A deeper dive into graphing in base R

Compiled on 11 June 2022 20:48.

Typical tutorials on graphing in R aim to rapidly develop understanding of higher level graphing functions to get users generating useful graphs quickly. While this approach has merit in certain use cases, it tends to gloss over the lower level understanding of detail on how high level graphing functions work. This lack of detail limits the creativity with which a user is able to create graphs not supported by the higher level functions or the specificity with which a user can make a graphical rendering look exactly as desired. Lack of detail also limits the skill with which a user can troubleshoot graphing code that does not render as expected.

This tutorial is designed to provide a deeper dive into graphing with base R functions. It first reviews the basics of graphics devices, then progresses to build scaffolding of examples with increasing levels of detail into how graphing is implemented in base R. While the tutorial attempts to build in complexity and detail one step at a time, some familiarity with the manipulation of fundamental data types and the operation of functions in R is probably necessary to take full advantage of this tutorial. Completing a higher level tutorial on graphing in base R before working through this more detailed tutorial may be an effective strategy.

The reader should also be prepared to experiment with an open session in R (perhaps starting with code copied from this document) to take full advantage of the tutorial. Challenging yourself to predict what a function will do if you change an argument, and then testing that prediction with your own code, is an effective way to self-assess complete understanding of these skills and principles presented here. Ongoing learning of the R functions that allow exploration of the data structures of arguments to R functions and the return values from R functions is highly recommended for rapid progress in R programming and debugging skill.

This tutorial focuses on graphics tools typically loaded by default in an R session, including the **grDevices** package and the **graphics** package. Not covered here are a family of tools based on the **grid** graphics package that is an alternative implementation of the **graphics** package and thus provides an alternative interface to creating graphical objects. Popular higher level graphing tools that use **grid** include the **lattice** and **ggplot2** packages. A full understanding of **grid** with the conveniences of **lattice** or **ggplot2** are another way to take full control over the appearance of complex data visualizations.

## 1. Graphics devices

Opening a graphics device creates a blank canvas upon which graphs will be rendered. The most common graphing tools in R (including base R, grid, and ggplot) all use the same canvas provided by a given graphics device. Therefore, if you understand the details of how to control the specific areas of the canvas that are used by a given graphing tool, you can combine or overlap the results of any or all of these tools on the same canvas.

Graphics devices are designed to have a common interface for the functions used to add symbols to the graphs. Ideally, calling a given graphing function should create the same rendered visualization regardless of which device is in use. This feature provides the modularity that a given block of graphing code should create the same image regardless of using a device that puts it on the screen, writing it to an raster image file, or writing it to a vector graphics format. This degree of modularity is seldom fully realized. Particular devices may not have all the features necessary to render all aspects of a given graphing tool. For example, identical fonts or font sizes may not be available across all graphics devices, and transparency in colors will not render properly on all devices. You should expect differences between how different devices plot the same graph, so early consideration of the appropriate device that generates the proper format for the final product is encouraged.

A default device (called the “null device”) is likely automatically open in R, where the device used as default depends upon the platform you are using for development. In RStudio, the default device typically renders to the “Plots” tab, which is typically available in the bottom right panel. The device that renders to the “Plots” tab is convenient, but the size and aspect ratio of graphs rendered in this panel will change depending on how your RStudio window is configured. This lack of consistency in size and aspect ratio is problematic when refining figures or sharing code that generates figures, and use of an external device that allows consistency through programmatic control over the dimensions of the figure is suggested.

The full array of available graphics devices cannot be demonstrated directly in an Rmarkdown file, because the compiler for R Markdown uses a specific device that embeds the resulting rendered graphics in the compiled document. However, a few examples of useful graphics devices are briefly outlined here, and experimentation with these devices is encouraged to understand the available options for figure development.

### 1.1. Writing graphs to postscript vector graphics

A useful device directly renders graphics to a file in vectorized postscript format (e.g., Adobe PDF files), and works on both Mac and PC computing platforms (as well as many others). The pdf() function is used to open this graphics device. Key arguments to the pdf() function allow direct programmatic control over the basic size and default font point size for a given canvas. The following code generates a sample PDF file, if run in an R session with the working directory set to where the file should be written. You are encouraged to cut and paste this code into an R script, and experiment with running the script with different values for the arguments to confirm you understand the behaviors they control. Note that the script may not run properly if you have the resulting graphics file open in a PDF viewer, if your PDF viewer locks the file.

# Shut down any existing devices  
graphics.off()  
  
# Open the PDF graphics device   
pdf(  
 file = "./graphics\_test.pdf",  
 width = 6,  
 height = 5,  
 pointsize = 12  
)  
  
# Plot points at coordinates 0, 0 and 1, 1  
plot(  
 x = c(0, 1),  
 xlab = "x",  
 y = c(0, 1),  
 ylab = "y"  
)  
  
# Terminate the graphics device  
dev.off()

Description of functions and arguments being used for the first time:

* graphics.off(): Closes all graphics devices opened previously. Calling this before starting a new graph is particularly useful when debugging code and graphics devices might have been left open due to a fatal error in a previous execution of a script. For devices that write to a file, previously opened devices writing to the same file name will likely cause this code to fail because the file cannot be overwritten if it is currently open.
* pdf(): Opens a graphics device that writes to a postscript file
  + file: Path to the file where the graphics will be written
    - "./graphics\_test.pdf": Results in the file “graphics\_test.pdf” being written to the working directory.
  + width: Width of the canvas where graphical elements can be added
    - 6: Width of the canvas will be 6 inches
  + height: Height of the canvas where graphical elements can be added
    - 5: Height of the canvas will b 5 inches
  + pointsize: The default point size of characters and symbols
    - 12: Renderer will use a 12 point font for characters (12/72 = 1/6 of an inch). You might note from the documentation of pdf() that 12 is the default value for this argument. Therefore, the same result would be obtained if this argument were removed from the function call.
* plot() : A high level function for creating a new plot with a single line of code
  + x: Numerical vector of values for the x coordinate of each data point graphed
    - c(...): A function returning a numerical vector with the x coordinates (see below)
  + xlab: Character string to use as the label for the x axis
    - "x": x axis will be labeled with “x”
  + y: Numerical vector of values for the y coordinate of each data point graphed
    - c(...): A function returning a numerical vector with the y coordinates (see below)
  + ylab: Character string to use as the label for the y axis
    - "y": y axis will be labeled with “y”
* c(): A general concatenation function
  + 0, 1: Providing a collection of numbers to the generic arguments of the c() function results in a numeric atomic vector being returned with elements containing values corresponding to the numbers provided (in this case, a vector with two elements containing the values 0 and 1)
* dev.off(): Closes the most recently opened graphics device. For devices that write to a file (like pdf()), this typically needs to be called for the file to be written to the disk and available to be opened by other applications.

### 1.2. Graphing in Rmarkdown

When Rmarkdown is being compiled to an html file, the default graphics device used by code chunks embedded in the markdown is png(). This device creates a lossless portable graphics format file that can be embedded in the html and later displayed by a web browser application when viewing the file. The graphics device is open by default, so code in an Rmarkdown chunk does not directly open the device. Chunk options can be changed to allow graphics device arguments to be altered from their default values. For example, the code chunk generated the test graph below sets new values for three chunk options that result in a customized canvas for the associated graphics device. Code chunk options are set by including a list of key-value pairs after the r in the curly brackets specifying the beginning of the chunk.

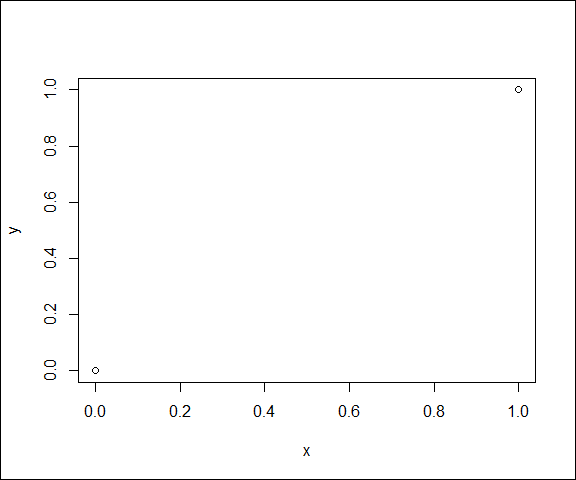
{r fig.width = 6, fig.height = 5, dev.args = list(pointsize = 12)}

Description of Rmarkdown chunk options being used for the first time:

* fig.width: Width of the canvas where graphical elements can be added
  + 6: Width of the canvas will be 6 inches
* fig.height: Height of the canvas where graphical elements can be added
  + 5: Width of the canvas will be 6 inches
* dev.args: List of arguments to be passed to the device
  + 12: Renderer will use a 12 point font for characters (12/72 = 1/6 of an inch)

The following code chunk provides a simple example of how using these chunk options specifies the properties of the canvas created by the default png() device used by Rmarkdown.

