functions

Peilin Chen

11/14/2017

basic structure

*Functions can be passed as arguments to other functions.

*Functions can be nested, so that you can define a function inside of another function.

*The return value of a function is the last expression in the function body to be evaluated.

```
f <- function(<arguments>) {
## Do something interesting
}
```

See the input of a function using args(function name) For example lm() is fitting a linear model

```
args(lm)
```

```
## function (formula, data, subset, weights, na.action, method = "qr",
##     model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
##     contrasts = NULL, offset, ...)
## NULL
```

```
f<-function(x){x+1}
f(1)
```

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

```
f(10)
```

```
## [1] 11
```

In addition to not specifying a default value, you can also set an argument value to NULL.

```
f <- function(a, b = 1, c = 2, d = NULL) {
  a*b+c
}
f(0)
```

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

```
f(a=0,c=3) #update the default value
```

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

Missing input will yield error

```
f <- function(a, b) {
print(a)
print(b)
}
```

The ... argument is also necessary when the number of arguments passed to the function cannot be known in advance

```
f <- function(a,b,c) {
  a=a
  b=b
  rest=c
  print(a+b+sum(c))
}
f(1,2,c(3,4,5))
```

```
## [1] 15
```

Build in functions to see the args

```
args(functionname)
```

For example: see the input arguments in lm()

```
args(lm)
```

```
## function (formula, data, subset, weights, na.action, method = "qr",
##     model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
##     contrasts = NULL, offset, ...)
## NULL
```

Functional programming concentrates on four constructs:

Data (numbers, strings, etc) Variables (function arguments) *Functions* Function Applications (evaluating functions given arguments and/or data)

Core Functional Programming Funcitons: Map, reduce, search, filter

The function Map allows the mapping from one vector to another using a map function, which can be specified by lambda.

```
x=1:20
Map({function (a) a*2}, x)
```

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

```
## [[3]]
## [1] 6
##
## [[4]]
## [1] 8
##
## [[5]]
## [1] 10
##
## [[6]]
## [1] 12
##
## [[7]]
## [1] 14
##
## [[8]]
## [1] 16
##
## [[9]]
## [1] 18
##
## [[10]]
## [1] 20
##
## [[11]]
## [1] 22
##
## [[12]]
## [1] 24
##
## [[13]]
## [1] 26
##
## [[14]]
## [1] 28
##
## [[15]]
## [1] 30
##
## [[16]]
## [1] 32
##
## [[17]]
## [1] 34
##
## [[18]]
## [1] 36
##
## [[19]]
## [1] 38
##
## [[20]]
## [1] 40
```

The function Reduce will perform the function on a list of vectors one by one, and finally return a single value.

```
x=1:5
x
```

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

```
Reduce(function (x, y) x+y, x)
```

```
## [1] 15
```

The function Filter will remove all elements when they do not satisfy the condition

```
x=1:10
Filter(function (x) x%%2==0, x)
```

```
## [1]  2  4  6  8 10
```

```
require(pryr)
x<-3
where("x")  #get the enviroment of x
```

```
## <environment: R_GlobalEnv>
```

```
f<-function(x){
 print(environment(x))
  return(x+1)
  }
f(x=5)
```

```
## NULL
```

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

```
x
```

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

```
environment(f)
```

```
## <environment: R_GlobalEnv>
```