Week 7 Module 2: R Functions

CSCI 5a: Programming in R

Let's clear the global computing environment:

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

Let's load in the R objects:

```
load( "Module 2 R Objects.Rdata" )

ls()
```

```
## [1] "example.1.data"
```

Module Overview and Learning Outcomes

Hello! And welcome to Module 2: R Functions.
In this module, we'll take a look at some important technical issues with R functions.

- In Section 1 we'll see two different way of assigning input values when we call a function.
- In Section 2 we'll learn a little more about default values for optional parameters.

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

- Explain the two different methods of assigning input values.
- Explain the concepts of optional parameters and default values.

There are no new built-in R functions in this module.
Let's get rolling!

Section 1: Specifying Input Argument Values

> **Main Idea:** *There are two different ways to specify the input arguments for a function*

In this section, we'll see two different way of assigning input values when we call a function.
There are essentially two different ways to specify values for a function's input arguments:

- We can assign a value by explicitly using the name of the input argument.
- We can assign a value by using the order of the arguments.

Specify input arguments by input argument name

Let's start by using the `rep()` function to create a vector consisting of the value 7, repeated 3 times.

```
rep( x = 7, times = 3)
```

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

This function takes two arguments: the number 7, which specifies what is to be repeated, and the number 3, which specifies the number of repetitions.

How does R know that we want to repeat the object 7?

Because we told it: by binding the value 7 to the input argument `x`, we indicated that 7 was the object that was supposed to be repeated.

Similarly, we indicated that we wanted to repeat the object 3 times by binding the value 3 to the input argument `times`.

When we bind values to input arguments using the `=` operation, we don't have to worry about the specific ordering of the arguments.

For instance, we could have written this:

```
rep( times = 3, x = 7 )
```

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

As long as we're binding values to input arguments by explicitly using the name of the input argument, it doesn't matter what order in which the arguments are specified.

Exercise 1: Input argument re-orderings

Generate a vector consisting of the number 8 repeated 5 times, using two different orderings of the input arguments.

Solution

Specifying input arguments by position

We don't have to explicitly name the arguments.

Instead, we can simply use the position of the input arguments.

For the `rep()` function, if the input argument names have **not* been specified, then R always interprets the first argument as the `x` input argument, and the second argument as the `times` input argument.

Thus, to repeat the value 7 for a total of 3 times, we have:

```
rep( 7, 3 )
```

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

Here the first argument is the value 7, so that's the argument that will be repeated.

The second argument is 3, so that's the number of repetitions.

When we explicitly indicated the names of the input arguments, then it didn't matter what order we specified them in.

But when we don't use the names, and just rely on the ordering, then the order of the values matters a great deal!

For instance, look what happens when we change the order of the previous command:

```
rep( 3, 7 )
```

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

This is very different from what we saw before!

Now, we have the number 3, repeated 7 times, and of course this is very different from the number 7, repeated 3 times.

You can generate strange errors by being insensitive to the ordering.

For instance, suppose we want to repeat the character string "hedgehog" 3 times.

If we don't want to explicitly name the input arguments, we have to write this:

```
rep( "hedgehog", 3 )
```

```
## [1] "hedgehog" "hedgehog" "hedgehog"
```

Now look what happens if we try to reverse the order of these two arguments:

```
rep( 3, "hedgehog" )
```

```
## Warning: NAs introduced by coercion
```

```
## Error in rep(3, "hedgehog"): invalid 'times' argument
```

This error message is telling us that the `times` argument is invalid.

Why is that?

The `times` argument **has** to be numeric, because it is specifying the number of times to repeat the object.

But the way that we have written the function call, the character string "hedgehog" is what is being associated with the `times` argument, and this can't be interpreted as a number.

So R issues an error, stating that we have an "invalid `times` argument".

Of course, if we had used names for the input arguments, we would not have to worry about the order:

```
rep( times = 3, x = "hedgehog" )
```

```
## [1] "hedgehog" "hedgehog" "hedgehog"
```

Traditionally, in most programming languages, all functions are specified using a standard order, and the technique of naming arguments is not used.

Personally, I like to always explicitly name arguments, because I think this makes the code much clearer.

Also, I might forget the order of the input arguments.

For instance, there is another function named `replicate()` that is very similar to `rep()` – but the first input argument for `replicate()` is the number of replications, not the object to be replicated.

Conversely, the second argument to `replicate()` is the object to be repeated, not the number of repetitions:

```
replicate(3, 8)
```

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

Ugh! `replicate()` can be a very useful function, but it's very unfortunate that the order of input arguments for `replicate()` is different from the closely related `rep()` function.