# Plot points at coordinates 0, 0 and 1, 1  
plot(  
 x = c(0, 1),  
 xlab = "x",  
 y = c(0, 1),  
 ylab = "y"  
)  
  
# Draw a box around the outer edge of the canvas  
box(which = "outer")



Description of functions and arguments being used for the first time:

* box(): Draws a box around predefined regions of a graphing canvas
  + which: Specifies which region of the graphing canvas around which to draw the box
    - "outer": Results in a box around the outer edge of the canvas

**Note to RStudio users:** The device used by RStudio to show graphics within the coding window is not the same device used to generate the compiled document. Be sure to view figures in the final compilation to be sure they have the desired characteristics, and do not assume the test figures in the coding window will necessarily match the final figure.

### 1.3. Other useful devices

Other useful graphics devices will render to a new pop-up window on your computer. Because these windows are operating system dependent, a different device is needed for Windows or Mac computers. The windows() function will open a pop-up graphics device on a PC, and the quartz() function will open a pop-up graphics device on a Mac. Use the following code as a basis for experimentation. Note that you should not call dev.off() after opening these devices or the pop-up will close before you get a chance to look at it. Also note that graphics.off() will close any lingering devices, including previously opened pop-up windows.

For a PC:

# Shut down any existing devices  
graphics.off()  
  
# Open the windows graphics device   
windows(  
 width = 6,  
 height = 5,  
 pointsize = 12  
)  
  
# Plot points at coordinates 0, 0 and 1, 1  
plot(  
 x = c(0, 1),  
 xlab = "x",  
 y = c(0, 1),  
 ylab = "y"  
)

For a Mac:

# Shut down any existing devices  
graphics.off()  
  
# Open the quartz graphics device   
quartz(  
 width = 6,  
 height = 5,  
 pointsize = 12  
)  
  
# Plot points at coordinates 0, 0 and 1, 1  
plot(  
 x = c(0, 1),  
 xlab = "x",  
 y = c(0, 1),  
 ylab = "y"  
)

Description of functions and arguments being used for the first time:

* windows(): Opens a graphics device that writes to a pop-up window in the Windows operating system
  + width: Width of the canvas where graphical elements can be added
    - 6: Width of the canvas will be 6 inches
  + height: Height of the canvas where graphical elements can be added
    - 5: Height of the canvas will b 5 inches
  + pointsize: The default point size of characters and symbols
    - 12: Renderer will use a 12 point font for characters (12/72 = 1/6 of an inch)
* quartz(): Opens a graphics device that writes to a pop-up window in MacOS
  + width: Width of the canvas where graphical elements can be added
    - 6: Width of the canvas will be 6 inches
  + height: Height of the canvas where graphical elements can be added
    - 5: Height of the canvas will b 5 inches
  + pointsize: The default point size of characters and symbols
    - 12: Renderer will use a 12 point font for characters (12/72 = 1/6 of an inch)

Finally, Windows users may want to experiment with the win.metafile() graphics device if you would like to render to the clipboard in a form that can be directly pasted into a Microsoft Word document or a Microsoft Powerpoint slide. Devices like bmp(), jpeg(), png(), and tiff() are available to render graphics into various common raster image files. These files tend to take up more space (depending on image quality), but are also more reliably transferable across operating systems.

## 2. Dissecting the plot() function

Taking full control over your figures requires a deeper understanding of what is happening when high-level functions like plot() are used. This understanding specifically requires knowing how S3 objects work in R, and why plot.default() is actually the function that gets called when you call plot() on a numeric vector. It also requires understanding how the device canvas is scaled and how individual graphical elements are added to the canvas by lower level graphing functions.

### 2.1. Why is plot.default() called when I call plot()?

One way to explore the S3 abstraction for plot() is to look at the implementation of the function. For functions implemented with R code, the implementation of the function can be viewed by typing the function name at the console prompt, without parentheses.

plot

## function (x, y, ...)   
## UseMethod("plot")  
## <bytecode: 0x000001b38eb8bff8>  
## <environment: namespace:base>

The entire implementation of this function is simply a call to the UseMethod() function, which means that the function that ultimately gets called depends on the data type of the first argument to the plot(x, y, ...) function. In other words, a call to plot is not enough information to know what is going to happen, you also need to know the data type of the first argument to plot(x, y, ...). This abstraction is an artifact of the S3 implementation of object oriented programming, and is the reason why calling help("plot", package = "base") yields so little specific information about the arguments that can be used with plot(). The remaining arguments that can be used depend on the data type of the first argument.

Throughout these exercises, the first argument (x) to plot(x, y, ...) will always be an atomic vector, which results in the plot.default() function being called by UseMethod() after plot() is called in a script. Note that the implementation of plot.default() contains far more complex code that actually creates a figure. For further exploration, note that calling help("plot.default") also yields far more detailed information about how to use the plot() function on vector arguments than help("plot", package = "base").

plot.default

## function (x, y = NULL, type = "p", xlim = NULL, ylim = NULL,   
## log = "", main = NULL, sub = NULL, xlab = NULL, ylab = NULL,   
## ann = par("ann"), axes = TRUE, frame.plot = axes, panel.first = NULL,   
## panel.last = NULL, asp = NA, xgap.axis = NA, ygap.axis = NA,   
## ...)   
## {  
## localAxis <- function(..., col, bg, pch, cex, lty, lwd) Axis(...)  
## localBox <- function(..., col, bg, pch, cex, lty, lwd) box(...)  
## localWindow <- function(..., col, bg, pch, cex, lty, lwd) plot.window(...)  
## localTitle <- function(..., col, bg, pch, cex, lty, lwd) title(...)  
## xlabel <- if (!missing(x))   
## deparse1(substitute(x))  
## ylabel <- if (!missing(y))   
## deparse1(substitute(y))  
## xy <- xy.coords(x, y, xlabel, ylabel, log)  
## xlab <- if (is.null(xlab))   
## xy$xlab  
## else xlab  
## ylab <- if (is.null(ylab))   
## xy$ylab  
## else ylab  
## xlim <- if (is.null(xlim))   
## range(xy$x[is.finite(xy$x)])  
## else xlim  
## ylim <- if (is.null(ylim))   
## range(xy$y[is.finite(xy$y)])  
## else ylim  
## dev.hold()  
## on.exit(dev.flush())  
## plot.new()  
## localWindow(xlim, ylim, log, asp, ...)  
## panel.first  
## plot.xy(xy, type, ...)  
## panel.last  
## if (axes) {  
## localAxis(if (is.null(y))   
## xy$x  
## else x, side = 1, gap.axis = xgap.axis, ...)  
## localAxis(if (is.null(y))   
## x  
## else y, side = 2, gap.axis = ygap.axis, ...)  
## }  
## if (frame.plot)   
## localBox(...)  
## if (ann)   
## localTitle(main = main, sub = sub, xlab = xlab, ylab = ylab,   
## ...)  
## invisible()  
## }  
## <bytecode: 0x000001b38f565678>  
## <environment: namespace:graphics>

For other examples of S3 functional abstractions, look at the implementation of the print() or summary() functions. See if you can find the specific implementations of these functions for particular data types that might be passed as the first argument to the abstraction (e.g., print.default() or print.data.frame()) . While detailed knowledge of the benefits of these object oriented principles of polymorphism is not strictly necessary to use R, deeper understanding of this topic will allow for more informed troubleshooting when a function does something unexpected. If nothing else, deeper understanding of S3 abstractions will provide a pathway to finding more detailed help() documentation on the functions you are actually using when calling abstract methods.

### 2.2. Scaling the canvas

Placing graphics on the canvas at a desired locations first requires definition of the scales for the axes of the canvas. Much of the first part of the plot.default() function is aliasing and bookkeeping that are a topic for different discussion. The lines of code that actually start to build the plot are

plot.new()  
localWindow(xlim, ylim, log, asp, ...)

Earlier in the function, localWindow() is established as a wrapper around the function plot.window() with the line

localWindow <- function(..., col, bg, pch, cex, lty, lwd) plot.window(...)

which means that calls to localWindow() are ultimately resulting in calls to plot.window(). This pattern is applied to many graphical functions being used to simplify the large number of arguments that need to be passed explicitly to those functions, but understanding the details of this practice is not necessary at this time.

