Programming

Optimisation and Operations Research algorithms

with Julia

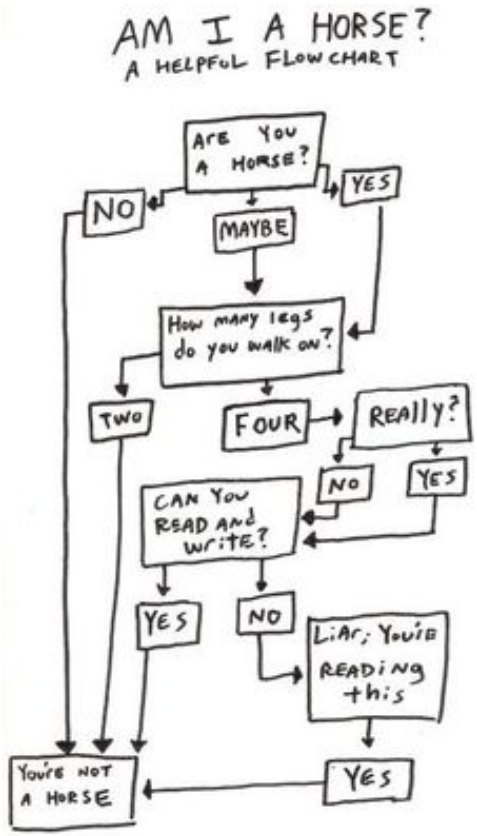
Prof. Dr. Xavier Gandibleux

Université de Nantes
Département Informatique – UFR Sciences et Techniques
France

Lesson 4 – May-June 2021

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")
       elsif zipcode == 4030
          println("Welcome to Linz South area")
       elsif zipcode == 4040
          println("Welcome to Linz North area")
       else
          println("Welcome to Austria")
       end
```

Logical operators

With `a` and `b`, two logical conditions:

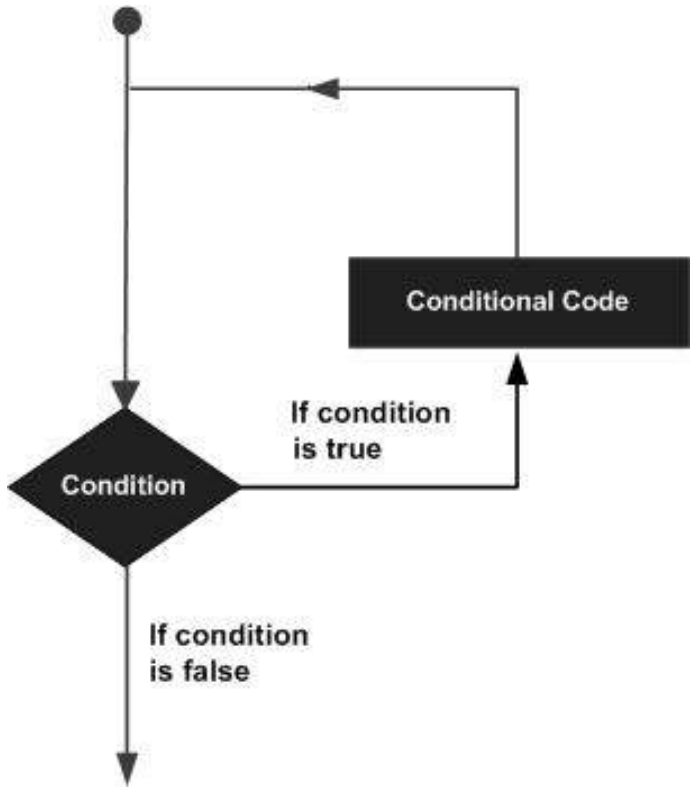
Operator	*Expression*	*Signification*
`!`	`!a`	NOT a
`&&`	`a && b`	a AND (then) b
`||`	`a || b`	a OR (else) b

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: $(val_1, val_2, \ldots, val_n)$
- array: $[val_1, val_2, \ldots, val_n]$
- set: $\texttt{Set}([val_1, val_2, \ldots, val_n])$
- dict: $\texttt{Dict}(key_1 \texttt{=>} val_1, key_2 \texttt{=>} val_2, \ldots, key_n \texttt{=>} val_n)$

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: $(val_1, val_2, \ldots, val_n)$
- array: $[val_1, val_2, \ldots, val_n]$
- set: $\texttt{Set}([val_1, val_2, \ldots, val_n])$
- dict: $\texttt{Dict}(key_1 \texttt{=>} val_1, key_2 \texttt{=>} val_2, \ldots, key_n \texttt{=>} val_n)$

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: `(val1, val2, ..., valn)`
- array: `[val1, val2, ..., valn]`
- set: `Set([val1, val2, ..., valn])`
- dict: `Dict(key1=>val1,key2=>val2, ..., keyn=>valn)`

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
```

