Dataframe 1

```
data.frame(data, row.names = NULL,...)
data.frame(name1 = vector1, name2 = vector2,...)
> vec1<-c( 25, 24, 21, 23, 22)
> vec2<-c("M","F","F","M","F")
> vec3<-c("A1","A2","A3","A1","A2")
> (df1 <- data.frame(age=vec1, sex=vec2, group=vec3))
  age sex group
1 25 M A1
2 24 F A2
3 21 F A3
4 23 M A1
5 22 F A2
```

(**Basics)

Dataframe 2

`is.object(df1)`	TRUE
`typeof(df1)`	"list"
`class(df1)`	"data.frame"
`sloop::otype(df1)`	"S3"

`df1[c(1,2)]`
`df1[[2]]`
`df1[["sex"]]`
`df1$sex`
	Subsetting

(**Basics)

Dataframe 3

Important functionality:

`str(df1)`

`attributes(df1)`

`merge()`

merge by common columns or row names

`cbind() rbind()`

`head()`

visualize the first rows of a df

`attach()`

df can be attached

`summary()`

basic statistics

(**Basics)

Dataframe 4

Dataframes can contain lists:

```
1) df <- data.frame(x=1:3); df$y <- list(1:2,1:3,1:4)
2) df <- data.frame(x = 1:3, y = I(list(1:2, 1:3, 1:4)))
```

Can be also matrices:

```
dfm <- data.frame(x = 1:3 * 10)
dfm$y <- matrix(1:9, nrow = 3)
dfm$z <- data.frame(a = 3:1, b = letters[1:3])
```

(***Adanced)

Tibbles 1

Part of the `tidyverse`

More flexible than datasets:

```
(tib0<-tibble(x = 1:3, y = 1, z = x^2+y,
name = letters[1:3], lis = list(1:5,1:10,1:20)))
```

Can be more rigid: recycling

```
(tib2 <- tibble::tibble(age=1, sex=vec2, group=vec3))
```

(**Basics)

Tibbles 2

`attrib ute s(tib0)`	$names, $row.names, $class
`mode(tib0)`	"list"
`typeof(tib0)`	"list"
`class(tib0)`	"tbl_df"
`sloop: :ot ype - (tib0)`	"S3"
`is_tibble()`	Test if tibble
`as.data.frame()`	Convert to df
`tribble()`	Table format

(**Basics)

Control flow

`if()` statements

```
> if(x > 10) {y<-x ; print(y)}
+ else {y<-x+1 ; print(y)}
```

`switch()` tests an expression against elements of a list.

```
switch(x, "red", "green",
"blue", stop("No"))
```

`case_w hen()`statements

```
> dplyr::case_when(
+ x%%5==0~"a",
+ x>6~"great",
+ TRUE ~ as.character(x))
```

`for` loops

```
> for (i in 1:4) print(i)
```

`next` exits an iteration

`break` exits the loop

`while()` statement

```
> while(x < 4) {print(x); x<-
x+1}
```

`repeat{}` statement

```
repeat {print(x); x<-x+1;
if(x==5) break}
```

(**Basics)(***Advanced)

Functions 1

"Everything that exists is an object. Everything that happens is a function call."

Types of functions:

closure functions, these are the usual functions create or encounter while using R.

builtin functions, primitive/internal functions that generally rely on C code and pass their arguments in an evaluated way.

special functions, primitive/internal functions that rely on C code and pass their arguments in an unevaluated way

The typical syntax for a closure function

```
fname <- function(arg1, arg2,
... ){
body: expressions and return
value }
```

Functions 1 (cont)

```
fname(arg1 = val1, arg2 = val2)
```

Example:

```
> mysum <- function(x,y){x+y
#just make a sum +}
> mysum(1,2)
[1] 3
```

(**Basics)

Functions 2

`is.object(mysum)`	FALSE
`typeof(mysum)`	"closure"
`class(mysum)`	"function"
`sloop::otype(mysum)`	"base"
`mysum`	`function(x,y) {x+y}` prints
`typeof(sum)`	"builtin" functions
`mean(2,3,10) != mean(c(2,3,10))`	2 != 5

(**Basics)(***Advanced)

Functions 2

`is.obj ect - (mysum)`	FALSE
`typeof (mysum)`	"closure"
`class( mysum)`	"function"
`sloop: :ot - ype (mysum)`	"base"
`mysum`	`functi on( - x,y ){x+y}` prints
typeof(sum)	"builtin" functions
`mean(2 ,3,10) != mean(c (2, - 3,10))`	2 != 5

(**Basics)(***Advanced)

Functions 3

Make lists of functions:

```
> funs <- list(
+ half = function(x) x / 2,
+ double = function(x) x * 2 +)
> funs$double(10); funs$half(10)
```

Arguments are evaluated only if accessed

```
> f<- function(x,y) x+1 #y is
ignored
> f(2)
[1] 3
```

Arguments can be defined later

```
> f<- function(x,y) {y<-2;x+y}
> f(2)
[1] 4
```

Arguments can be forced

```
> f <- function(x) {
+ force(x)
+ 10
+}
> f(stop("This is an error!"))
Error in force(x) : This is an
error!
```

A promise is forced when its value is needed

```
> f<-function(x, label =
deparse(x)) {
+ label #this forces the promise
+ x <- x + 1 # change in x does
not affect promise
+ print(label)}
> f1<-function(x, label =
deparse(x)) {
+ x <- x + 1 #the change in x
will affect the label promise
+ print(label)}
> f(1)
[1] "1"
> f1(1)
[1] "2"
```

(***Advanced)