The function plot.new() prepares the canvas for a new plot, and the function plot.window() scales the axes of the prepared canvas to establish the coordinate system for the canvas. The effect of these functions can be explored by calling them directly with simplified arguments, consistent with examples provided above. Here, a box is drawn around the outer edge to aid with visualization of the extent of the blank canvas. As above, the code chunk options are set to control the dimensions of the canvas being embedded in the compiled document, and all code chunks in the remainder of this section are configured the same way.

{r fig.width = 6, fig.height = 5, dev.args = list(pointsize = 12)}

# Prepare the canvas for a new plot  
plot.new()  
  
# Scale the canvas   
plot.window(  
 xlim = c(0, 1),  
 ylim = c(0, 1)  
)  
  
# Draw a box around the outer edge of the canvas  
box(which = "outer")



Description of functions and arguments being used for the first time:

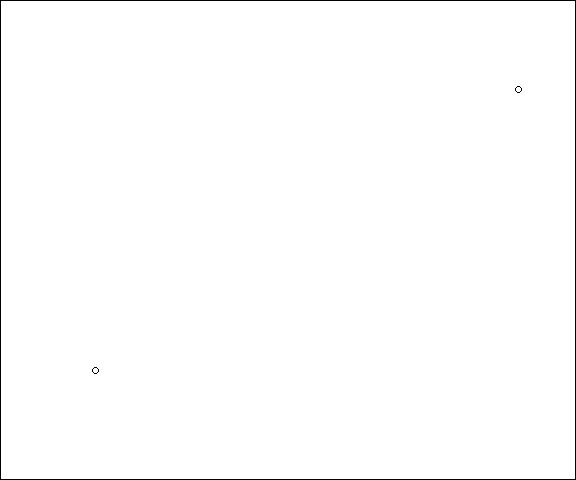
* plot.new(): Prepares the device canvas for a new plot
* plot.window(): Scales the canvas, providing an orthogonal 2-dimensional coordinate system (x is horizontal and y is vertical) for locating graphical objects placed on the canvas by later calls to graphing functions.
  + xlim: Numerical vector with 2 elements specifying the leftmost and rightmost values to be used in the graphing area of the coordinate system
  + ylim: Numerical vector with 2 elements specifying the bottommost and topmost values to be used in the graphing area of the coordinate system

The results are not particularly exciting. The fact that we have established a coordinate system does not result in any graphical elements being added to the canvas. However, it has provided a spatially rectified blank slate, such that there is now a context for adding graphical elements exactly where desired.

### 2.3. Plotting the data

Graphics objects can now be added to the canvas in layers. Lower layers will still be visible, unless opaque elements are added on top of them. The plot.default() function plots the data as the first visible element of the figure, using the function plot.xy(). The first argument (xy) is expected to be a list of vectors characterizing the coordinates to be plotted. This list can be conveniently constructed using the xy.coords() function with arguments specifying the x and y coordinates for a number of points to be plotted. The points plotted here are at coordinates 0, 0 and 1, 1. Multiple arguments are available to adjust the symbolism of each data point, but symbols will be black open circles if those arguments are not specified.

# Prepare the canvas for a new plot  
plot.new()  
  
# Scale the canvas   
plot.window(  
 xlim = c(0, 1),  
 ylim = c(0, 1)  
)  
  
# Draw a box around the outer edge of the canvas  
box(which = "outer")  
  
# Plot example data points  
plot.xy(  
 xy = xy.coords(  
 x = c(0, 1),  
 y = c(0, 1)  
 ),  
 type = "p"  
)



Description of functions and arguments being used for the first time:

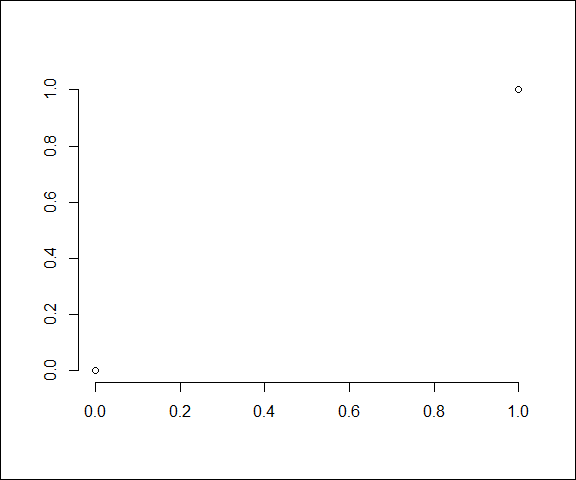
* plot.xy(): Renders the symbolism of a bivariate scatter plot on the canvas
  + xy: An coordinate list specifying information about the x an y coordinates of a set of data to be plotted.
    - xy.coords(...): A function returning the list of x and y coordinates for the data set to be plotted (see below)
  + type: The character string code for the type of scatter plot to render
    - "p": Results in data being plotted as point symbols in the scatter plot
* xy.coords(): Creates a coordinate list for a bivariate data set to be plotted
  + x: Numerical vector of values for the x coordinate of each data point graphed
    - c(0, 1): Two data points with x coordinates of 0 and 1, respectively.
  + y: Numerical vector of values for the y coordinate of each data point graphed
    - c(0, 1): Two data points with y coordinates of 0 and 1, respectively.

The result is two points on the graph, exactly as would be expected by two points plotted within the domain defined in plot.window(). However, without further reference for how the scaling of the canvas was completed, interpretation of these locations as 0, 0 and 1, 1 is impossible.

### 2.4. Creating the axes

The Axis() function provides a tool for adding axes describing the scale of the canvas in the area where the data are being plotted. This function is highly configurable to make an axis with ticks and labels exactly where desired, but the default arguments will attempt to create ticks and labels that are reasonably legible. Here, the only arguments provided are the side of the graph where the axis should be located (1 is the bottom and 2 is the left) and the range of values representing the extremes of the ticks that should be provided on the axis (0 to 1 for both axes).

# Prepare the canvas for a new plot  
plot.new()  
  
# Scale the canvas   
plot.window(  
 xlim = c(0, 1),  
 ylim = c(0, 1)  
)  
  
# Draw a box around the outer edge of the canvas  
box(which = "outer")  
  
# Plot example data points  
plot.xy(  
 xy = xy.coords(  
 x = c(0, 1),  
 y = c(0, 1)  
 ),  
 type = "p"  
)  
  
  
# Add an x axis  
Axis(  
 x = c(0, 1),  
 side = 1  
)  
  
# Add a y axis  
Axis(  
 x = c(0, 1),  
 side = 2  
)



Description of functions and arguments being used for the first time:

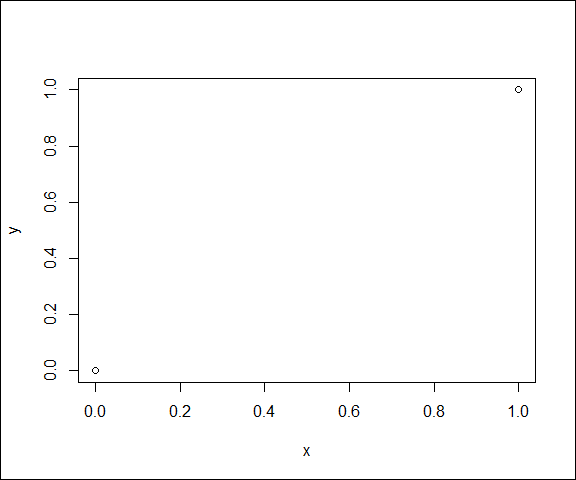
* Axis(): Renders the symbolism of an axis with labeled ticks
  + x: A numerical vector with 2 elements specifying the range of values on the axis from the lowest to highest tick. Note that use of “x” as the argument name in this case has nothing to do with the x axis of the figure. By convention, the first argument of many R functions is generically named “x”, due to the nature of S3 classes and the potential for this argument to be of diverse types. The argument containing the extremes for the scaling is called x regardless of the axes to which the call to Axis() is being targeted, and the side argument (see below) designates the orientation with which an axis is created and applied.
    - c(0, 1): Start ticks at 0 and end ticks at 1
  + side: An integer specifying the side of the plotting area upon which to place the axis (1 = bottom, 2 = left, 3 = top, 4 = right)
    - 1 or 2: Call with 1 creates the x axis and call with 2 creates the y axis

The axes clearly label the scaling of the canvas in the area dictated by the arguments provided to the Axis() function. The consistency of the scaling established by plot.window() is evident in that the data points plotted earlier line up with the appropriate ticks on the axes. These are the fundamentals of building any graph on the canvas.

