chunk option there, then when RStudio encounters the error it would stop knitting.

Another example is the quantity of electric charge on an electron, which is:

$$0.000000000000000000160218 \text{ C} \;=\; 1.60218 \times 10^{-19}\text{C}$$

Notice how awkward it is to have to work with the decimal expansion, but the scientific notation is easy to understand.

In fact, I'm still not sure that I typed in the decimal expansion correctly, but I know that the value in scientific notation is correct.

In R, this is:

```
1.60218e-19
```

```
[1] 1.60218e-19
```

For the most part, it's unlikely that you'll work with such extreme values, but if you do then it's useful to know about scientific notation.

So that's how we define the concept of a number in R.

Now let's see how to perform operations on numbers in R.

## Section 2: Arithmetic operations

**Main Idea:** *We can perform all standard arithmetic operations in R.*

In this section, we'll learn about fundamental arithmetic operations in R.

Once we have some R objects, it's natural to want to do something with them.

We can use one or more objects to compute a new value, for instance by adding them or multiplying them, and this is called an "operation".

If the operation takes just one value as its input, then it is called a "unary" operator, and if the operation takes two values as its inputs, it is called a "binary" operator.

R implements all the standard arithmetic operations that you know from elementary mathematics.

There is one unary arithmetic operation i.e. an operation that takes just one number as its input.

This operation is *negation*, denoted by `-`, which takes a single numeric value as its input and returns the negative of that value:

```
- 6
```

```
[1] -6
```

We can use this operator twice:

```
- - 6
```

```
[1] 6
```

There are many binary arithmetic operations, and they are represented using standard notation.

Addition is denoted by `+`:

```
2 + 2
```

```
[1] 4
```

Subtraction is denoted by `-`:

```
7 - 5
```

```
[1] 2
```

Multiplication is denoted by `*`:

```
4 * 9
```

```
[1] 36
```

The exponentiation operation, in which one number is raised to a given power, is denoted by `^`:

```
5^3
```

```
[1] 125
```

You can also use `**` to denote exponentiation:

```
5**3
```

```
[1] 125
```

When we get to division, we have to be careful, because there are different kinds of division operations.

The simplest is floating point division, which takes two numbers and returns a floating point number, this operation is denoted by `/`:

```
11 / 7
```

```
[1] 1.571429
```

This is the form of division that is used most often in practice.

R can also perform *integer* division, in which a number is represented as a quotient times a divisor and a remainder.

For instance, if we divide 17 by 5, then the divisor is 5, the quotient is 3, and the remainder is 2:

$$17 = 3 \times 5 + 2$$

R calculates the quotient for integer division by using the `%/%` operator:

```
17 %/% 5
```

```
[1] 3
```

The remainder is calculated using the `%%` operator:

```
17 %% 5
```

```
[1] 2
```

If you want a complete listing of the various operators in R, you can go to the Help system and search for "Syntax" (make sure you use a capital 'S').

So those are the fundamental arithmetic operations in R.

Now let's explore the concept of operator precedence.

## Exercise 1.1: Weird Arithmetic

Consider this method for determining a number:

- We start with the value 17, then we multiply it by 11.
- We then raise the resulting quantity to the third power.
- We then add 1,653 to this value.

Determine if the final value is evenly divisible by 7.

**Solution**

```
# Type your solution here.
```

## Section 3: Operator precedence

> **Main Idea:** *We must be aware of operator precedence issues.*

In this section, we'll review the concept of operator precedence.

Once we start to write complicated expressions containing multiple operations, we have to think about the question of operator *precedence*, which is simply the ordering of which operation to apply first.

For example, suppose we encounter this expression:

$$2 + 3 \times 4$$

How should we interpret this?

There are two plausible approaches:

- In the first approach, we first add 2 and 3, obtaining 5, and then we multiply by 4, giving a final result of 20.
- In the second approach, we first multiply 3 by 4, obtaining 12, and then we add 2, giving a final result of 14.

So depending on what order we perform the addition and multiplication operations, we end up with different values.

What does R do? Let's find out . . .

```
2 + 3 * 4
```

```
[1] 14
```

So it seems that R uses the second approach.

We can think of this as that both the addition operator `+` and the multiplication operator `*` are competing for the value 3, and the multiplication operator wins out.

We call this "operator precedence", in the sense that the multiplication operation has higher precedence than the addition operation.

