Stat 33A - Lecture 2

February 3, 2020

Announcements

First homework assignment posted to the bCourse.

Using Functions, Part 3

Last time we saw how to call a function and set the arguments.

```
sum(1, 2, 3)
```

```
## [1] 6
```

```
x = 1
```

Each argument gets assigned to a *parameter*, which is like a variable inside of the function.

Parameters are listed in each function's help file:

```
?log
```

You can set arguments based on the order of the parameters:

```
log(10, 2) # log of 10, base 2
```

```
## [1] 3.321928
```

Or by using the names of the parameters:

```
log(x = 10, base = 2)
```

```
## [1] 3.321928
```

```
log(base = 2, 10)
```

```
## [1] 3.321928
```

```
log(bas = 2, x = 10)
```

```
## [1] 3.321928
```

Data Types & Classes

In statistics, we categorize data into different types:

- Continuous (real numbers)
- Discrete (integers, or finite number of values)
- Logical (1 and 0s, T and Fs)
- Nominal (categorical values with no ordering)
- Ordinal (categorical values with ordering)
- Graph (network data)
- Textual (books, websites, etc)

In R, we categorize values into different classes.

Every value has at least one class.

Class answers the question "How does this thing behave?"

Check class with `class()`:

```r
class(5)
```

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

```r
class("hi")
```

```
## [1] "character"
```

```r
class(TRUE)
```

```
## [1] "logical"
```

```r
class(cos)
```

```
## [1] "function"
```

```r
x = 1
class(x)
```

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

R also has something called *types*. Not as important as classes for day-to-day programming.

Special Values

R has four special values.

`NA` represents a missing or unknown value in a data set.

```r
NA
```

```
## [1] NA
```

```r
class(NA)
```

```
## [1] "logical"
```

The missing value `NA` is contagious!

```r
5 + NA
```

```
## [1] NA
```

Using a unknown argument in a computation usually produces an unknown result.

`NULL` represents a value that's not defined *in R*. Functions often use `NULL` to mean the absence of a result.

```r
dim(5)
```

```
## NULL
```

```r
class(NULL)
```

```
## [1] "NULL"
```

For instance, if we try to get the matrix dimensions of a vector:

`NaN`, or "not a number", represents a value that's not defined mathematically. Produced by some computations:

```r
0 / 0
```

```
## [1] NaN
```

```r
class(NaN)
```

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

`Inf` represents infinity. Produced by some computations:

```r
- 5 / 0
```

```
## [1] -Inf
```

```r
class(Inf)
```

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

Comparison Operators

We saw operators for doing arithmetic: `+`, `-`, `*`, `/`, `^`

R also has operators for making comparisons.

Use `==` to check equality:

```r
1 == 2
```

```
## [1] FALSE
```

```r
NA == 1
```

```
## [1] NA
```

If you want to check equality to `NA`, use `is.na()` instead:

```r
is.na(NA)
```

```
## [1] TRUE
```

There are `is.` functions for the other special values, too.

```r
is.infinite(Inf)
```

```
## [1] TRUE
```

Check inequality with `<`, `<=`, `>`, and `>=`. For instance:

```r
4 <= 5
```

```
## [1] TRUE
```

Vectors

A vector is an ordered collection of values.

In R, all of the values in a vector must be the same data type.

Several ways to create a vector. One is the `c()` function, which ___c___ombines values:

```r
c(1, 2, 3)
```

```
## [1] 1 2 3
```

```r
c("hi", "hello", '33A')
```

```
## [1] "hi"    "hello" "33A"
```

Check length with `length()`:

```
length(c(1, 2, 3))
```

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

```
length(6)
```

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

```
class(c(1, 2, 3))
```

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

```
x = c(1, 7, 9, 8)
```

Another way is to create a sequence with `:` or `seq()`:

```
1:5
```

```
## [1] 1 2 3 4 5
```

More ways are described in this week's reading.

```
7:20
```

```
##  [1]  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

```
?seq
seq(1, 5, 0.5)
```

```
## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

R does not make any distinction between scalars and vectors.

A scalar is just a vector with one element.

You can name elements of a vector:

```
# Can use quotes to use numbers as names, but not recommended
x = c("dogs" = 3, "2" = 4)
```

```
x = c(dogs2 = 3, cats2 = 4)
```

```
names(x)
```

```
## [1] "dogs2" "cats2"
```

To set set names, use:

```
names(x) = c("a", "b")
```

```
names(x)
```

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

Many operations are *vectorized*, which means they are computed element-by-element. For instance:

```
c(1, 2, 3) + c(4, 5, 6)
```

```
## [1] 5 7 9
```

```
sin(c(1, 2, 3))
```

```
## [1] 0.8414710 0.9092974 0.1411200
```

```r
c(1, 2) == c(3, 2)
```

```
## [1] FALSE  TRUE
```

```r
c(1, 2) + c(3, 4, 5)
```

```
## Warning in c(1, 2) + c(3, 4, 5): longer object length is not a multiple of
## shorter object length
```

```
## [1] 4 6 6
```

```r
c(1, 2, 3) + 1
```

```
## [1] 2 3 4
```

What happens if we pass several different data types to `c()`?

```r
class(c(7, "hi", TRUE))
```

```
## [1] "character"
```

Implicit Coercion

R can automatically convert between ("coerce") types in one direction:

`logical -> integer -> numeric -> complex -> character`

```r
class(1 + 4i)
```

```
## [1] "complex"
```

```r
1 + 0i
```

```
## [1] 1+0i
```

For instance:

```r
5 + TRUE
```

```
## [1] 6
```

So if we pass different types to `c()`, R will attempt coercion:

```r
c(TRUE, 2.0)
```

```
## [1] 1 2
```

```r
class(c(1, sin))
```

```
## [1] "list"
```

There are data types R will never implicitly coerce:

```r
class(sin)
```

```
## [1] "function"
```

Lists

In a vector, every element has to have the same type.

A list is a container for elements with *different* types.

```r
list(1, sin, "hi")
```

```
## [[1]]
## [1] 1
##
## [[2]]
## function (x)  .Primitive("sin")
##
## [[3]]
## [1] "hi"
```