Chapter 5 Markdowns

By @TNTprizz80315.

Programming in R

Special Usage of Functions

User Defined Operator

The operator should both start from and end in `%`.

```r
"%+-*%" <- function(front, back) {
    c(front + back, front - back, front * back)
}
3 %+-*% 2
```

Output:

```
[1] 5 1 6
```

Replacement Functions

It can receive assignments to change certain values in the inputs.
The operator should end in `<-`.

The first parameter should be the value to be modified, while the last parameter would be the value assigned to the function.

```r
"add<-" <- function(x, value) {
    for (i in 1:length(x)) {
        x[i] <- x[i] + value
    }
    return(x)
}
x <- c(1, 2, 3)
add(x) <- 5
x
```

Output:

```
[1] 6 7 8
```

Flexible Number of Arguments

Equivant to `*args`. You can receive flexible number of arguments for processing.

The function argument should be `...` at the end in this case, if you need necessary arguments.

```
lensum <- function(...) {
    x <- list(...)
    sl <- 0
    for (i in x) sl <- sl + length(i)
    return(sl)
}
lensum(1:5, 0:9, sample(1:3))
```

Output:

```
[1] 18
```

Returning Multiple Values

Just return the vector.

```
dummy <- function() {
    return(c(name="Keqing", stars=5, element="Electro"))
}
kq <- dummy()
kq
```

Output:

name	stars	element
Keqing	5	Electro

IO Control

Options

In a R session, we can do global configurations to ensure formatted output.

`options(**kwargs)`

kwargs	default value
add.smooth	TRUE
check.bounds	FALSE

kwargs	default value
continue	"+ "
digits	7
echo	TRUE
encoding	"native.enc"
error	NULL
expressions	5000
keep.source	interactive()
keep.source.pkgs	FALSE
max.print	99999
OutDec	"."
prompt	"> "
scipen	0
show.error.messages	TRUE
timeout	60
verbose	FALSE
warn	0
warning.length	1000
width	80

Most of them are self-explanatory.

Sprintf

You can generate user-friendly text output using `sprintf`.

```
sprintf(str, *args, **kwargs)
```

`str` String. Use like which in C language.
`*args` The variables.

conversion specification	Represent
`%d`	Integer
`%o`	Octal
`%x`	Hexadecimal
`%f`	Float

conversion specification	Represent
`%e`	Scientific notation

for any conversion specification, do use numbers between `%` and the character to adjust the width and decimal points of the value.

```
sprintf("pi is %10.3f", pi)
# Width 10, 3 d.p.
```

Output:

```
[1] "pi is      3.142"
```

cat

Can be used to directly display strings, without the vector indicator.

```
sprintf("114514%d%d\n", 19, 19)
cat(sprintf("114514%d%d\n", 19, 19))
```

Output:

```
[1] "1145141919\n"
1145141919
```

Readline

`readline(prompt)`
Receive user input.

`prompt` String. The message to be displayed before the cursor.

```
ouo <- ""
repeat {
    ouo <- readline("type 'exit' to exit.\n$ ")
    if (ouo == "exit") break
}
```

Output:

```
type 'exit' to exit.
$ e
type 'exit' to exit.
$ exit
```

Vectorize a Function

Consider the factorial function introduced in Ch4:

```
factorial(c(4, 3))
```

Output:

```
Error in if (x == 0) return(1) else return(x * factorial(x - 1)) : 
  the condition has length > 1
```

We can use `vapply` to tackle this problem.

`vapply(input, function, fun.value)`

`input` Vector input. The function will be applied to each values of the vector.
`function` The function to be used.
`fun.value` To verify the output type. Use `sapply` if you don't want this.

```
vapply(seq(1, 10, 1), factorial, double(1))
```

Output:

```
 [1]       1       2       6      24     120     720    5040   40320
362880
[10] 3628800
```

Vector Process

By creating a formula with vector, you can do bulk operations without using loop.

```
i <- 1:100
sum((i**2 + i)/i**(1/2))
```

Output:

```
[1] 41172.69
```