Getting started with Julia

Michiel Stock Bram De Jaegher Daan?

December 2019

1 Basic computing

Let's get started with the basics. Some mathematical operations,

```
1 + 2       # adding integers

1.0 + 2.0   # adding floats

2 / 4       # standard division

div(2, 4)   # Computes 2/4 truncated to an integer

35 \ 7      # inverse division

1 // 3      # fractions

1//2 + 1//4

'c'         # characters (unicode)

:symbol     # symbols, mostly used for macros
```

```
Error: type QuoteNode has no field head
```

variable assignment,

```
x = 2

τ = 1 / 37  # unicode variable names are allowed
```

```
0.02702702702702703
```

unicode! In most Julia editing environments, unicode math symbols can be typed when starting with a "\ and hitting [TAB].

```
# type \alpha  and <TAB>
```

Operators are overrated.

```
5x          # This works
```

```
10
```

But strings are quite essential,

```
mystery = "life, the universe and everything"
```

```
"life, the universe and everything"
```

and string interpolation is performed with $.

```
println("The answer to $mystery is $(3*2*7)")
```

```
The answer to life, the universe and everything is 42
```

All binary arithmetic and bitwise operators have an updating version that assigns the result of the operation back into the left operand. The updating version of the binary operator is formed by placing a, =, immediately after the operator.

```
x += 2  # inplace update of x
x += 2  # inplace update of x
```

```
6
```

2 Boolean operators

From zero to one.

```
# Boolean operators
!true   # => false
!false  # => true
1 == 1  # => true
2 == 1  # => false
1 != 1  # => false
2 != 1  # => true
1 < 10  # => true
1 > 10  # => false
2 <= 2  # => true
2 >= 2  # => true
# Comparisons can be chained
1 < 2 < 3  # => true
2 < 3 < 2  # => false
```

```
false
```

3 Control flow

The `if`, `else`, `elseif`-statement is instrumental to any programming language,

```
if 4 > 3
  println("A")
elseif 3 > 4
  println("B")
else
  println("C")
end
```

```
A
```

Julia allows for some very condense control flow structures.

```
y = condition ? valueiftrue : valueiffalse
```

```
Error: UndefVarError: condition not defined
```

4 Looping

```
characters = ["Harry", "Ron", "Hermione"]

for char in characters
  println("Character $char")
end
```

```
Character Harry
Character Ron
Character Hermione
```

```
for (i, char) in enumerate(characters)
  println("$i. $char")
end
```

```
1. Harry
2. Ron
3. Hermione
```

```
pets = ["Hedwig", "Pig", "Crookhanks"]

for (char, pet) in zip(characters, pets)
  println("$char has $pet as a pet")
end
```

```
Harry has Hedwig as a pet
Ron has Pig as a pet
Hermione has Crookhanks as a pet
```

```
n = 1675767616;

while n > 1
  println(n)
  if n % 2 == 0
    global n = div(n, 2)
  else
    global n = 3n + 1
  end
end
```

5 Functions

Julia puts the fun in functions. User-defined functions can be declared as follows,

```
function square(x)
  result = x * x
  return result
end

square(2)
```

```
square(2.0)
```

```
square("ni")   # the multiplication of strings is defined as a concatenation
```

```
"nini"
```

A more condensed version of of `square(x)`.

```
s(x) = x * x
```

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

Passing an array to a function that takes a single element as argument takes a special syntax. By putting a `.` before the brackets, the function is executed on all the elements of the Array. More on this in **Part3?: collections**.

```
s([1, 2, 3, 4, 5])   # Multiplication is not defined for Arrays
```

```
s.([1, 2, 3, 4, 5])  # This is an elements-wise execution of s()
```

```
5-element Array{Int64,1}:
  1
  4
  9
 16
 25
```

Keyword arguments are defined using a semicolon in the back signature and a default value can be assigned. "Keywords" assigned before the semicolon are default values but their keywords are not ignored.

```
safelog(x, offset=0.1; base=10) = log(x + offset) / log(base)
```

```
safelog(0)
```

```
safelog(0, 0.01)
```

```
safelog(0, 0.01, base=2)
```

```
-6.643856189774724
```

When unsure of what a function does, the documentation can be viewed by adding a "?" in front of the function.

```
# type ?sort and hit <ENTER>
```