### 2.5. Labeling the axes

The title() function is a unified tool for labeling different parts of the graph. Most style guides for figures suggest that the area where data are plotted should be differentiated from the rest of the figure. The box() function can be used with no arguments to put a box around the area with plotted data. As with previous graphics tools, the location and symbolism used for the elements being added are highly configurable with additional arguments. However, the default arguments are used for now, and detail on arguments that are commonly altered from the defaults are discussed later.

# Prepare the canvas for a new plot  
plot.new()  
  
# Scale the canvas   
plot.window(  
 xlim = c(0, 1),  
 ylim = c(0, 1)  
)  
  
# Draw a box around the outer edge of the canvas  
box(which = "outer")  
  
# Plot example data points  
plot.xy(  
 xy = xy.coords(  
 x = c(0, 1),  
 y = c(0, 1)  
 ),  
 type = "p"  
)  
  
  
# Add an x axis  
Axis(  
 x = c(0, 1),  
 side = 1  
)  
  
# Add a y axis  
Axis(  
 x = c(0, 1),  
 side = 2  
)  
  
# Add a box around the area where data is plotted  
box()  
  
# Add axis labels  
title(  
 xlab = "x",  
 ylab = "y"  
)



Description of functions and arguments being used for the first time:

* title(): Renders the symbolism of labels on various elements of a graphing canvas
  + xlab: A character string (or expression) specifying the label for the x axis
    - "x": Results in the x axis being labeled with an “x”
  + ylab: A character string (or expression) specifying the label for the y axis
    - "y": Results in the y axis being labeled with a “y”

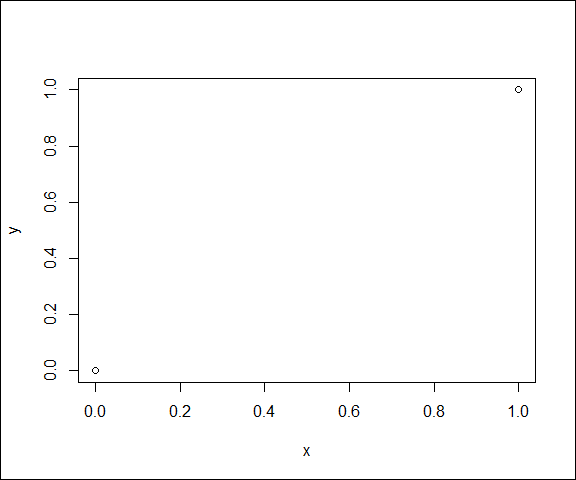
### 2.6. Summary

This dissection of plot.default() clearly requires substantially more lines of code than the original example of a single call to plot() that generates exactly the same figure. The point here is not that you should avoid using plot() and subsequently plot.default(), because it clearly provides substantial convenience for basic plotting. The point is that your ability to debug and be creative with graphing code will expand if you understand how functions like plot.default() construct a graph, such that you can take advantage of the power of altering the arguments for each component of the graph should it be necessary to do something plot.default() does not allow. Furthermore, understanding of these details allows for much more informed troubleshooting should high level functions like plot.default() produce something unexpected in a given application.

## 3. Taking control of base R graphics

A revisit of the code and result for the basic figure introduced earlier reveals that the plot() function must be making several assumptions about the desired format of the plot.

# Plot points at coordinates 0, 0 and 1, 1  
plot(  
 x = c(0, 1),  
 xlab = "x",  
 y = c(0, 1),  
 ylab = "y"  
)  
  
# Draw a box around the outer edge of the canvas  
box(which = "outer")



Note that many aesthetic or technical aspects of this plot did not have to be defined in values passed to arguments in the function used, including: the use of point symbols rather than lines; the shape and color of the points; the thickness of boxes and axes lines; and the location of the axes relative to the edges of the canvas (among many others). However, computers are generally not able to guess at what these aspects of plots should look like, so copious information about the default appearance of graphs must exist somewhere in the underlying R data structures. The trick is to know where that information is stored or to understand the functions that allow it to be altered.

A common location for default graphical information is the definition of default values for the arguments to graphical functions. For example, the plot.default()function (e.g., what is called if using plot() on vectors) accepts the optional argument type that controls the type of symbolism used for the data in the graph. The values that can be passed to type to control the symbolism are documented in the package containing the function and in this case can be accessed using the help("plot.default"). The potential values for arguments are usually described in either the **Arguments** section or **Details** section of the function documentation. Any default value defined for a parameter is usually described in the **Usage** or **Arguments** section. A quick and succinct way to view the definition of a function’s arguments is to use the args() function (e.g., args(plot.default)).

# Display the definition of arguments for the function plot.default  
args(name = plot.default)

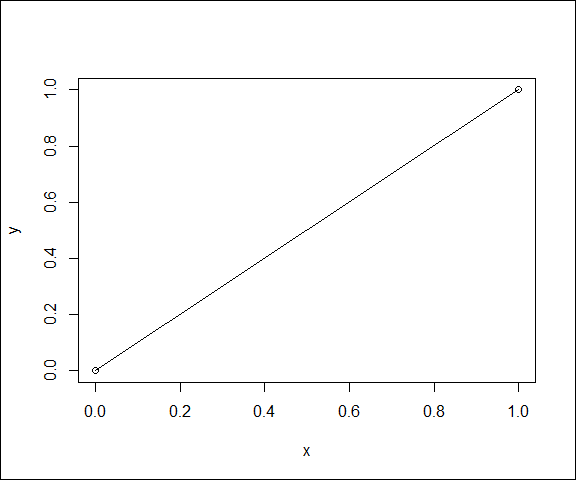
## function (x, y = NULL, type = "p", xlim = NULL, ylim = NULL,   
## log = "", main = NULL, sub = NULL, xlab = NULL, ylab = NULL,   
## ann = par("ann"), axes = TRUE, frame.plot = axes, panel.first = NULL,   
## panel.last = NULL, asp = NA, xgap.axis = NA, ygap.axis = NA,   
## ...)   
## NULL

Description of functions and arguments being used for the first time:

* args(): Returns the argument definitions for a function. Return type is a function with the argument definition but a NULL code block.
  + name: The name of the function for which the argument definition is retrieved
    - plot.default: Causes the function to return the argument definitions for the function plot.default()

Note that the argument type is followed by the assignment = "p", which designates that it will take on the value of the character string "p" if a value for that argument is not provided in a given call to the function. Any differing value provided for type in a call to the function will replace the default value and thus change the symbolism in the plot. For example, passing the value "o" to the argument type will result in an overlay of both points and lines.

# Plot points at coordinates 0, 0 and 1, 1  
plot(  
 x = c(0, 1),  
 xlab = "x",  
 y = c(0, 1),  
 ylab = "y",  
 type = "o"  
)  
  
# Draw a box around the outer edge of the canvas  
box(which = "outer")



Description of functions and arguments being used for the first time:

* plot() : A high level function for creating a new plot with a single line of code
  + type : A character string that defines the symbolism used when plotting data. For example, a value of “p” denotes points, “l” denotes lines, and “o” denotes an overlay of points and lines (among others).
    - "o": Results in graphing data as an overlay of points and lines

Extensive information used to plot graphical elements is also stored in a collection of default values for common graphical parameters associated with the open device. The current default values for these graphical parameters can be accessed by passing the names of those parameters as arguments to the par() function. For example, the graphics parameter that determines the symbols used for data points is pch, so the current value of “pch” can viewed with…

# Display the value of the graphical parameter pch for the current graphics device  
par("pch")

## [1] 1

Description of functions and arguments being used for the first time:

* par(): Returns the argument definitions for a function. Return type is a function with the argument definition but a NULL code block.
  + ...: The ellipses in the definition of arguments to a function indicate that the function will accept any number of generic arguments when called. “Generic” means that the function will either interpret the comma-delimited generic collection of elements without explicit definition of argument names, or it means that any generic arguments provided will be passed generically to other functions called when the function’s code is executed. Use of ellipses in graphical functions is a very common practice for passing values for graphical parameters generically from higher level functions to lower level functions.
    - "pch": Passing the character string “pch” generically to par() results in a return value of the current default value of the correspondingly named graphical parameter associated with the current graphics device. Note that multiple parameter names can be provided in a comma delimited list of character strings, which will return a named list of values with each element providing the current default value for each graphical parameter provided. For example par("pch", "lty") will return the current default values for the point type and line type graphical parameters.

