

Programming

optimisation and operations research algorithms with Julia

for Business Tasks

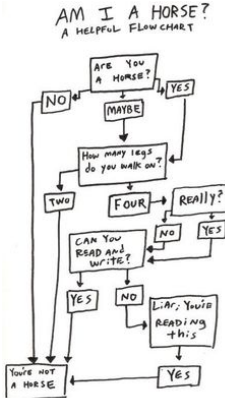
Prof. Dr. Xavier Gandibleux

Université de Nantes
Département Informatique – UFR Sciences et Techniques
France

Lesson 5 – May-June 2022

Control flow

The conditionals



if ... endif

Definition:

```
if condition  
    instruction(s)  
end
```

Example:

```
julia> if zipcode == 4020  
    println("Welcome to Linz")  
end
```

if ... endif

Definition:

```
if condition  
    instruction(s)  
end
```

Example:

```
julia> if zipcode == 4020  
    println("Welcome to Linz")  
end
```

if ... else ... endif (1/2)

```
if condition
    instruction(s) 1
else
    instruction(s) 2
end
```

```
julia> if zipcode == 4020
    println("Welcome to Linz")
else
    println("Welcome to Austria")
end
```

if ... else ... endif (2/2)

ifelse instruction:

```
ifelse( condition, case_true, case_false )
```

```
julia> println("Welcome to ",  
              ifelse(zipcode == 4020, "Linz", "Austria")  
              )
```

Ternary operator:

```
condition ? case_true : case_false
```

```
julia> println("Welcome to ",  
              zipcode == 4020 ? "Linz" : "Austria")  
              )
```

if ... else ... endif (2/2)

ifelse instruction:

```
ifelse( condition, case_true, case_false )
```

```
julia> println("Welcome to ",  
              ifelse(zipcode == 4020, "Linz", "Austria")  
              )
```

Ternary operator:

```
condition ? case_true : case_false
```

```
julia> println("Welcome to ",  
              zipcode == 4020 ? "Linz" : "Austria")  
              )
```

if ... elsif ... [else ...] endif

```
if condition 1
    instruction(s) 1
elsif condition 2
    instruction(s) 2
else
    instruction(s) n+1
end
```

```
julia> if zipcode == 4020
    println("Welcome to Linz Central area")
elseif zipcode == 4030
    println("Welcome to Linz South area")
elseif zipcode == 4040
    println("Welcome to Linz North area")
else
    println("Welcome to Austria")
end
```


Logical operators

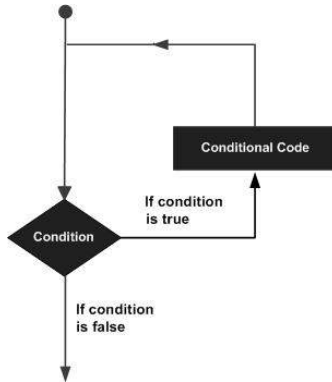
With a and b, two logical conditions:

<i>Operator</i>	<i>Expression</i>	<i>Signification</i>
!	!a	NOT a
&&	a && b	a AND (then) b
	a b	a OR (else) b

Example:

```
julia> countrycode == "AT" && zipcode == 4020
```

Control flow Loops



while ... endWhile

Definition:

```
while condition  
    instruction(s)  
end
```

Example:

```
julia> zipcode = 4020  
while zipcode <= 4040  
    print(zipcode, " ")  
    zipcode = zipcode + 10  
end
```

while ... endWhile

Definition:

```
while condition  
    instruction(s)  
end
```

Example:

```
julia> zipcode = 4020  
while zipcode <= 4040  
    print(zipcode, " ")  
    zipcode = zipcode + 10  
end
```

for ... endFor (1/3)

Definition:

```
for variable in collection  
    instruction(s)  
end
```

Also:

```
for variable = collection  
for variable ∈ collection
```

Collection:

