Data Types

2020

Basic Data Types

In this section we are going to review the 5 basic data types:

- logical
- integer
- double
- complex
- character

They define how the values are stored in the computer.

You can get an object's type using the `typeof` function.

`is.<class type>()` checks the <class type> of the object
`as.<class type>()` converts the object to <class type>

Logical

Logical values can take one of two values: `TRUE` or `FALSE`. They can also be represented as `T` and `F`.
Where `True` is the same as `1` and `False` is the same as `0`.

```
> x <- 1
> y <- 2
> z <- x > y
```

`typeof` function prints the type of the object

```
> typeof(z)
[1] "logical"
```

`is.logical` checks if the class type of the object is logical

```
> is.logical(2.5)
[1] FALSE
> is.logical(y)
```

```
[1] FALSE
> is.logical(z)
[1] TRUE
```

`as.logical` converts the type of the object to logical one

```
> x <- as.logical(10.5); x; typeof(x)
[1] TRUE
[1] "logical"
> x <- as.logical(1); x; typeof(x)
[1] TRUE
[1] "logical"
> x <- as.logical(0); x; typeof(x)
[1] FALSE
[1] "logical"
> x <- as.logical("string"); x; typeof(x)
[1] NA
[1] "logical"
```

Some more examples:

```
> F < T
[1] TRUE
> 5 * TRUE
[1] 5
> NA != NA
[1] NA
> NA == NA # You can use is.na(NA) to check if an object
is NA
[1] NA
> 2 < Inf
[1] TRUE
> Inf == Inf
[1] TRUE
> -Inf < Inf
[1] TRUE
> NA < Inf
[1] NA
```

Strings are compared lexicographically

```
> "data" == "statistics"
```

```
[1] FALSE
> "data" < "math"
[1] TRUE
```

Integer

If you want variable's type to be integer, you must set it explicitly.

```
> x <- 3; x
[1] 3
> typeof(x)
[1] "double"
> is.integer(x)
[1] FALSE
```

We can define it as integer in one of the following ways:

```
> y <- as.integer(12.35); y
[1] 12
> typeof(y)
[1] "integer"
> z <- 3L; z
[1] 3
> typeof(z)
[1] "integer"
```

Some more examples

```
> typeof(4L * 2)
[1] "double"
> typeof(6L / 2L)
[1] "double"
```

Double

It describes all the real numbers.

```
> k <- 1; k
[1] 1
> typeof(k)
[1] "double"
> is.numeric(k)
[1] TRUE
```

```
> x <- 10.5; x
[1] 10.5
> typeof(x)
[1] "double"
```

Complex

It describes the set of complex numbers.

```
> z <- 5 + 6i; z
[1] 5+6i
> typeof(z)
[1] "complex"
> is.complex(z)
[1] TRUE
> as.complex(4)
[1] 4+0i
```

Character

```
> s <- "statistics"; s
[1] "statistics"
> typeof(s)
[1] "character"
> is.character(s)
[1] TRUE
> as.character(123)
[1] "123"
```

Some operations with characters

`nchar` gives the number of characters in the string

```
> nchar(s)
[1] 10
```

`substr` extract substrings

```
> substr(s, start = 1, stop = 4)
[1] "stat"
```

It is worth mentioning that you can't concatenate strings with `+` sign.

```
> x <- "1"
> y <- "3.5"
> x + y
Error in x + y: non-numeric argument to binary operator
```

and we need to use `paste` function

```
> paste(x, y)
[1] "1 3.5"
```

Where the function converts the parameters to strings.

```
> x <- 1
> y <- 3.5
> paste(x, y)
[1] "1 3.5"
```

You can work with regular expressions using `grep` function.
For examples lets look at `state.name` data containing the 50 full state names of the United States of America.

```
> state.name
 [1] "Alabama"        "Alaska"         "Arizona"
"Arkansas"
 [5] "California"     "Colorado"       "Connecticut"
"Delaware"
 [9] "Florida"        "Georgia"        "Hawaii"
"Idaho"
[13] "Illinois"       "Indiana"        "Iowa"
"Kansas"
[17] "Kentucky"       "Louisiana"      "Maine"
"Maryland"
[21] "Massachusetts"  "Michigan"       "Minnesota"
"Mississippi"
[25] "Missouri"       "Montana"        "Nebraska"
"Nevada"
[29] "New Hampshire"  "New Jersey"     "New Mexico"
"New York"
[33] "North Carolina" "North Dakota"   "Ohio"
"Oklahoma"
[37] "Oregon"         "Pennsylvania"   "Rhode Island"
"South Carolina"
```

```
[41] "South Dakota"    "Tennessee"       "Texas"
"Utah"
[45] "Vermont"         "Virginia"        "Washington"
"West Virginia"
[49] "Wisconsin"       "Wyoming"
```

Let’s take the states containing `v` in their name

```
> grep(pattern = "v", state.name, value = TRUE)
[1] "Nevada"        "Pennsylvania"
```

parameter `value=TRUE` shows the matching elements, otherwise we will see their indexes

```
> grep(pattern = "v", state.name)
[1] 28 38
```

If we want to also see the states starting with `V` we need to add `ignore.case=TRUE`

```
> grep(pattern = "v", state.name, ignore.case = TRUE,
value = TRUE)
[1] "Nevada"         "Pennsylvania"  "Vermont"
"Virginia"
[5] "West Virginia"
```

And for example if we want to see only those names of states starting with `V`

```
> grep(pattern = "^V", state.name, value = TRUE)
[1] "Vermont"   "Virginia"
```

Derived Data Types

In this section we are going to review 2 derived data types:

- Factor
- Date

These data types are stored as either of the basic data types, but have additional attribute information that allows to be treated in special ways by certain functions in R. These attributes define the object's class and can be extracted from that object via the `attributes` function.