By default, pch typically has a value of 1, which specifies an open circle, hence the symbols plotted in the graphs so far have all been open circles. The par function can also be used to change the values of graphical parameters based on a collection of <parameter> = <value> in the argument to the function call. Thus, the default symbol can be changed to an open triangle by changing the value of the graphics parameter pch to 2.

# Change the value of the graphical parameter pch for the current graphics device  
par(pch = 2)  
  
# Display the value of the graphical parameter pch for the current graphics device  
par("pch")

## [1] 2

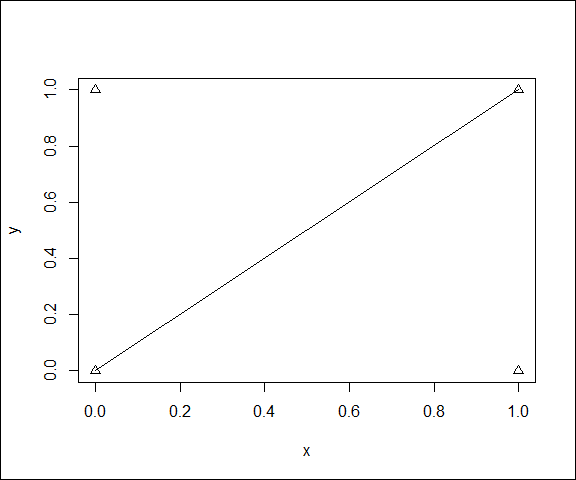
Description of functions and arguments being used for the first time:

* par(): Returns the argument definitions for a function. Return type is a function with the argument definition but a NULL code block.
  + ...: The ellipses in the definition of arguments to a function indicate that the function will accept any number of generic arguments when called
    - pch = 2: Passing a named value to the generic arguments in par() will result in the default value for the named graphical parameter to be changed to the provided value. Setting the graphical parameter pch to 2 will result in data being represented by open triangles.

Changing the value of a graphical parameter with par() will change the default value within the open graphics device for all subsequent calls to graphical functions, until the device is closed or reset. If we add an additional plot to the basic graph, it will also use the new setting of the graphical parameter pch.

The effect of this change can be demonstrated by calling two different graphing functions that plot data points in sequence. The understanding gained from dissecting the plot() function provides one method for adding an additional plot to the same basic graph, putting points in the unused corners of the basic graph using the plot.xy() function.

# Change the value of the graphical parameter pch for the current graphics device  
par(pch = 2)  
  
# Plot points at coordinates 0, 0 and 1, 1  
plot(  
 x = c(0, 1),  
 xlab = "x",  
 y = c(0, 1),  
 ylab = "y",  
 type = "o"  
)  
# Plot example data points  
plot.xy(  
 xy = xy.coords(  
 x = c(0, 1),  
 y = c(1, 0)  
 ),  
 type = "p"  
)  
  
# Draw a box around the outer edge of the canvas  
box(which = "outer")



All the data points are open triangles, demonstrating how changing the pch graphical parameter with par() altered the value for all subsequent calls to graphics functions within the code chunk. Note that in Rmarkdown, each code chunk will open a separate graphics device, such that any changes to graphics parameters applied to the device in one code chunk will not propagate to other code chunks without further action.

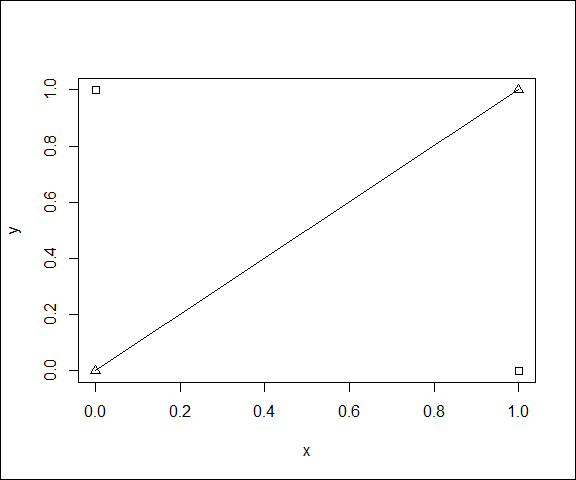
The main reason that graphics parameters controlled by par() have influence with subsequent graphing code is that they often represent the default values for arguments to lower level graphing functions. Therefore, the default values of the graphics parameters can often be locally overridden using values passed directly to those arguments. For example, consider the default value for the pch parameter in the list of arguments to the plot.xy() function.

args(plot.xy)

## function (xy, type, pch = par("pch"), lty = par("lty"), col = par("col"),   
## bg = NA, cex = 1, lwd = par("lwd"), ...)   
## NULL

The default value for the pch argument comes directly from the graphical parameters via a call to par("pch"). Therefore, the value for the graphing parameter that would normally result in triangle symbols from the previous example can be overridden for the call to plot.xy() by directly setting a value for the pch argument.

# Change the value of the graphical parameter pch for the current graphics device  
par(pch = 2)  
  
# Plot points at coordinates 0, 0 and 1, 1  
plot(  
 x = c(0, 1),  
 xlab = "x",  
 y = c(0, 1),  
 ylab = "y",  
 type = "o"  
)  
# Plot example data points  
plot.xy(  
 xy = xy.coords(  
 x = c(0, 1),  
 y = c(1, 0)  
 ),  
 type = "p",  
 pch = 0  
)  
  
# Draw a box around the outer edge of the canvas  
box(which = "outer")



Description of functions and arguments being used for the first time:

* plot.xy(): Renders the symbolism of a bivariate scatter plot on the canvas
  + pch: A vector of integers specifying the symbol to use for data points.
    - 0: A value of zero specifies an open square symbol

In this case, the graphical parameter of pch is ignored in place of the value directly passed to the pch argument of the plot.xy() function, hence the squares in the second plot rather than the triangles.

The remainder of this section provides tutorials on adjusting the values passed to arguments of graphical functions or the default values of graphics parameters used by those functions, which together provide a great deal of explicit control over the scaling, formatting, symbolism, and labeling used in base R graphics. Throughout the remainder of this section, frequent reference to the base R documentation using help("par") or the help for lower level graphing functions is suggested to convince yourself you understand the consequences of changing the default value for each graphics parameter before calling graphical functions. As always, experimentation is also highly encouraged.

### 3.1. Taking control of scaling

Scaling of the device canvas is controlled by function arguments or graphical parameters that define the spacing desired both within the plotting area and in the margins. Two sets of margins provide full control around both individual plots and around multi-panel plots. Buffering of the limits in the x and y directions within the plotting area also influence the overall scaling of the canvas.

#### 3.1.1 Inner margins

Scaling or rescaling a device canvas in base R (i.e., use of plot.window()) is first determined by defining the locations of the bottom, left, top, and right edges of a *plot region*. The canvas is subsequently scaled such that the coordinate values associated with those edges will result in the symbols for plotted data within the ranges of those coordinates to appear within that region.

The location of the edges of a plot region are determined by two sets of nested margins: the *margins* for individual plots and the *outer margins* for the area around multi-panel plots. Default graphics parameters are typically initially configured for single graphs (not multi-panel graphs), so the outer margins are typically set to zero by default. Depending on units used, the outer margins can be set with the graphical parameters oma or omi.

# Retrieve values of the oma and omi  
# graphics parameters  
par("oma", "omi")

## $oma  
## [1] 0 0 0 0  
##   
## $omi  
## [1] 0 0 0 0

Description of functions and arguments being used for the first time:

* par(): Returns the argument definitions for a function. Return type is a function with the argument definition but a NULL code block.
  + ...: The ellipses in the definition of arguments to a function indicate that the function will accept any number of generic arguments when called
    - “oma” or “omi” `: Returns the current setting for the graphical parameter defining the outer margins. The return value will be a vector with 4 elements designating the bottom, left, top, and right margins, in that order.

Multi-panel plots will be covered in more detail later, so we will not experiment with outer margins here. Just note for now that the default values here are typically zero and thus have no effect on scaling.

With all outer margins defaulting to zero, the scaling for the basic plot we have examined has been determined by the default values for the margins for individual plots.