Alternatively, sometimes people will say that the multiplication operator "binds more tightly" than the addition operator.

In general, there is a hierarchy of operator precedence relations:

- The exponentiation operation has the highest precedence.
- Next comes the unary minus operator.
- Next, multiplication and division have equal precedence, but are lower than the unary minus operator.
- Addition and subtraction have equal precedence, but are lower than multiplication and division.

We can use parentheses to control the exact sequence of operations: expressions inside a set of parentheses are evaluated first, and then used in later computations.

For instance, if we really want to add 2 to 3 first, and then multiply by 4, we can write:

$$(2 + 3) \times 4$$

In the US, a common mnemonic device for this hierarchy is PEMDAS, which stands for Parentheses, Exponentiation, Multiplication, Division, Addition, and Subtraction.

Here's a simple rule for you to follow with operator precedence: don't write complicated expressions that can only be understood by careful application of the rules of precedence.

If it's so hard to work out, then it's difficult to be sure that the expression is really correct.

It can also be challenging for someone who is not familiar with the code as you are when you write it, and that could be another analyst – or even you in three months!

When in doubt, always use parentheses to make the order of operations absolutely clear, and don't be afraid to use 2 or 3 steps instead of trying to compress an expression into 1 step.

The goal is always correctness first, closely followed by readability, and you're not winning if you write a highly compressed and elegant expression that does the wrong calculation.

There's lots more to the issue of precedence, but this is the basic idea.

Again, if you want a comprehensive listing of all operators and their precedence levels, you can search for "Syntax" in the Help system.

So that's the concept of operator precedence.

Now let's investigate some special values.

## Exercise 2.2: Operator Precedence

I recently had to perform a calculation in which I took the three numbers -4, 2, and 10, squared them, multiplied these squared values by the weights 0.25, 0.25, and 0.50, and then added up the resulting values:

| Value | Squared Value | Weight | Squared Value × Weight |
|---|---|---|---|
| -4 | 16 | 0.25 | 4 |
| 2 | 4 | 0.25 | 1 |
| 10 | 100 | 0.50 | 50 |
| | | **Sum** | 55 |

I wrote this R code:

```
(-4^2 * 0.25) + (2^2 * 0.25) + (10^2 * 0.50)
```

Why did I get the wrong answer with this expression?

Then write R code that will perform the calculation correctly.

**Solution**

## Section 4: Special numeric values

**Main Idea:** *R has some special numeric values.*

In this section, we'll encounter some special numeric values.

There are four special numeric values in R: `Inf`, `-Inf`, `NaN`, and `NULL`.

### `Inf` and `-Inf`

The special numeric values `Inf` and `-Inf` represent positive and negative infinity, respectively.

R treats these special values as `numeric` data, even though properly speaking positive infinity and negative infinity are not really "numbers".

Instead, they refer to some sort of limiting process.

The most obvious way to generate an `Inf` value is to attempt to generate a number that is too large:

```
1000^1000
```

```
[1] Inf
```

Numerical operations involving `Inf` or `-Inf` often have an intuitive interpretation, and in this sense we can think of `Inf` and `-Inf` as "numbers".

For instance, if we add `Inf` to a finite number, we end up with `Inf`:

```
2 + Inf
```

```
[1] Inf
```

If we subtract `Inf` from a finite number, the result is `-Inf`:

```
14 - Inf
```

```
[1] -Inf
```

If we divide a finite number by `Inf`, we end up with 0:

```
17 / Inf
```

```
[1] 0
```

If we attempt to divide a finite nonzero number by 0, we obtain `Inf`:

```
8 / 0
```

```
[1] Inf
```

This last example is problematic.

Attempting to divide a finite number by 0 is actually an illegal operation, and this should generate an error.

R however just returns a numeric value, that is, the special numeric value `Inf`.

This isn't such a useful feature because if I've written code that performs a zero division then almost certainly that means that there is a mistake in my code, and the best way to fix that bug is for the computer to generate an error so I know about it.

Instead, by generating the special value `Inf`, R effectively hides this problem from me.

I'm not enthusiastic about this approach, but that's how R works, and we just have to accept it.

To be fair, occasionally there are legitimate uses of `Inf` and `-Inf`.

In our course, I can think of only one situation where we use these values.

But otherwise if you see an `Inf` or `-Inf` value, that's a strong indication that you made a mistake in your program.

