刘程华 2018011687 计91

1、What are the three properties of a vector,other than its contents?

The three properties of a vector are type, length, and attributes.

2、What are the four common types of atomic vectors? What are the two rare type?

The four common types of atomic vector are logical, integer, double (sometimes called numeric), and character. The two rarer types are complex and raw.

3、What are attributes? How do you get them and set them?

Attributes are any kind of information (e.g. additional metadata) that is assigned to an element of an object, and you can use them yourself to add information to any object. You can get them using the attribute() function and set them using the attr() function. The attr() function takes two important arguments. attr(object, name of attribute want to set/change).

Note: Objects in R can have many properties associated with them, called attributes. These properties explain what an object represents and how it should be interpreted by R. Quite often, the only difference between two similar objects is that they have different attributes. Some important attributes are listed below. Many objects in R are used to represent numerical data—in particular, arrays, matrices, and data frames. So many common attributes refer to properties of these objects.

- class - the calss of the object
- comment - a comment on the object; often a description of what the object means
- dim - dimensions of the object
- dimnames - names associated with each dimension of the object
- names - names of an object. Depends on object type, e.g. it returns the name of each data column in a data frame
- row.names - the name of each row in an object
- tsp - start time for an object. Useful for time series data.
- levels - levels of a factor

code example：

```r
# Example 1: adding a new attribute
head(iris)
attributes(iris)
attr(iris, "Species") <- "There are three: setosa, virginica, and versicolor"
attributes(iris)
# Through observation,we noticed that the attributes have increased

# Example 2: Renaming column names
head(cars)
attributes(cars)
attr(x = cars, which = "names") <- c("aaa", "bbb")
attributes(cars)
# Through observation,we noticed that the column names have changed
```

4、How is a list different from an atomic vector? How is a matrix different from a data frame?

The elements of a list can be any type (even a list); the elements of an atomic vector are all of the same type. You can't perform mathematical functions on a list, but you can on a vector. You can have a list of lists, you cannot have a vector of vectors.

Every element of a matrix must be the same type; in a data frame, the different columns can have different types.

5、Can you have a list that is a matrix? Can a data frame have a column that is a matrix?

No, but You can make a "list-array" by assigning dimensions to a list.

Yes, you can make a matrix a column of a data frame with df$x <- matrix(), or using I() when creating a new data frame data.frame(x = I(matrix())).

6、What's the difference between [, [[, and $ when applied to a list?

[selects sub-lists. It always returns a list; if you use it with a single positive integer, it returns a list of length one. `[[` selects an element within a list. `$` is a convenient shorthand: `x$y` is equivalent to `x[["y"]]`.

7、Why is the default missing value, NA, a logical vector? (Hint: What's special about logical vectors?)

The presence of missing values shouldn't affect the type of an object. Recall that there is a type-hierarchy for coercion from character >> double >> integer >> logical. When combining `NA`s with other atomic types, the `NA`s will be coerced to integer (`NA_integer_`), double (`NA_real_`) or character (`NA_character_`) and not the other way round. If `NA` was a character and added to a set of other values all of these would be coerced to character as well.

8、Why does x <- 1:5; x[NA] yield five missing values? Why is it different from x[NA_real_]?

In contrast to `NA_real`, `NA` has logical type and logical vectors are recycled to the same length as the vector being subset, i.e. `x[NA]` is recycled to `x[NA, NA, NA, NA, NA]`.

9、Why do you need to use doesn't unlist() to convert a list to an atomic vector? Why doesn't as.vector() work?

A list is already a vector, though not an atomic one!

Note that `as.vector()` and `is.vector()` use different defintions of "vector"!

```
is.vector(as.vector(mtcars))
#> [1] FALSE
```

10、See p. 66. When should we use drop = FALSE?

Use `drop = FALSE` if you are subsetting a matrix, array, or data frame and you want to preserve the original dimensions. You should almost always use it when subsetting inside a function.

11、If x is a matrix, what does x[] <- 0 do? How is it different to x <- 0?

If `x` is a matrix, `x[] <- 0` will replace every element with 0, keeping the same number of rows and columns. `x <- 0` completely replaces the matrix with the value 0.

12、How can you use a named vector to relabel categorical variables? Give a specific example to illustrate your method.

A named character vector can act as a simple lookup table: `c(x = 1, y = 2, z = 3)[c("y", "z", "x")]`

13、Why does mtcars[1:20] return an error? How can you fix it?

mtcars[1:20] calls the first 20 positions, but does not specify any columns.

We can use mtcars[1:20,].

14、Implement your own function that extracts the diagonal entries from a matrix (it should behave like diag(x) where x is a matrix).

The elements in the diagonal of a matrix have the same row- and column indices. This characteristic can be used to create a suitable numeric matrix used for subsetting.

```
diag_my <- function(x){
  n <- min(nrow(x), ncol(x))
  idx <- cbind(seq_len(n), seq_len(n))

  x[idx]
}
```

15、What happens to a factor when you modify its levels?