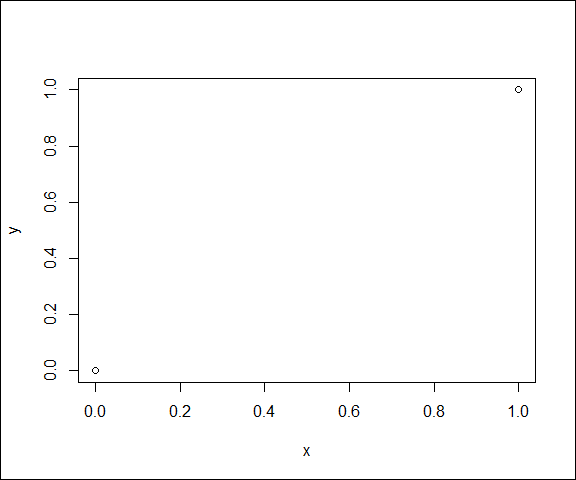
# Retrieve values of the mar and mai   
# graphics parameters  
par("mar", "mai")

## $mar  
## [1] 5.1 4.1 4.1 2.1  
##   
## $mai  
## [1] 1.02 0.82 0.82 0.42

Description of functions and arguments being used for the first time:

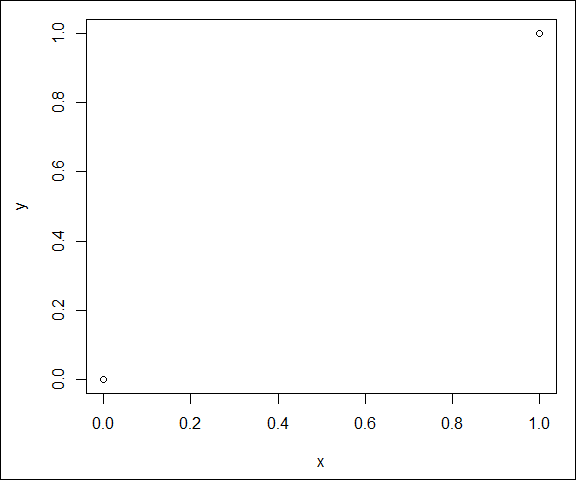
* par(): Returns the argument definitions for a function. Return type is a function with the argument definition but a NULL code block.
  + ...: The ellipses in the definition of arguments to a function indicate that the function will accept any number of generic arguments when called
    - "mar" or "mai": Returns the current setting for the graphical parameter defining the default margins for an individual graph. The return value will be a vector with 4 elements designating the bottom, left, top, and right margins, in that order. The parameter mar is in the relative units of *lines*, which are sized relative to the point size of the font being used in labels in the margin of the graph. The parameter mai is in absolute units of inches, which is independent of font size. If either of these parameter values is changed using the par() function, the other is automatically updated based on the font size being used.

# Plot points at coordinates 0, 0 and 1, 1  
plot(  
 x = c(0, 1),  
 xlab = "x",  
 y = c(0, 1),  
 ylab = "y"  
)  
  
# Draw a box around the outer edge of the canvas  
box(which = "outer")



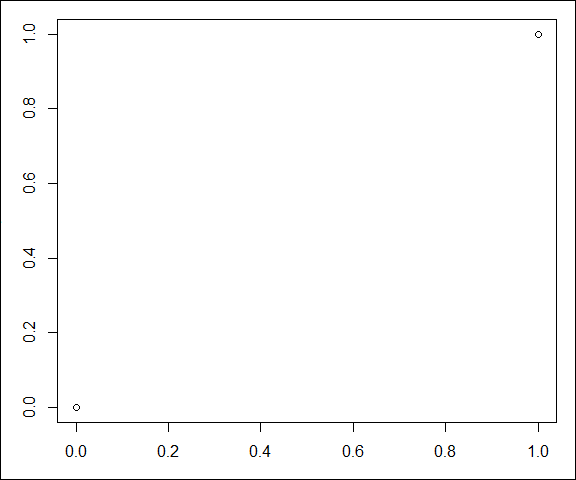
The extra space at the top and right of this basic plot is determined by the default values for mar or mai. Full control and consistency of the scaling can be gained by setting these graphical parameters explicitly before using graphical functions (like plot.window()) that reference them.

# Define margins leaving room for axes on the  
# bottom and left sides  
par(  
 omi = c(0, 0, 0, 0),  
 mai = c(0.9, 0.9, 0.2, 0.2)  
)  
  
# Plot points at coordinates 0, 0 and 1, 1  
plot(  
 x = c(0, 1),  
 xlab = "x",  
 y = c(0, 1),  
 ylab = "y"  
)  
  
# Draw a box around the outer edge of the canvas  
box(which = "outer")



Experiment with values for the margins to be sure you understand their behavior. For example, recognize the potentially confusing behavior that labels might be cut off if the scaling dictates they fall outside the canvas. Understanding behavior of margins will prevent going down the incorrect troubleshooting path of assuming the axis labels are not being generated in the following graph. They are being generated, just outside the canvas that is in view.

# Define margins without leaving room for axes labels on the  
# bottom and left sides  
par(  
 omi = c(0, 0, 0, 0),  
 mai = c(0.6, 0.6, 0.2, 0.2)  
)  
  
# Plot points at coordinates 0, 0 and 1, 1  
plot(  
 x = c(0, 1),  
 xlab = "x",  
 y = c(0, 1),  
 ylab = "y"  
)  
  
# Draw a box around the outer edge of the canvas  
box(which = "outer")



#### 3.1.2 Buffers within the plot region

The scaling that determines the coordinates associated with the edges of the plot region is controlled by the combination of the xlim and ylim arguments to plot.window() and the style of each axis. Unless otherwise specified, the xlim and ylim will be determined by the associated minimum and maximum values in the vectors begin plotted. In the basic plot, note that there is extra space between the edges of the plot region and the minima of 0 and the maxima of 1. This space is created because the default value for each axis type designates a surrounding buffer of 4% of the difference between the maxima and minima designated for each axis.

# Retrieve the values of the xaxs and yaxs   
# graphics parameters  
par("xaxs", "yaxs")

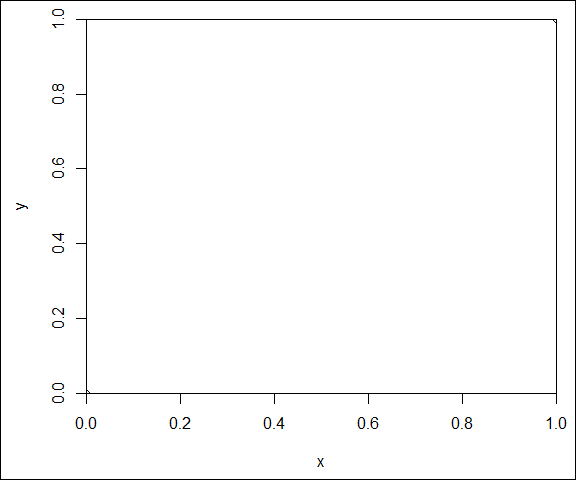
## $xaxs  
## [1] "r"  
##   
## $yaxs  
## [1] "r"

Description of functions and arguments being used for the first time:

* par(): Returns the argument definitions for a function. Return type is a function with the argument definition but a NULL code block.
  + ...: The ellipses in the definition of arguments to a function indicate that the function will accept any number of generic arguments when called
    - "xaxs" or "yaxs": Returns the current setting for the graphical parameter defining the type of the x or y axes. By default, values are the character string “r” indicated a “regular” scaling (includes a 4% buffer around the limits of the plot region).

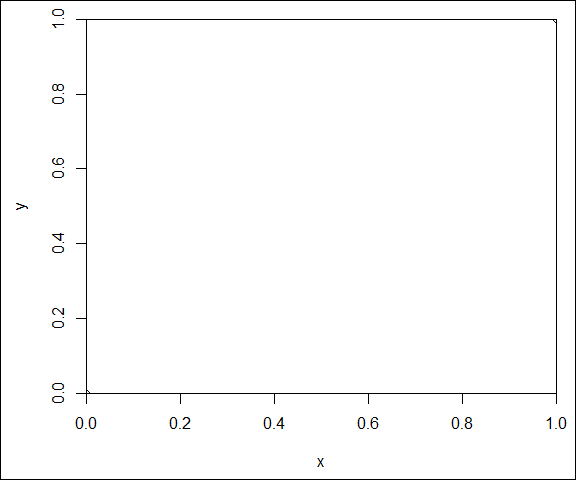
The default types for the axes can be specified generally for the device using the par() function.