- ▶ range: *start:stop* or *start:step:stop*
- ▶ string: "*characters*"
- ▶ tuple: (*val*₁, *val*₂, ..., *val*_{*n*})
- ▶ array: [*val*₁, *val*₂, ..., *val*_{*n*}]
- ▶ set: Set([*val*₁, *val*₂, ..., *val*_{*n*}])
- ▶ dict: Dict(*key*₁=>*val*₁, *key*₂=>*val*₂, ..., *key*_{*n*}=>*val*_{*n*})

for ... endFor (1/3)

Definition:

```
for variable in collection  
    instruction(s)  
end
```

Also:

```
for variable = collection  
for variable ∈ collection
```

Collection:

- ▶ range: *start:stop* or *start:step:stop*
- ▶ string: "*characters*"
- ▶ tuple: (*val*₁, *val*₂, ..., *val*_{*n*})
- ▶ array: [*val*₁, *val*₂, ..., *val*_{*n*}]
- ▶ set: Set([*val*₁, *val*₂, ..., *val*_{*n*}])
- ▶ dict: Dict(*key*₁=>*val*₁, *key*₂=>*val*₂, ..., *key*_{*n*}=>*val*_{*n*})

for ... endFor (1/3)

Definition:

```
for variable in collection  
    instruction(s)  
end
```

Also:

```
for variable = collection  
for variable ∈ collection
```

Collection:

- ▶ range: *start:stop* or *start:step:stop*
- ▶ string: "*characters*"
- ▶ tuple: (*val*₁, *val*₂, ..., *val*_{*n*})
- ▶ array: [*val*₁, *val*₂, ..., *val*_{*n*}]
- ▶ set: Set([*val*₁, *val*₂, ..., *val*_{*n*}])
- ▶ dict: Dict(*key*₁=>*val*₁, *key*₂=>*val*₂, ..., *key*_{*n*}=>*val*_{*n*})

for ... endFor (1/3)

Definition:

```
for variable in collection  
    instruction(s)  
end
```

Also:

```
for variable = collection  
for variable ∈ collection
```

Collection:

- ▶ range: *start:stop* or *start:step:stop*
- ▶ string: "*characters*"
- ▶ tuple: (*val*₁, *val*₂, ..., *val*_{*n*})
- ▶ array: [*val*₁, *val*₂, ..., *val*_{*n*}]
- ▶ set: Set([*val*₁, *val*₂, ..., *val*_{*n*}])
- ▶ dict: Dict(*key*₁=>*val*₁, *key*₂=>*val*₂, ..., *key*_{*n*}=>*val*_{*n*})

for ... endFor (2/3)

Examples:

```
julia> for i in 1:10
        print(i , " ")
      end
```

```
julia> for i in 1:2:10
        print(i , " ")
      end
```

```
julia> for i in "Linz"
        print(i , " ")
      end
```

for ... endFor (3/3)

```
julia> for i in (4020,4030,4040)
        print(i , " ")
    end
```

```
julia> for i in [4020,4030,4040]
        print(i , " ")
    end
```

```
julia> for i in Set([4020,4030,4040])
        print(i , " ")
    end
```

```
julia> for i in Dict("Center"=>4020,"South"=>4030,
                    "North"=>4040)
        print(i , " ")
    end
```

Multiple for ... endFor

```
for var1 in collection1  
  for var2 in collection2  
    instruction(s)  
  end  
end
```

```
for var1 in collection1, var2 in collection2  
  instruction(s)  
end
```

Example:

```
julia> for i in 1:3, j in "hello"  
    println(i, " ", j)  
end
```

Multiple for ... endFor

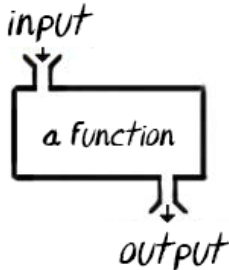
```
for var1 in collection1  
    for var2 in collection2  
        instruction(s)  
    end  
end
```

```
for var1 in collection1, var2 in collection2  
    instruction(s)  
end
```