The real issue here isn't that so much that one ordering of the input arguments is better than the other; it's that the two functions don't agree on one convention.

This is a very important point: it's really important to have standards and conventions that are consistently implemented, both in your own code as well as across a project team or even an enterprise.

Recomendations

Personally, I think it's a good idea to specify the input arguments to your functions by explicitly assigning a value to a named input argument.

In other words, don't just type in the values of the function and rely on the order of the arguments.

By writing out the names of the options, you're documenting your code, and you're being clear about exactly what you're doing.

As the instructor for CSCI S-5a, I'm trying to model for you what I think are good practices in coding, and so I usually use the name of options.

You'll have to decide for your own practice whether or not to use this approach.

So those are the two different way of assigning input values when we call a function.

Now let's see how to create default values for input arguments.

Section 2: Default Values

> **Main Idea:** *We can specify default values for input arguments*

In this section, we'll learn a little more about default values for input arguments.

One feature of R functions is that an input argument can have a *default* value – that is, in the absence of an explicit specification the input argument will automatically be assigned a particular value.

Default values are very useful, because they enable us to work with functions that have many options without being required to explicitly specify each one.

We use default values a lot with graphics functions.

For instance, look at the help documentation for the `hist()` function.

What if I just want to make a simple histogram – do I really have to specify every single one of these parameters, most of which I don't really care about?

The answer is: no.

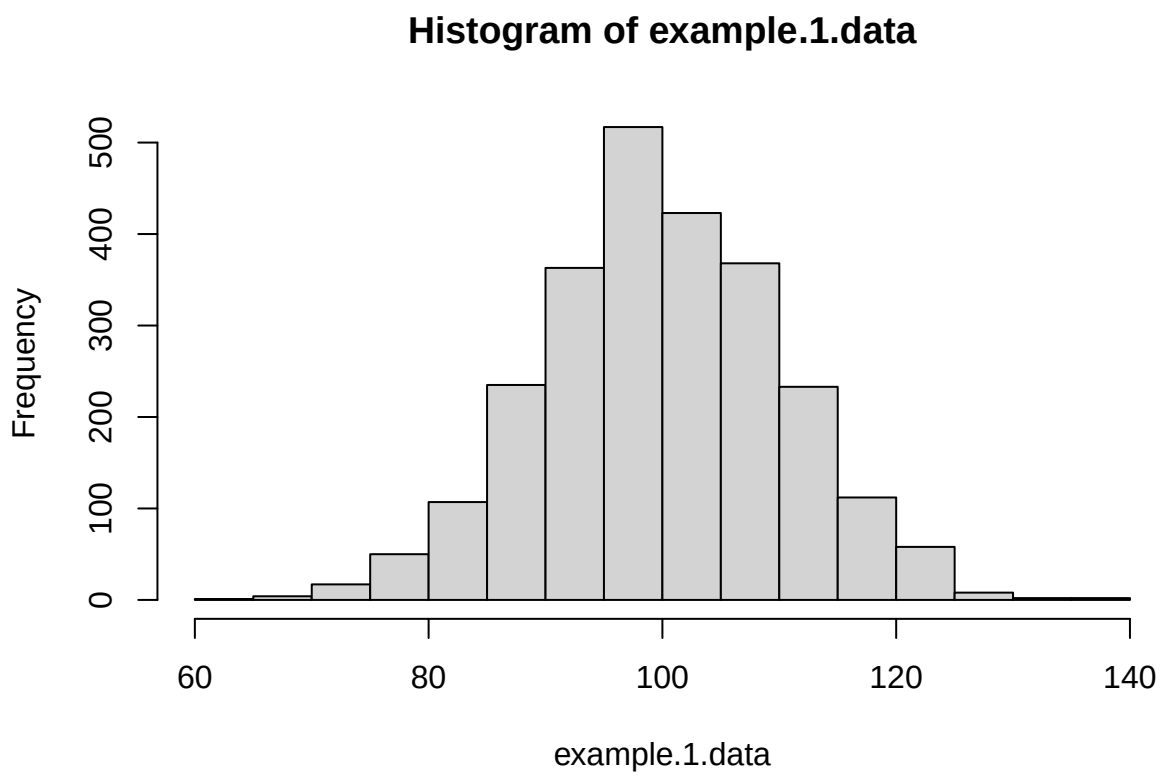
R will assign many parameters certain default values, so that you don't have to.

On the other hand, if you want to override one of the default values, you always can do that.

The default values are designed to provide a useful set of initial option values, and they do not attempt to make the graph look pleasing or attractive.