For a lot of standard Julia functions a in-place version is defined. In-place means that the function changes one of the input arguments of the function. As an example, `sort()` sorts and returns the array passed as argument, it does not change the original array. In contrast, `sort!()` is the inplace version of sort and directly sorts the array passed as argument.

```
my_unsorted_list = [4, 5, 9, 7, 1, 9]
```

```
sort(my_unsorted_list)
```

```
my_unsorted_list
```

```
6-element Array{Int64,1}:
 4
 5
 9
 7
 1
 9
```

```
sort!(my_unsorted_list)
```

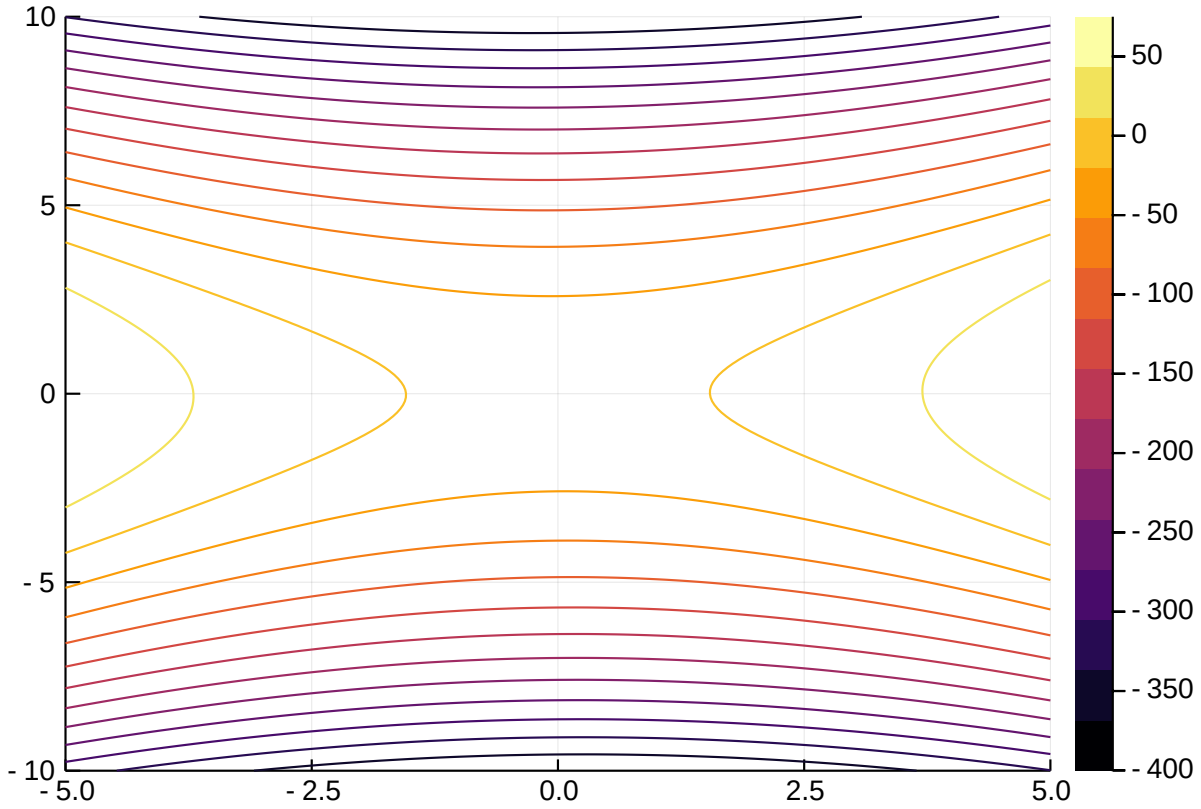
```
my_unsorted_list
```

```
6-element Array{Int64,1}:
 1
 4
 5
 7
 9
 9
```

6 Plotting

Quite essential for scientific programming is the visualisation of the results. `Plots` is the Julia package that handles a lot of the visualisation. `rand(10)`, returns an array of 10 random floats between 0 and 1.

```
using Plots

plot(1:10, rand(10), label="first")
plot!(1:10, rand(10), label="second")

scatter!([1:10], randn(10), label="scatter")

xlabel!("x")
ylabel!("f(x)")
title!("My pretty Julia plot")

plot(0:0.1:10, x -> sin(x) / x, xlabel="x", ylabel="sin(x)/x", legend=:none)

contour(-5:0.1:5, -10:0.1:10, (x, y) -> 3x^2-4y^2 + x*y/6)
```



7 Exercise