Example:

```
julia> for i in 1:3, j in "hello"  
        println(i, " ", j)  
    end
```

Control flow Functions



Declaring a function

Julia gives us different ways to write a function:

- ▶ A single expression function
- ▶ An anonymous function
- ▶ A general function

Declaring and calling a single expression function

Function in a single line

Example of declaration:

```
julia> f(x) = x^2 + 7
```

Example of call:

```
julia> f(2)
```

Declaring and calling a single expression function

Function in a single line

Example of declaration:

```
julia> f(x) = x^2 + 7
```

Example of call:

```
julia> f(2)
```


Declaring/calling an anonymous function

No named function

Example over a scalar:

```
julia> map(x -> x^2 + 7 , 2)
```

Example over a vector:

```
julia> map(x -> x^2 + 7 , [2, 7, 4])
```

Example with multiple parameters:

```
julia> map((x,b) -> x^2 + b , 2, 7)
```

Declaring/calling an anonymous function

No named function

Example over a scalar:

```
julia> map(x -> x^2 + 7 , 2)
```

Example over a vector:

```
julia> map(x -> x^2 + 7 , [2, 7, 4])
```

Example with multiple parameters:

```
julia> map((x,b) -> x^2 + b , 2, 7)
```

Declaring/calling an anonymous function

No named function

Example over a scalar:

```
julia> map(x -> x^2 + 7 , 2)
```

Example over a vector:

```
julia> map(x -> x^2 + 7 , [2, 7, 4])
```

Example with multiple parameters:

```
julia> map((x,b) -> x^2 + b , 2, 7)
```

Declaring and calling a general function (1/6)

With a single parameter

Example of declaration:

```
julia> function affinefct(x)
    y = x^2 + 7
    return y
end
```

Example of call:

```
julia> affinefct(2)
```

Declaring and calling a general function (1/6)

With a single parameter

Example of declaration:

```
julia> function affinefct(x)
    y = x^2 + 7
    return y
end
```

Example of call:

```
julia> affinefct(2)
```

Declaring and calling a general function (2/6)

With multiple parameters

Example of declaration:

```
julia> function affinefct(x,b)
    y = x^2 + b
    return y
end
```

Example of call:

```
julia> affinefct(2,7)
```

Declaring and calling a general function (2/6)

With multiple parameters

Example of declaration:

```
julia> function affinefct(x,b)
    y = x^2 + b
    return y
end
```

Example of call:

```
julia> affinefct(2,7)
```

Declaring and calling a general function (3/6)

When the type of parameters is specified

Example of declaration:

```
julia> function affinefct(x::Int64,b::Int64)
        y = x^2 + b
        return y
    end
```

Example of call:

```
julia> affinefct(2,7)
```


Declaring and calling a general function (3/6)

When the type of parameters is specified

Example of declaration:

```
julia> function affinefct(x::Int64,b::Int64)
    y = x^2 + b
    return y
end
```

Example of call:

```
julia> affinefct(2,7)
```

Declaring and calling a general function (4/6)

With optional arguments

Example of declaration:

```
julia> function affinefct(x,b=3)
    y = x^2 + b
    return y
end
```

Examples of call:

```
julia> affinefct(2)
```

```
julia> affinefct(2,7)
```

Declaring and calling a general function (4/6)

With optional arguments

Example of declaration:

```
julia> function affinefct(x,b=3)
    y = x^2 + b
    return y
end
```

Examples of call:

```
julia> affinefct(2)
```

```
julia> affinefct(2,7)
```

Declaring and calling a general function (5/6)

The return Keyword

- ▶ return a single value

```
return expression
```

- ▶ return several values

```
return expression1, expression2 . . . expressionn
```

- ▶ return no value

```
return nothing
```

Declaring and calling a general function (6/6)

Mutable and immutable objects (see `ismutable` function)

The following type are immutable:

- ▶ integer
- ▶ float
- ▶ boolean
- ▶ character
- ▶ tuple

If a function has a parameter with this type, modifying the variable inside the function didn't modify the value outside the function.