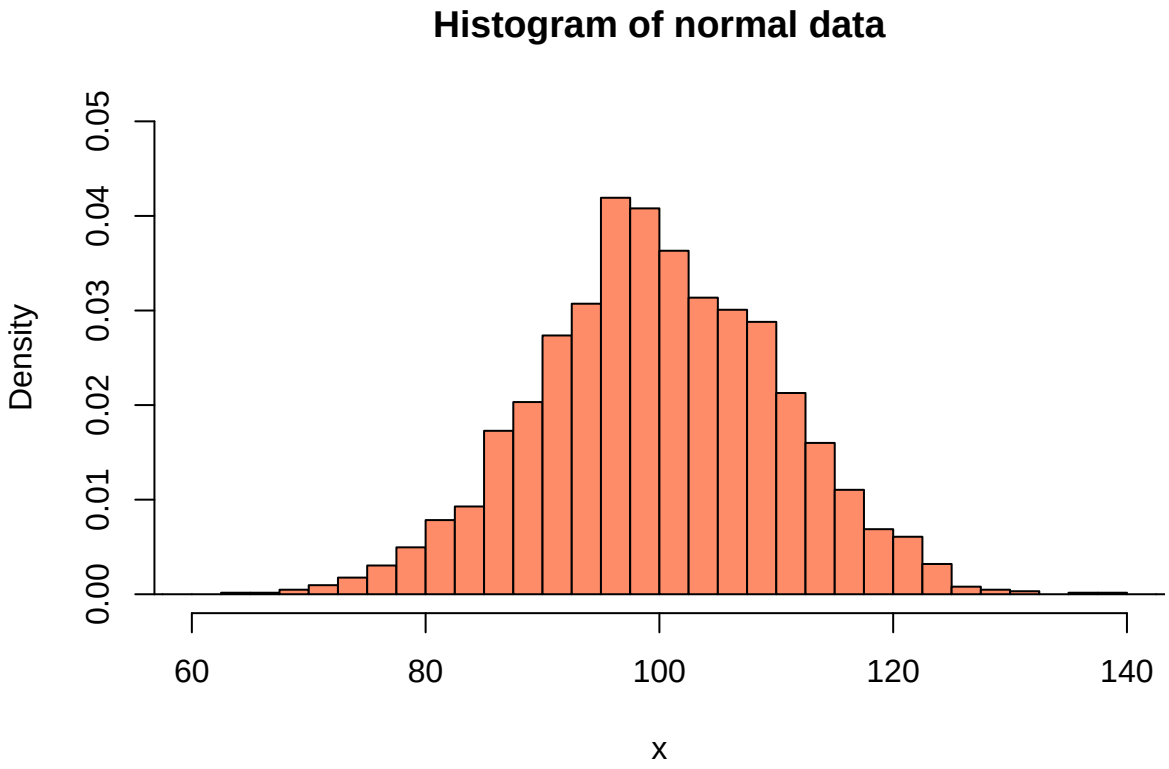
Here is the most basic form of the `hist()` function, using all default values:

```
hist( example.1.data )
```



That's not bad, but of course we can make a lot of improvements:

```
hist(
    example.1.data,
    xlim = c(60, 140),
    ylim = c(0, 0.05),
    prob = TRUE,
    main = "Histogram of normal data",
    xlab = "x",
    ylab = "Density",
    col = "salmon1",
    breaks = seq( from = 0, to = 1000, by = 2.5 )
)
```

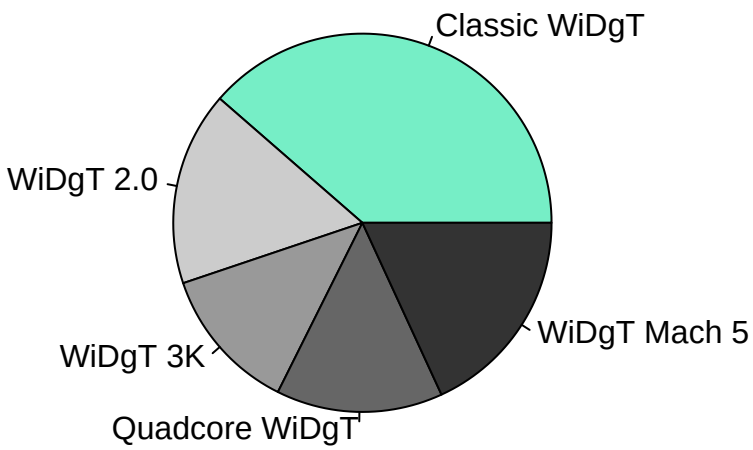


Sometimes default specifications can be very sophisticated.

Let's take a closer look at the `pie()` function, that draws a pie chart.