# Define margins leaving room for axes on the  
# bottom and left sides  
# Set the types of both axes to internal  
par(  
 omi = c(0, 0, 0, 0),  
 mai = c(0.9, 0.9, 0.2, 0.2),  
 xaxs = "i",  
 yaxs = "i"  
)  
  
# Plot points at coordinates 0, 0 and 1, 1  
plot(  
 x = c(0, 1),  
 xlab = "x",  
 y = c(0, 1),  
 ylab = "y"  
)  
  
# Draw a box around the outer edge of the canvas  
box(which = "outer")



Values for the xaxs and yaxs arguments can also be passed generically to the plot() function, which will ultimately pass them on to plot.window() to achieve the same result. However, this technique will not change the default graphical parameter values, so the effects of using these arguments would not propagate to other graphs created later on the same device.

# Define margins leaving room for axes on the  
# bottom and left sides  
par(  
 omi = c(0, 0, 0, 0),  
 mai = c(0.9, 0.9, 0.2, 0.2)  
)  
  
# Plot points at coordinates 0, 0 and 1, 1  
plot(  
 x = c(0, 1),  
 xlab = "x",  
 xaxs = "i",  
 y = c(0, 1),  
 ylab = "y",  
 yaxs = "i"  
)  
  
# Draw a box around the outer edge of the canvas  
box(which = "outer")



Note that without the buffering, the symbols on this graph are barely visible. Only a quarter of each circle symbol can be seen in the corners of the plot region. The full circles are actually there, the parts of them outside the plot region are just being clipped. This clipping is, yet again, controlled by a graphics parameter (i.e., xpd).

# Retrieve the value of the xpd graphics parameter  
par("xpd")

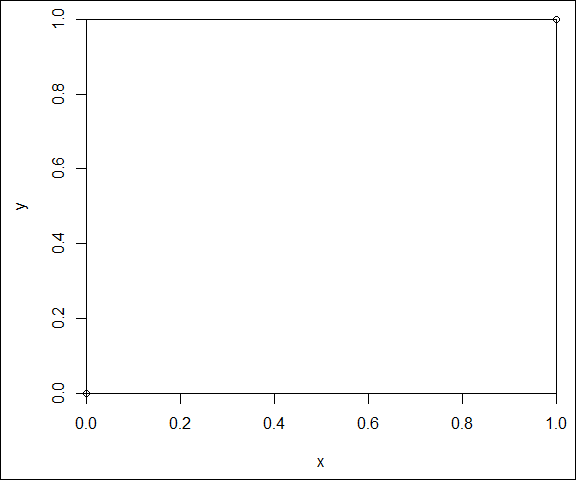
## [1] FALSE

Description of functions and arguments being used for the first time:

* par(): Returns the argument definitions for a function. Return type is a function with the argument definition but a NULL code block.
  + ...: The ellipses in the definition of arguments to a function indicate that the function will accept any number of generic arguments when called
    - "xpd": Returns the current setting for the logical control on whether plot region symbols are rendered when located in the margins. A value of “TRUE” means that symbols in the margins will be visible. A value of “FALSE” means they will not.

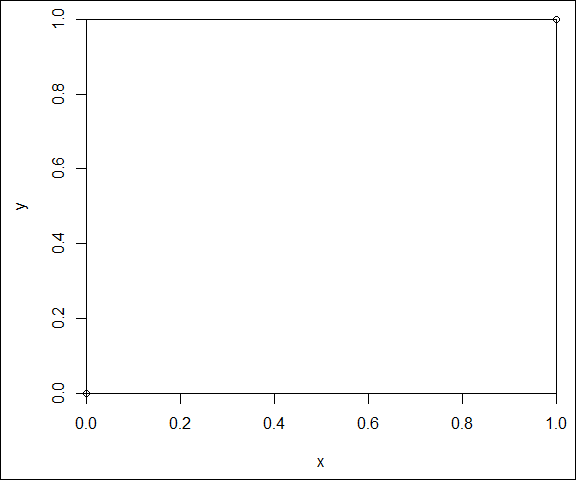
This graphics parameter can be altered to allow plot region symbols to be visible if plotted in the margins.

# Define margins leaving room for axes on the  
# bottom and left sides  
# Allow rendering of plot region symbolism  
# in the margins  
par(  
 omi = c(0, 0, 0, 0),  
 mai = c(0.9, 0.9, 0.2, 0.2),  
 xpd = TRUE  
)  
  
# Plot points at coordinates 0, 0 and 1, 1  
plot(  
 x = c(0, 1),  
 xlab = "x",  
 xaxs = "i",  
 y = c(0, 1),  
 ylab = "y",  
 yaxs = "i"  
)  
  
# Draw a box around the outer edge of the canvas  
box(which = "outer")



As with axis types, a value for the xpd argument can be generically provided to plot(). This generic argument is subsequently passed to plot.xy(), which has the effect of disabling clipping for the generated plot. However, this does not change the default graphics parameter, so this approach would have no effect on future plotting on the same device.

# Define margins leaving room for axes on the  
# bottom and left sides  
par(  
 omi = c(0, 0, 0, 0),  
 mai = c(0.9, 0.9, 0.2, 0.2)  
)  
  
# Plot points at coordinates 0, 0 and 1, 1  
plot(  
 x = c(0, 1),  
 xlab = "x",  
 xaxs = "i",  
 y = c(0, 1),  
 ylab = "y",  
 yaxs = "i",  
 xpd = TRUE  
)  
  
# Draw a box around the outer edge of the canvas  
box(which = "outer")



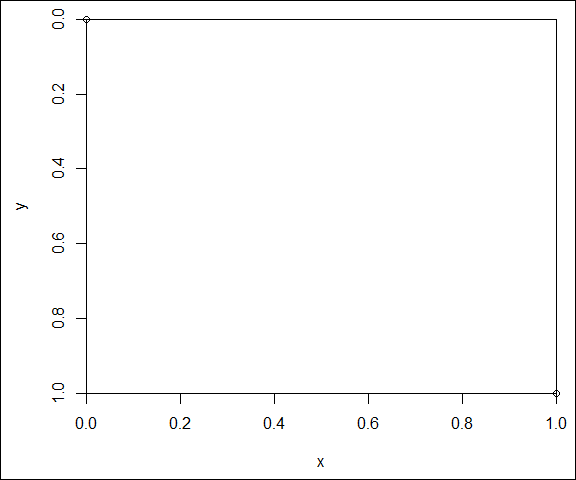
While locating the symbols representing data outside the plot region is not typical, this exercise illustrates the effect of the xpd graphical parameter or argument, which is useful in common labeling tasks like placing legends outside the axes of plots.

#### 3.1.3 Inverted and log scales

An inverted or logarithmic number system for coordinates does not affect the basic considerations of scaling a canvas.

An inverted scale on a given axis arises from switching the order of the limits on the plot region. The value for the limits of the plot region are given from the bottom to the top for the y axis and from the left to the right for the y axis. For example, the following will invert the y axis from the previous example by setting the y value associated with the bottom of the plot region to 1 and the y value associated with the top of the plot region to 0.

# Define margins leaving room for axes on the  
# bottom and left sides  
par(  
 omi = c(0, 0, 0, 0),  
 mai = c(0.9, 0.9, 0.2, 0.2)  
)  
  
# Plot points at coordinates 0, 0 and 1, 1  
plot(  
 x = c(0, 1),  
 xlab = "x",  
 xaxs = "i",  
 y = c(0, 1),  
 ylim = c(1, 0),  
 ylab = "y",  
 yaxs = "i",  
 xpd = TRUE  
)  
  
# Draw a box around the outer edge of the canvas  
box(which = "outer")

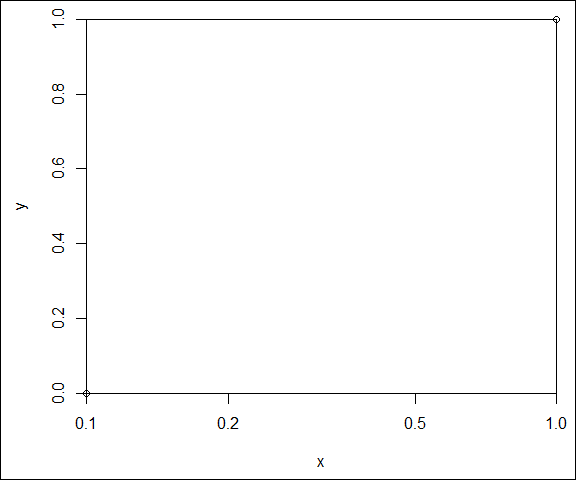


Description of functions and arguments being used for the first time:

* plot() : A high level function for creating a new plot with a single line of code
  + ylim: The limits of the values defining the plot region on the y axis
    - c(1, 0): Results in the bottom edge of the plot region being defined by a y value of 1 and the top edge of the plot region being defined by a y value of 0.