The following type are mutable:

- ▶ array

If a function has a parameter with this type, modifying the variable inside the function changes the value outside of the function.

By convention, **functions followed by ! alter their contents.**

Declaring and calling a general function (6/6)

Mutable and immutable objects (see `ismutable` function)

The following type are immutable:

- ▶ integer
- ▶ float
- ▶ boolean
- ▶ character
- ▶ tuple

If a function has a parameter with this type, modifying the variable inside the function didn't modify the value outside the function.

The following type are mutable:

- ▶ array

If a function has a parameter with this type, modifying the variable inside the function changes the value outside of the function.

By convention, **functions followed by ! alter their contents.**

Broadcast

```
julia> function add1(x::Int64)
           return x+1
       end
```

```
julia> s=1
julia> add1(s)
```

```
julia> a=[1,2,3]
julia> add1(a)                                     # error
```

Broadcast

```
julia> function add1(x::Int64)
    return x+1
end
```

```
julia> s=1
julia> add1(s)
```

```
julia> a=[1,2,3]
julia> add1(a) # error
```


Broadcast

"broadcast" a function to work over each elements of an array:

1)

```
broadcast(funcName, elements)
```

```
julia> broadcast(add1, a)
```

2)

```
funcName.(elements)
```

```
julia> add1.(a)
```

Broadcast

"broadcast" a function to work over each elements of an array:

1)

```
broadcast(funcName, elements)
```

```
julia> broadcast(add1, a)
```

2)

```
funcName.(elements)
```

```
julia> add1.(a)
```

Broadcast

"broadcast" a function to work over each elements of an array:

1)

```
broadcast(funcName, elements)
```

```
julia> broadcast(add1, a)
```

2)

```
funcName.(elements)
```

```
julia> add1.(a)
```

Broadcast

"broadcast" a function to work over each elements of an array:

1)

```
broadcast(funcName, elements)
```

```
julia> broadcast(add1, a)
```

2)

```
funcName.(elements)
```

```
julia> add1.(a)
```

Multiple-dispatch (1/2)

- ▶ The same function can be defined with different number and type of parameters; example:

```
julia> brot(x::Int64)= println("Int:  ",x)
julia> brot(x::Float64)= println("Flt:  ",x)
julia> brot(x::Bool)= println("Bool:  ",x)
julia> brot(x)= println("Others:  ",x)
```

- ▶ These different versions are named **methods** in Julia
- ▶ Inspect the methods of a function with

```
methods(funcName)
```

- ▶ When calling such functions, Julia will pick up the correct one depending from the parameters in the call (by default the stricter version).

Multiple-dispatch (1/2)

- ▶ The same function can be defined with different number and type of parameters; example:

```
julia> brot(x::Int64)= println("Int:  ",x)
julia> brot(x::Float64)= println("Flt:  ",x)
julia> brot(x::Bool)= println("Bool:  ",x)
julia> brot(x)= println("Others:  ",x)
```

- ▶ These different versions are named **methods** in Julia
- ▶ Inspect the methods of a function with

```
methods(funcName)
```

- ▶ When calling such functions, Julia will pick up the correct one depending from the parameters in the call (by default the stricter version).

Multiple-dispatch (1/2)

Example:

```
julia> bro1(8.3)
julia> bro1('c')
julia> bro1(3)
julia> bro1(true)
julia> bro1("hello")
```

```
julia> methods(bro1)
```

Multiple-dispatch (1/2)

Example:

```
julia> bro1(8.3)
julia> bro1('c')
julia> bro1(3)
julia> bro1(true)
julia> bro1("hello")
```

```
julia> methods(bro1)
```


Review and exercises

(notebook)