> *f1 <- **factor**(letters) **levels**(f1) <- **rev**(**levels**(f1))*

The underlying integer values stay the same, but the levels are changed, making it look like the data as changed.

```
f1 <- factor(letters[1:10])
levels(f1)
#>  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
f1
#>  [1] a b c d e f g h i j
#> Levels: a b c d e f g h i j
as.integer(f1)
#>  [1]  1  2  3  4  5  6  7  8  9 10

levels(f1) <- rev(levels(f1))
levels(f1)
#>  [1] "j" "i" "h" "g" "f" "e" "d" "c" "b" "a"
f1
#>  [1] j i h g f e d c b a
#> Levels: j i h g f e d c b a
as.integer(f1)
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

16、What does this code do? How do f2 and f3 differ from f1?

> *f2 <- **rev**(**factor**(letters)) f3 <- **factor**(letters, levels = **rev**(letters))*

For `f2` and `f3` either the order of the factor elements *or* its levels are being reversed. For `f1`both transformations are occurring.

17、What makes array(1:5, c(1, 1, 5)) different to 1:5?

array(1:5, c(1, 1, 5)) is to make five 1×11×1 matrix.

18、What attributes does a data frame possess?

We can use `attributes()` to find out the attributes of data frame.

```
names(attributes(data.frame(x = 1:3, y = c("a", "b", "c"))))
## [1] "names"     "class"     "row.names"
```

19、What does as.matrix() do when applied to a data frame with columns of different types?

The numeric values will be coerced to characters.

```
str(as.matrix(data.frame(x = 1:3, y = c("a", "b", "c"))))
##  chr [1:3, 1:2] "1" "2" "3" "a" "b" "c"
##  - attr(*, "dimnames")=List of 2
##   ..$ : NULL
##   ..$ : chr [1:2] "x" "y"
```

If the data frame contain both logical values and numeric value, the logical values will be coerced to numeric values.

```
as.matrix(data.frame(x = c(T, T, F), y = c(1, 2, 3)))
##      x y
## [1,] 1 1
## [2,] 1 2
## [3,] 0 3
```

20、Can you have a data frame with 0 rows? What about 0 columns?

Yes.

```
data.frame(x = character(), y = numeric())
## [1] x y
## <0 rows> (or 0-length row.names)
```

21、Find the Fibonacci sequence below 4000000, and sum the even numbers in your result.

Code:

```
fib=numeric(100000)
fib[1]=1
fib[2]=1
i=3
while(fib[i-1]<4000000){
  fib[i]=fib[i-1]+fib[i-2]
  i=i+1
}
sum=0
i=1
while(fib[i]<4000000){
  print(fib[i])
  if(fib[i]%%2==0){
  sum=sum+fib[i]
  }
  i=i+1
}
print(sum)
```

Result:

```
# Fibonacci sequence below 4000000
[1] 1
[1] 1
[1] 2
[1] 3
[1] 5
[1] 8
[1] 13
[1] 21
[1] 34
[1] 55
[1] 89
[1] 144
[1] 233
[1] 377
[1] 610
[1] 987
[1] 1597
[1] 2584
[1] 4181
[1] 6765
[1] 10946
[1] 17711
[1] 28657
[1] 46368
[1] 75025
[1] 121393
[1] 196418
[1] 317811
[1] 514229
[1] 832040
[1] 1346269
[1] 2178309
[1] 3524578
#sum
[1] 4613732
```

22、Find the largest prime factor of the number 600851475143.

No algorithm has been published that can factor all integers in polynomial time, that is, that can factor a b-bit number n in time O(b^k) for some constant k. Neither the existence nor non-existence of such algorithms has been proved, but it is generally suspected that they do not exist and hence that the problem is not in class P. The problem is clearly in class NP, but it is generally suspected that it is not NP-complete, though this has not been proven.

There are published algorithms that are faster than O((1 + ε)b) for all positive ε, that is, sub-exponential. The published algorithm with best asymptotic running time is the general number field sieve (GNFS), running on a b-bit number n in time:

$$\exp\left(\left(\sqrt[3]{\frac{64}{9}}+o(1)\right)(\ln n)^{\frac{1}{3}}(\ln\ln n)^{\frac{2}{3}}\right).$$

The above statement is to show that it is almost impossible to write an efficient integer factorization program. For this problem, we first get a relatively large prime number table and then try to decompose it.

Fortunately,

600,851,475,143 =71* 839 * 1471 * 6857.

the largest prime factor of the number 600851475143 is 6857.

Code：

```r
#install.packages("pracma")
library(pracma)
p=primes(100000)
a=600851475143
while(a!=1){
  i=1
  while(i<=9592){
    if(a%%p[i]==0){
      a=a/p[i]
      print(p[i])
      break
    }
    i=i+1
  }
}
```