## The special value `NaN`

The special numeric value `NaN` stands for "not a number" and is something very different from `Inf` and `-Inf`.

The class of `NaN` is numeric:

```
class( NaN )
```

```
[1] "numeric"
```

This is ironic, since the name `NaN` is an abbreviation for "Not a Number".

The point is that `NaN` arises when we attempt to perform illegal operations on numbers, and the classic example is when we try to divide 0 by 0:

```
0/0
```

```
[1] NaN
```

R can't interpret this, so it returns a `NaN`.

Another way that R generates a `NaN` is when attempting to take a fractional root of a negative number:

```
(-8)^(1/3)
```

```
[1] NaN
```

Also, if you try to take the logarithm of a negative number:

```
log( -3 )
```

```
Warning in log(-3) : NaNs produced
```

```
[1] NaN
```

Here's one last example: attempting to subtract `Inf` from `Inf`:

```
Inf - Inf
```

```
[1] NaN
```

I mentioned that there are times when the special values `Inf` and `-Inf` can serve a legitimate purpose, even though they are often the result of a coding error.

I'm usually reluctant to say the word "always", but I think if your output has `NaN` values in it, then that's a strong indication that there is a bug in your code, and you need to fix it.

### `NULL`

There's one last special value: `NULL`.

We've already used this value – can you remember where?

When we create an empty plot with no data, we have to set the `x` input argument to `NULL`.

The R documentation says that "`NULL` represents the null object in R.", which, while no doubt true, is not particularly helpful.

You can think of `NULL` as representing the concept of nothing.

That's why it makes sense to use it when we're creating an empty plot with no data, because we're not graphing any data.

This is starting to get metaphysical, and to be honest we won't encounter `NULL` very often, so I think the best strategy is to be aware of it but not worry about it if you don't have to.

However, if it starts showing up in your output, that's probably a sign that there's a serious bug in your code.

So those are some special values in R that you need to know.

Now let's review what we've learned in this module.

## Module Review

In this module, we focused on the basics of arithmetic in R.

- In Section 1, we defined the concept of a number in R.
- In Section 2, we learned about fundamental arithmetic operations in R.
- In Section 3, we reviewed the concept of operator precedence.
- In Section 4, we encountered some special values.

Now that you've completed this module, you should be able to:

- Define the concepts of integers and floating point numbers.
- Define scientific notation in R.
- Perform all basic arithmetic operations in R.
- Explain the concept of operator precedence.

- Explain the four special values `Inf`, `-Inf`, `NaN`, and `NULL`.

We didn't meet any new built-in R functions in this module, but we did encounter all the standard arithmetic operations.

All right! That's it for Module 1: Arithmetic.

Now let's move on to Module 2: Variables.

# Solutions to the Exercises

## Exercise 1.1: Weird Arithmetic

Consider this method for determining a number:

- We start with the value 17, then we multiply it by 11.
- We then raise the resulting quantity to the third power.
- We then add 1,653 to this value.

Determine if the final value is evenly divisible by 7.

**Solution**

I will perform all the arithmetic operations to construct the final number, and then determine whether or not it is divisible by 7 by examinging the remainder when using integer division with a divisor of 7.

```
( (17 * 11)^3 + 1653) %% 7
```

Since the remainder is 0, we conclude that the final number is evenly divisible by 7. To check this, let's first calculate the final number:

```
(17 * 11)^3 + 1653
```

Let's perform integer division on this value:

```
6540856 %/% 7
```

Now let's multiply this value by 7:

```
934408 * 7
```

## Exercise 2.2: Operator Precedence

I recently had to perform a calculation in which I took the three numbers -4, 2, and 10, squared them, multiplied these squared values by the weights 0.25, 0.25, and 0.50, and then added up the resulting values:

| Value | Squared Value | Weight | Squared Value × Weight |
|---|---|---|---|
| -4 | 16 | 0.25 | 4 |
| 2 | 4 | 0.25 | 1 |

| Value | Squared Value | Weight | Squared Value × Weight |
|---|---|---|---|
| 10 | 100 | 0.50 | 50 |
| | | **Sum** | 55 |

I wrote this R code:

```
(-4^2 * 0.25) + (2^2 * 0.25) + (10^2 * 0.50)
```

Why did I get the wrong answer with this expression?

Then write R code that will perform the calculation correctly.

**Solution**

```
((-4)^2 * 0.25) + (2^2 * 0.25) + (10^2 * 0.50)
```