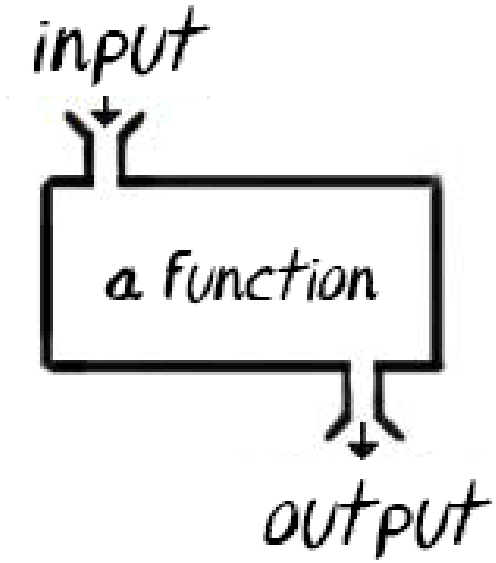
```
for var1 in collection1, var2 in collection2
   instruction(s)
end
```

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 expression_1 , expression_2 ... expression_n
```

- return none 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
- string

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
- string

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

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

```
funcName.(parameters)
```

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

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

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

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

Broadcast

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

```
funcName.(parameters)
```

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

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

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

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

Broadcast

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

```
funcName.(parameters)
```

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

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

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

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

Broadcast

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

```
funcName.(parameters)
```

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

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

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

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

Multiple-dispatch (1/2)

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

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

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

```
methods(functName)
```

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

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

```
methods(functName)
```

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

```
julia> methods(brol)
```

Multiple-dispatch (1/2)

Example:

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

```
julia> methods(brol)
```

A selection of usual functions (1/4)

Arithmetic functions

With `x` and `y`, two variables; `n` and `p`, two integer variables:

Function	*Description*
`div(x,y)`	truncated division; quotient rounded towards zero
`mod(x,y)`	modulus
`abs(x)`	a positive value with the magnitude of x
`sign(x)`	indicates the sign of x, returning -1, 0, or +1
`sqrt(x)`	square root of x
`exp(x)`	e^x
`exp2(x)`	2^x
`log(x)`	natural logarithm of x
`log(b,x)`	base b logarithm of x
`factorial(n)`	$n!$
`binomial(n,p)`	the binomial coefficient

A selection of usual functions (2/4)

Rounding functions

With `x`, a variable and `T`, a numeric type:

Function	*Description*	*Return type*
`round(x)`	round x to the nearest integer	`typeof(x)`
`round(T,x)`	round x to the nearest integer	`T`
`floor(x)`	round x towards -Inf	`typeof(x)`
`floor(T,x)`	round x towards -Inf	`T`
`ceil(x)`	round x towards +Inf	`typeof(x)`
`ceil(T,x)`	round x towards +Inf	`T`
`trunc(x)`	round x towards zero	`typeof(x)`
`trunc(T,x)`	round x towards zero	`T`

A selection of usual functions (3/4)

Trigonometric functions

With `x`, a variable:

Function	*Description*
`sin(x)`	Compute sine of x, where x is in radians
`cos(x)`	Compute cosine of x, where x is in radians
`tan(x)`	Compute tangent of x, where x is in radians
`sind(x)`	Compute sine of x, where x is in degrees
`cosd(x)`	Compute cosine of x, where x is in degrees
`tand(x)`	Compute tangent of x, where x is in degrees
`deg2rad(x)`	Convert x from degrees to radians
`rad2deg(x)`	Convert x from radians to degrees

A selection of usual functions (4/4)

functions on strings

With `c`, a character, `s`,`s2`, strings, :

Function	*Description*
`reverse(s)`	Reverse a string
`occursin(c,s)`	Determine if `c` is a substring of `s`
`split(s,c)`	Split `s` into an array of substrings on occurrences of `c`
`endswith(s,s2)`	Return true if `s` ends with `s2`
`uppercase(s)`	Return `s` with all characters converted to uppercase
`textwidth(s)`	Give the number of columns needed to print `s`
`isdigit(c)`	Test if `c` is a decimal digit (0-9)
`isletter(c)`	Test if `c` is a letter

Review and exercises

(notebook)