The use of a log scale alters the distribution of the number system used for either or both of the axes. A logarithmic number system can be specified by a passing a generic log argument to plot(), which is ultimately meaningfully passed on to the plot.window() function. The values for this argument can be “x” or “y” to specify a log scale on one of the axes, or “xy” to specify a log scale on both. For example, the x scaling can be altered to be on a log scale with x coordinate values at 0.1 and 1 (values of zero cannot be plotted in log space).

# Define margins leaving room for axes on the  
# bottom and left sides  
par(  
 omi = c(0, 0, 0, 0),  
 mai = c(0.9, 0.9, 0.2, 0.2)  
)  
  
# Plot points at coordinates 0, 0 and 1, 1  
plot(  
 x = c(0.1, 1),  
 xlab = "x",  
 xaxs = "i",  
 y = c(0, 1),  
 ylab = "y",  
 yaxs = "i",  
 xpd = TRUE,  
 log = "x"  
)  
  
# Draw a box around the outer edge of the canvas  
box(which = "outer")



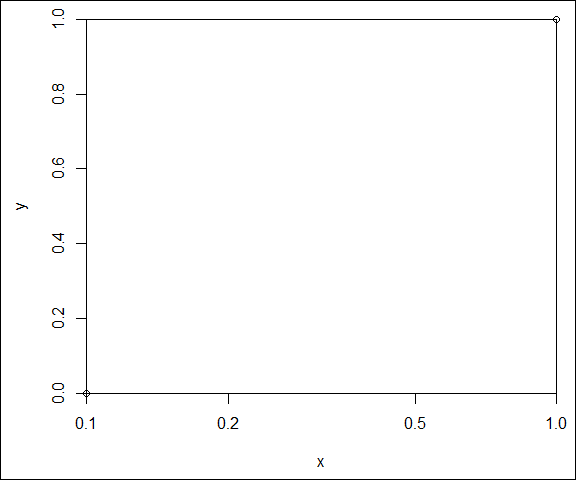
Description of functions and arguments being used for the first time:

* plot() : A high level function for creating a new plot with a single line of code
  + log: A character string specifying which axes should be in log space. The value “x” or “y” specifies one of the axes is in log space, and the value of “xy” specifies that both axes are in log space.
    - "x": Specifies that the x axis should be scaled in log space

#### 3.1.4 Summary

So far, we have been able to gain complete control of scaling while continuing to use the high-level abstract function plot(), which ultimately calls plot.default() and passes generic arguments on to lower level graphing functions. Understanding the behavior of these examples comes from recognizing that the resulting plot is being generated by the following sequence of calls to lower level graphical functions. For example, the graph with a logarithmic x axis, with altered margins, without buffers, and with visible symbols in the margins can be properly scaled and created with lower level functions as follows.

# Define margins leaving room for axes on the  
# bottom and left sides  
par(  
 omi = c(0, 0, 0, 0),  
 mai = c(0.9, 0.9, 0.2, 0.2)  
)  
  
# Prepare the canvas for a new plot  
plot.new()  
  
# Scale the canvas   
plot.window(  
 xlim = c(0.1, 1),  
 xaxs = "i",  
 ylim = c(0, 1),  
 yaxs = "i",  
 log = "x"  
)  
  
# Draw a box around the outer edge of the canvas  
box(which = "outer")  
  
# Plot example data points  
plot.xy(  
 xy = xy.coords(  
 x = c(0.1, 1),  
 y = c(0, 1)  
 ),  
 type = "p",  
 xpd = TRUE  
)  
  
  
# Add an x axis  
Axis(  
 x = c(0.1, 1),  
 side = 1  
)  
  
# Add a y axis  
Axis(  
 x = c(0, 1),  
 side = 2  
)  
  
# Add a box around the area where data is plotted  
box()  
  
# Add axis labels  
title(  
 xlab = "x",  
 ylab = "y"  
)  
  
# Draw a box around the outer edge of the canvas  
box(which = "outer")

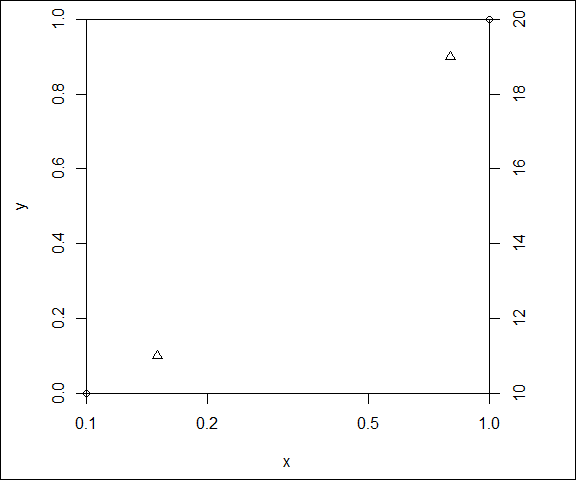


Description of functions and arguments being used for the first time:

* plot.window(): Scales the canvas, providing an orthogonal 2-dimensional coordinate system (x is horizontal and y is vertical) for locating graphical objects placed on the canvas by later calls to graphing functions.
  + xaxs or yaxs: Character string specifying the type of scaling to use for the x or y axis
    - "i": An “internal” axis does not include any buffering outside the minima and maxima of the values in the x and y vectors of coordinates
  + log: A character string specifying which axes should be in log space. The value “x” or “y” specifies one of the axes is in log space, and the value of “xy” specifies that both axes are in log space.
    - "x": Specifies that the x axis should be scaled in log space
* plot.xy(): Renders the symbolism of a bivariate scatter plot on the canvas
  + xpd: A logical value indicating whether plot region symbolism located at canvas coordinates in the margins outside the plot region should be visible (TRUE) or clipped (FALSE)
    - TRUE: Results in the portions of the data symbols (circles) in the margins outside the plot region to be visible.

Just these few scaling features and recognition that the same canvas can be rescaled over and over again shows the way forward to more advanced techniques, such as adding data to the plot region on an secondary y axis. Starting with the previous example, we can alter margins to make room for the secondary axis on the right, rescale the canvas using plot.window(), plot the data using plot.xy(), and add the secondary axis to the right side of the plot region using Axis(), without the need to learn any new functions or arguments.

# Define margins leaving room for axes on the  
# bottom, left, and right sides  
par(  
 omi = c(0, 0, 0, 0),  
 mai = c(0.9, 0.9, 0.2, 0.9)  
)  
  
# Prepare the canvas for a new plot  
plot.new()  
  
# Scale the canvas   
plot.window(  
 xlim = c(0.1, 1),  
 xaxs = "i",  
 ylim = c(0, 1),  
 yaxs = "i",  
 log = "x"  
)  
  
# Draw a box around the outer edge of the canvas  
box(which = "outer")  
  
# Plot example data points  
plot.xy(  
 xy = xy.coords(  
 x = c(0.1, 1),  
 y = c(0, 1)  
 ),  
 type = "p",  
 xpd = TRUE  
)  
  
# Add an x axis  
Axis(  
 x = c(0.1, 1),  
 side = 1  
)  
  
# Add a y axis  
Axis(  
 x = c(0, 1),  
 side = 2  
)  
  
# Rescale the canvas for a secondary axis  
plot.window(  
 xlim = c(0.1, 1),  
 xaxs = "i",  
 ylim = c(10, 20),  
 yaxs = "i",  
 log = "x"  
)  
  
# Plot example data points using the secondary scaling  
plot.xy(  
 xy = xy.coords(  
 x = c(0.15, 0.8),  
 y = c(11, 19)  
 ),  
 type = "p",  
 pch = 2  
)  
  
# Add the secondary y axis on the right side  
Axis(  
 x = c(10, 20),  
 side = 4  
)  
  
# Add a box around the area where data is plotted  
box()  
  
# Add axis labels  
title(  
 xlab = "x",  
 ylab = "y"  
)  
  
# Draw a box around the outer edge of the canvas  
box(which = "outer")



While the functions necessary to add appropriate axis labels and legends to this graph have not yet been discussed, detailed understanding of just a few scaling controls and lower level graphing functions add tremendous potential for creativity and informed troubleshooting.

The graphics parameters and graphical function arguments that control margins, limits on the plot region, and axis types together give complete control over canvas scaling. The xpd graphical parameter or graphical function argument allows subsequent control over whether plot region symbols are visible when they are placed at coordinates that are in the margins outside the plot region.

### 3.2. Taking control of axes

### 3.3. Taking control of plotting data

### 3.4. Taking control of labeling