Factor

Factors are used to group variables into a fixed number of unique categories or levels.

For example imagine we have some weather sample:

```
> weather <- c("Sunny", "Sunny", "Sunny", "Cloudy", "Sunny", "Sunny", "Cloudy",
+               "Cloudy", "Stormy", "Cloudy", "Fog", "Sunny", "Sunny", "Cloudy",
+               "Cloudy", "Stormy", "Cloudy", "Fog", "Rain", "Rain", "Snow", "Snow")
> weather
 [1] "Sunny"  "Sunny"  "Sunny"  "Cloudy" "Sunny"  "Sunny"  "Cloudy" "Cloudy"
 [9] "Stormy" "Cloudy" "Fog"    "Sunny"  "Sunny"  "Cloudy" "Cloudy" "Stormy"
[17] "Cloudy" "Fog"    "Rain"   "Rain"   "Snow"   "Snow"
```

Lets convert it to factors. We have the same vector, but now with the unique labels of the categories.

```
> weather.factor <- as.factor(weather)
> weather.factor
 [1] Sunny  Sunny  Sunny  Cloudy Sunny  Sunny  Cloudy Cloudy Stormy Cloudy
[11] Fog    Sunny  Sunny  Cloudy Cloudy Stormy Cloudy Fog    Rain   Rain
[21] Snow   Snow
Levels: Cloudy Fog Rain Snow Stormy Sunny
```

So how is this object represented?

Using the `attributes` function we can see the attributes of the object.

```
> attributes(weather.factor)
$levels
[1] "Cloudy" "Fog"    "Rain"   "Snow"   "Stormy" "Sunny"

$class
```

```
[1] "factor"
```

The factor object has 2 attributes: `levels` and `class`. Where`levels` shows the unique values of the object.

What is the type of the factor object?

```
> typeof(weather)
[1] "character"
> typeof(weather.factor)
[1] "integer"
```

The type of the factor object is `integer`. Why is it storing them as `integer`?

The reason is the factor objects are storing each value as an integer that points to one of the unique levels.

We can see the class of the object using `class` function

```
> class(weather.factor)
[1] "factor"
```

We can see the unique levels and their order using `levels` function

```
> levels(weather.factor)
[1] "Cloudy" "Fog"    "Rain"    "Snow"    "Stormy" "Sunny"
```

The order in which the levels are displayed match their integer representation.

This will be used when making plots in the next chapter.

Date

`Date`, `POSIXlt` and `POSIXct` classes represent calendar dates.

- `as.Date` stores just dates using different origins

```
> d <- as.Date("2019-05-27"); d
[1] "2019-05-27"
```

You can see that the object is of class `Date` and its type is `double` representing the number of dates after the origin.

```
> class(d)
[1] "Date"
> typeof(d)
[1] "double"
```

Dates can be in different format, but the output is in ISO 8601 international standard format %Y-%m-%d

```
> d <- as.Date("05/27/19", format = "%m/%d/%y"); d
[1] "2019-05-27"
> d <- as.Date(18043, origin = "1899-12-30"); d
[1] "1949-05-25"
```

- `as.POSIXct` stores data and time as the number of seconds since January 1, 1970

```
> t <- as.POSIXct("2019-05-27 17:42"); t
[1] "2019-05-27 17:42:00 EEST"
> typeof(d)
[1] "double"
> class(t)
[1] "POSIXct" "POSIXt"
> as.Date(t)
[1] "2019-05-27"
```

- `as.POSIXlt` stores them as a list with elements for year, month, day, hour, minutes and seconds. We can see this using the function `attribute`.

```
> t <- as.POSIXlt("2019-05-27 17:42"); t
[1] "2019-05-27 17:42:00 EEST"
> typeof(d)
[1] "double"
> class(t)
[1] "POSIXlt" "POSIXt"
> as.Date(t)
[1] "2019-05-27"
> attributes(t)
$names
 [1] "sec"    "min"    "hour"   "mday"   "mon"    "year"
"wday"   "yday"
 [9] "isdst"  "zone"   "gmtoff"
```

```
$class
[1] "POSIXlt" "POSIXt"
```

You can easily take the year from the list using `$`. It shows the number of years since 1900, so we see 2019 as 119.

```
> t$year
[1] 119
```

Shows the month in 0 - 11 format, so we see May as 4.

```
> t$mon
[1] 4
```

Shows the day in 1 - 31 format

```
> t$mday
[1] 27
```

Shows the hours, minutes and seconds

```
> t$hour
[1] 17
> t$min
[1] 42
> t$sec
[1] 0
```