Let's make a simple pie chart, but without specifying clockwise rotation:

```
pie(
    x = c( 34.7, 14.9, 11.2, 12.8, 16.3),
    labels =
        c( "Classic WiDgT", "WiDgT 2.0", "WiDgT 3K",
        "Quadcore WiDgT", "WiDgT Mach 5" ),
     main = "Widget sales for 2018",
     col = c("aquamarine2", "gray80", "gray60",
             "gray40", "gray20")
)
```



When we specified the data for the pie chart, notice that the first item was for "Classic WiDgT", the second item was for the "WiDgT 2.0", the third item was for "WiDgT 3K", and so on.

If you look at the pie chart, the corresponding pie slices rotate in a *counter-clockwise* manner.

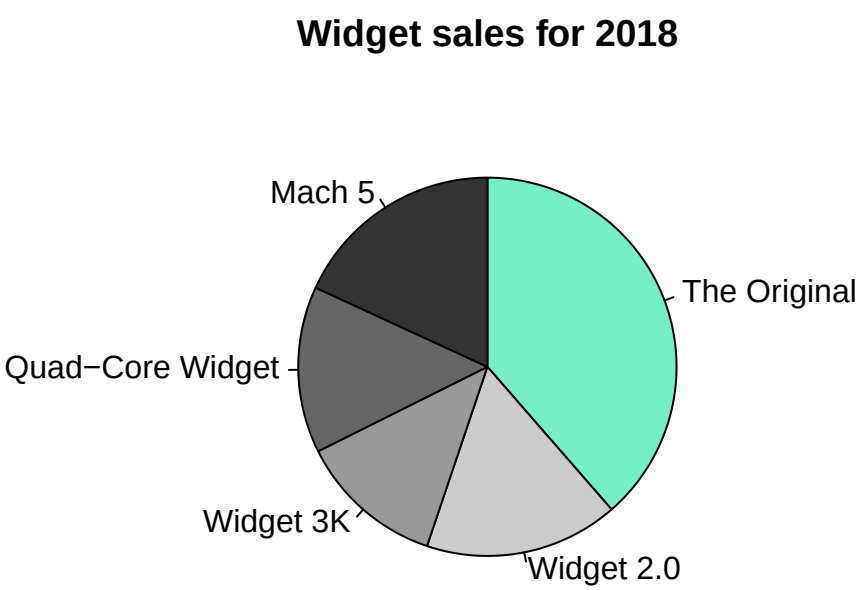
So the default rotation is counter-clockwise.

In fact, if you look at the documentation for the `pie()` function, you'll see that there is a parameter `clockwise` which has a default value of `FALSE`.

Let's check out the documentation for the `pie()` function to see this.

Now we're going to make the same graph, but this time we'll set the clockwise parameter to `TRUE`:

```
pie(
    x =
        c( 34.7, 14.9, 11.2, 12.8, 16.3),
    labels =
        c( "The Original", "Widget 2.0", "Widget 3K",
        "Quad-Core Widget", "Mach 5" ),
    clockwise = TRUE,
    main = "Widget sales for 2018",
    col = c("aquamarine2", "gray80", "gray60",
              "gray40", "gray20")
)
```



Now the pie slices go around the pie in a clockwise manner, because we explicitly overrode the default value.

There was something else going on: the pie slice started in a different position, depending on the value of the `clockwise` parameter.

This is determined by a formula in the default specification of the parameter `init.angle`:

- If the parameter `clockwise` is `FALSE`, then the first pie slice starts parallel to the horizontal axis.
- If the parameter `clockwise` is `TRUE`, then the first pie slice starts perpendicular to the horizontal axis.

Let's check out the documentation for the `pie()` function to see this.

Amazing! A default parameter can be specified depending on the value of another parameter.

This is fancy stuff, and I do **NOT** expect you to write code like this, or even encourage it!

Instead, I'm showing you this as a sophisticated example of creating default values for input arguments.

So that's how we can define default values for input arguments.

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

Module Review

In this module, we looked at some important technical issues with R functions.

- In Section 1 we saw two different ways of assigning input values when we call a function.

- In Section 2 we learned a little more about default values for optional parameters.

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

- Explain the two different methods of assigning input values.
- Explain the concepts of optional parameters and default values.

There were no new built-in R functions in this module.

So that's it for Module 2: R Functions.

Now let's move on to Module 3: Write Your Own Functions.

Solutions to the Exercises

Exercise 1: Input argument re-orderings

Generate a vector consisting of the number 8 repeated 5 times, using two different orderings of the input arguments.

Solution

The first sequence is:

```
rep( x = 8, times = 5 )
```

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

The second sequence is:

```
rep( times = 5, x = 8 )
```

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