Presentación

Programación en Julia: Primeros pasos

2. Funciones

Benjamín Pérez

Héctor Medel

Definiendo funciones

Una función es un objeto que toma como *input* algunos argumentos, les hace operaciones (cualquiera permitida) y regresa valores.

Nota: Pueden ser de distintos tipos y estructuras. Los argumentos pueden estar expresados por tipo (opcional pero recomendado); los tipos pueden ser definidos por uno mismo.

La sintaxis general de una función es la siguiente:

```
function funcname(argumentos)
    #Something
    return values
end
```

Ejemplo 1: Función sencilla

```
mult (generic function with 1 method)
```

```
function mult(x,y)
    println("x es $x, y es $y")
    return x*y
end

```

Nota:
El comando `return` es opcional para este ejemplo ya que tiene sóla operación

Ejemplo 2: Condicional

```
let
function mult(x,y)
    println("x es $x, y es $y")
    if x==1
        return y
    end
    x*y
end
    n = mult(1,4)
end
```

```
x es 1, y es 4
```

Ejemplo 3: Varios valores de salida

```
(16, 1, 0)
```

```
let
    function multi(n,m)
        n*m, div(n,n), n%m
    end

    multi(8,2)
end
```

Aquí la función entrega una tupla de valores (pueden ser de diferentes tipos)

```
md"Aquí la función entrega una tupla de valores (pueden ser de diferentes tipos)"
```

Ejemplo 4: Puntos suspensivos

Se pueden tener varios input sin definir todos (sin definir necesariamente sus tipos) usando los puntos suspensivos

```
function varargs(n,m,args...)
println("argumentos: $n,$m,$args")
end
```

```
let
    function varargs2(args...)
        println("argumentos: $args")
    end
    x = (3,4,5)
    varargs2(1,2,x)
    y = [6,7,8,9]
    varargs2(1,2,x,y)
end
```

```
argumentos: (1, 2, (3, 4, 5))
argumentos: (1, 2, (3, 4, 5), [6, 7, 8, 9])
```

Ejemplo 5: Tipos de variable definidos

Para optimizar código es conveniente restringir los parámetros de la función.

```
function mult(x::Float64,y::Float64)
    x*y
end
```

Nota: si la función tiene argumentos definidos no aceptará de otro tipo diferente al ya establecido

```
15.0
```

```
let
    function multi(x::T,y::T) where T<:Float64
        x*y
    end

    x = 3.0
    y = 2
    z = 5.0

    #multi(x,y)
    multi(x,z)
end
```

Ejemplo 6: Funciones como expresiones matemáticas

```
44.2
```

```
let
    f(x,y) = x^3 -x*y + 1/y;
    f(4,5)
end
```

Argumentos opcionales en las funciones

Se tiene la opción de definir funciones con valores pre establecidos

```
function pref(a,b=2;k="ABC")
    a + b
end
```

Esta función contiene argumentos opcionales en posición y palabras clave (keyword) opcionales.

Ejemplo 1

```
let
    function allargs(arg_normal, arg_pos_opt=2;arg_clave = "A")
        println("argumento normal = $arg_normal")
        println("argumento opcional = $arg_pos_opt")
        println("argumento clave = $arg_clave")
    end

    #allargs(1,3,arg_clave=4)
    #allargs(1,3)
    allargs(5)
end
```

```
argumento normal = 5
argumento opcional = 2
argumento clave = A
```

Ejemplo 2: Puntos suspensivos

```
Pairs(:k1 => "nombre1", :k2 => "nombre2", :k3 => 7)
```

```
let
    function varargs2(;args...)
        args
    end

    varargs2(k1="nombre1",k2="nombre2",k3=7)
end
```

Funciones anónimas

- Se pueden definir funciones sin nombre

```
function (x)
    x + 2
end
```

- Funciones Lambda

```
(x) -> x + 2
```

- Funciones Lambda

```
x -> x + 2
```

Funciones de funciones

Una función puede tomar una función como argumento

```
function ff(f::Function,x::Float)
    #Operaciones con f(x)
end
```

Ejemplo 1: Función de función

```
38.000000000000256
```

```
let
    function derivada(f::Function,x::Float64,dx::Float64=0.001)
        df = (f(x+dx) - f(x-dx))/(2*dx)
        return df
    end

    f = x-> 2*x^2 + 30*x + 9;

    derivada(f,2.0,0.01)
end
```

Ejemplo 2: Funciones anidadas

2.21

```
let
    function a(x)
        z = x^2
        function b(z)
            z += 1
        end
        return b(z)
    end

    # a(10)
    a(1.1)
end
```

Broadcasting

Las funciones puedes ser *transmitidas* sobre elementos de un arreglo y hacer operaciones sobre cada uno de ellos. Se usa el operador *punto*:

```
f.(arreglo)
```

Ejemplos

```
3×2 Matrix{Int64}:
  4   9
 16  25
 36  49
```

```
let
    function cuad(X)
        cX = X*X
    end

    #x = 2;
    #x = rand(1:10,2);
    x = [2 3; 4 5; 6 7]
    println("x = $x")
    cuad.(x)
end
```

```
x = [2 3; 4 5; 6 7]
```

Map y filter

Estas funciones de funciones, por decirlo de alguna forma, son muy útiles cuando se trabaja con colecciones de datos. Su sintaxis regularmente es:

```
map(func, collect)
```

donde `func` es una función que deseamos que acutúe sobre una colección de datos.

Ejemplo 1: mapeo de una función sobre vector

```
[40, 50, 60]
```

```
let
map(x-> x*10,[4,5,6])
end
```

```
[1, 8, 27, 64, 125]
```

```
let
    cubos = map(x->x^3,collect(1:5))
end
```

Ejemplo 2: Mapeo con más instrucciones

A veces se necesitan varias instrucciones para ejecutar sobre un arreglo. PAra eso se puede usar `begin` y `end`.

```
[1, 2, 1, 0, 1, 2, 1]
```

```
let
    map(x -> begin
        if x == 0 return 0
        elseif iseven(x) return 2
        elseif isodd(x) return 1
        end
    end,collect(-3:3))
end
```

Ejemplo 3: Comando `do`

El comando `do` crea una función anónima con argumento `x` y pasa el argumento al comando `map`.

```
[1, 2, 1, 0, 1, 2, 1]
```

```
let
    map(collect(-3:3)) do x
        if x == 0  return 0
        elseif iseven(x) return 2
        elseif isodd(x) return 1
        end
    end
end
```