1.2.2 *Functions and arguments*

At this point, you have obtained an impression of the way R works, and we have already used some of the special terminology when talking about the plot *function*, etc. That is exactly the point: Many things in R are done using *function calls*, commands that look like an application of a mathematical function of one or several variables; for example, `log(x)` or `plot(height, weight)`.

The format is that a function name is followed by a set of parentheses containing one or more arguments. For instance, in `plot(height,weight)` the function name is `plot` and the arguments are `height` and `weight`. These are the *actual arguments*, which apply only to the current call. A function also has *formal arguments*, which get connected to actual arguments in the call.

When you write `plot(height, weight)`, R assumes that the first argument corresponds to the *x*-variable and the second one to the *y*-variable. This is known as *positional matching*. This becomes unwieldy if a function has a large number of arguments since you have to supply every one of them and remember their position in the sequence. Fortunately, R has methods to avoid this: Most arguments have sensible defaults and can be omitted in the standard cases, and there are nonpositional ways of specifying them when you need to depart from the default settings.

The `plot` function is in fact an example of a function that has a large selection of arguments in order to be able to modify symbols, line widths, titles, axis type, and so forth. We used the alternative form of specifying arguments when setting the plot symbol to triangles with `plot(height, weight, pch=2)`.

The `pch=2` form is known as a *named actual argument*, whose name can be matched against the formal arguments of the function and thereby allow *keyword matching* of arguments. The keyword `pch` was used to say that the argument is a specification of the plotting character. This type of function argument can be specified in arbitrary order. Thus, you can write `plot(y=weight,x=height)` and get the same plot as with `plot(x=height,y=weight)`.

The two kinds of argument specification — positional and named — can be mixed in the same call.

Even if there are no arguments to a function call, you have to write, for example, `ls()` for displaying the contents of the workspace. A common error is to leave off the parentheses, which instead results in the display of a piece of R code since `ls` entered by itself indicates that you want to see the definition of the function rather than execute it.

The *formal arguments* of a function are part of the function definition. The set of formal arguments to a function, for instance `plot.default` (which is the function that gets called when you pass `plot` an `x` argument for which no special plot method exists), may be seen with

```
> args(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, ...)
```

Notice that most of the arguments have defaults, meaning that if you do not specify (say) the `type` argument, the function will behave as if you had passed `type="p"`. The `NULL` defaults for many of the arguments really serve as indicators that the argument is unspecified, allowing special behaviour to be defined inside the function. For instance, if they are not specified, the `xlab` and `ylab` arguments are constructed from the actual arguments passed as `x` and `y`. (There are some very fine points associated with this, but we do not go further into the topic.)

The triple-dot (`...`) argument indicates that this function will accept additional arguments of unspecified name and number. These are often meant to be passed on to other functions, although some functions treat it specially. For instance, in `data.frame` and `c`, the names of the `...`-arguments become the names of the elements of the result.