3. R Functions

Contents

1 R Functions

1.1 Your First R Function

```
add2 <- function(x, y) {
  x + y
}

add2(1,2)
```

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

```
above10 <- function(x) {
  use <- x > 10
  x[use]
}

above <- function(x,n=10){
  use <- x > n
  x[use]
}

x <- 1:20

above(x,12)
```

```
## [1] 13 14 15 16 17 18 19 20
```

```
above(x)
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20
```

```
columnmean <- function(y, removeNA = TRUE) {
  nc <- ncol(y)
  means <- numeric(nc)
  for (i in 1:nc) {
    means[i] <- mean(y[, i], na.rm = removeNA)
  }
  means
}

columnmean(airquality)
```

```
## [1]  42.129310 185.931507   9.957516  77.882353   6.993464  15.803922
```

```
columnmean(airquality, FALSE)
```

```
## [1]        NA        NA  9.957516 77.882353  6.993464 15.803922
```

1.2 Functions (part 1)

Functions are created using the function() directive and are stored as R objects just like anything else. In particular, they are R objects of class "function".

```
f <- function(<arguments>) {
## Do something interesting
}
```

Functions in R are "first class objects", which means that they can be treated much like any other R object. Importantly,

· Functions can be passed as arguments to other functions · Functions can be nested, so that you can define a function inside of another function · The *return value of a function is the last expression* in the function body to be evaluated.

1.2.1 Function Arguments

Functions have named arguments which potentially have default values. · The formal arguments are the arguments included in the function definition · The formals function returns a list of all the formal arguments of a function · Not every function call in R makes use of all the formal arguments · Function arguments can be missing or might have default values

1.2.2 Argument Matching

R functions arguments can be matched positionally or by name. So the following calls to sd are all equivalent

```
mydata <- rnorm(100)
sd(mydata)
```

```
## [1] 0.9911609
```

```
sd(x = mydata)
```

```
## [1] 0.9911609
```

```
sd(x = mydata, na.rm = FALSE)
```

```
## [1] 0.9911609
```

```
sd(na.rm = FALSE, x = mydata)
```

```
## [1] 0.9911609
```

```
sd(na.rm = FALSE, mydata)
```

```
## [1] 0.9911609
```

Even though it's legal, I don't recommend messing around with the order of the arguments too much, since it can lead to some confusion.

You can mix positional matching with matching by name. When an argument is matched by name, it is "taken out" of the argument list and the remaining unnamed arguments are matched in the order that they are listed in the function definition.

```
args(lm)
```

```
## function (formula, data, subset, weights, na.action, method = "qr",
##     model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
##     contrasts = NULL, offset, ...)
## NULL
```

The following two calls are equivalent.

```
lm(data = mydata, y ~ x, model = FALSE, 1:100)
```

```
## Error in model.frame.default(formula = y ~ x, data = mydata, subset = 1:100, : 'data' must be a data
```

```
lm(y ~ x, mydata, 1:100, model = FALSE)
```

```
## Error in model.frame.default(formula = y ~ x, data = mydata, subset = 1:100, : 'data' must be a data
```

Most of the time, named arguments are useful on the command line when you have a long argument list and you want to use the defaults for everything except for an argument near the end of the list

Named arguments also help if you can remember the name of the argument and not its position on the argument list (plotting is a good example).

Function arguments can also be partially matched, which is useful for interactive work. The order of operations when given an argument is 1. Check for exact match for a named argument
2. Check for a partial match
3. Check for a positional match

1.2.3 Defining a Function

```
f <- function(a,
              b = 1,
              c = 2,
              d = NULL) {

}
```

In addition to not specifying a default value, you can also set an argument value to NULL.

1.2.4 Lazy Evaluation

Arguments to functions are evaluated lazily, so they are evaluated only as needed.

```
f <- function(a, b) {
  a ^ 2
}
f(2)
```

```
## [1] 4
```

This function never actually uses the argument b, so calling f(2) will not produce an error because the 2 gets positionally matched to a.

```r
f <- function(a, b) {
  print(a)
  print(b)
}
f(45)
```

```
## [1] 45
```

```
## Error in f(45): argument "b" is missing, with no default
```

Notice that "45" got printed first before the error was triggered. This is because b did not have to be evaluated until after print(a). Once the function tried to evaluate print(b) it had to throw an error.

1.2.5 The "…" Argument

The … argument indicate a variable number of arguments that are usually passed on to other functions. · … is often used when extending another function and you don't want to copy the entire argument list of the original function

```r
myplot <-
  function(x, y, type = "l", ...) {
    plot(x, y, type = type, ...)
  }
```

- Generic functions use … so that extra arguments can be passed to methods (more on this later).

```r
mean
```

```
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x113597f20>
## <environment: namespace:base>
```

The … argument is also necessary when the number of arguments passed to the function cannot be known in advance.

```r
args(paste)
```

```
## function (..., sep = " ", collapse = NULL, recycle0 = FALSE)
## NULL
```

```r
args(cat)
```

```
## function (..., file = "", sep = " ", fill = FALSE, labels = NULL,
##     append = FALSE)
## NULL
```

1.2.6 Arguments Coming After the "…" Argument

One catch with … is that any arguments that appear after … on the argument list must be named explicitly and cannot be partially matched.

```r
args(paste)
```

```
## function (..., sep = " ", collapse = NULL, recycle0 = FALSE)
## NULL
```

```r
paste("a", "b", sep = ":")
```

```
## [1] "a:b"
```

```r
paste("a", "b", se = ":")
```

```
## [1] "a b :"
```