Introduction to Julia

Gang Liu, Hartmut Schmider
gang.liu@queensu.ca, hartmut.schmider@queensu.ca
CAC, Queen's University

Summer School, Compute Ontario
2017

Overview

- Some backgrounds
- Starting with Julia's shell
- Basic types, variables, operations
- Conditional branches
- Repeating constructs
- Functions
- Application example: π calculation of many digits
- Collection types and user defined types
- Input, output, and external files
- Type hierarchy, immutability, and parametric types
- Variable scopes, modules, and exception handling
- More features

Some backgrounds

Today's computers

- are very powerful and clever
- e.g. can help doctors to diagnose patient's problems (IBM Watson)
- e.g. can win world top players in board game GO (AlphaGo)
- however, machines/mechanics
- always need instruction to do next
- the instructions must be accurate, detailed, and complete
- computer code is for that purpose.

Programming/coding principle

- Logical
- Then feasible, no conflicts, no ambiguity.
- E.g. not cooking and playing volleyball at the same time.
- E.g. in a many-road intersection, can not ask one to walk some steps without indicating direction.
- E.g. not try to present your "fourth" apple to your friend when you have only three.
- E.g. not try to divide 92 by "Please accept this cruise for our anniversary!"

How to learn coding

- Testing
- Testing
- And testing …

Computers of bits

- Almost everything in computers is bit or bits.
- Each bit can be imagined as a simple circuit with current running or not, two states only.
- Normally the two states are represented with 0 and 1.
- Then do you mean a computer can only describe two states?
- No. Each bit can do that. But we have many many bits.
- E.g. 0101110100010001 may mean "I love you!"
- Actually unlimited number of bits can express anything.
- All data and code instructions are bits.
- And anything computers do is on bits essentially.

Minimum working unit in computers

- Not simply one bit, but
- 8 bits
- Called one BYTE.
- This means whatever you do anything, one or more BYTEs will be used.

What we get when we buy a computer?

Laptops & MacBooks

HP 15.6" Touchscreen Laptop (AMD A9-9410 7th Generation/1 TB HDD/8 GB RAM/Windows 10 Home) - Black

$499.99

Save: $200

Sale Ends: June 29, 2017

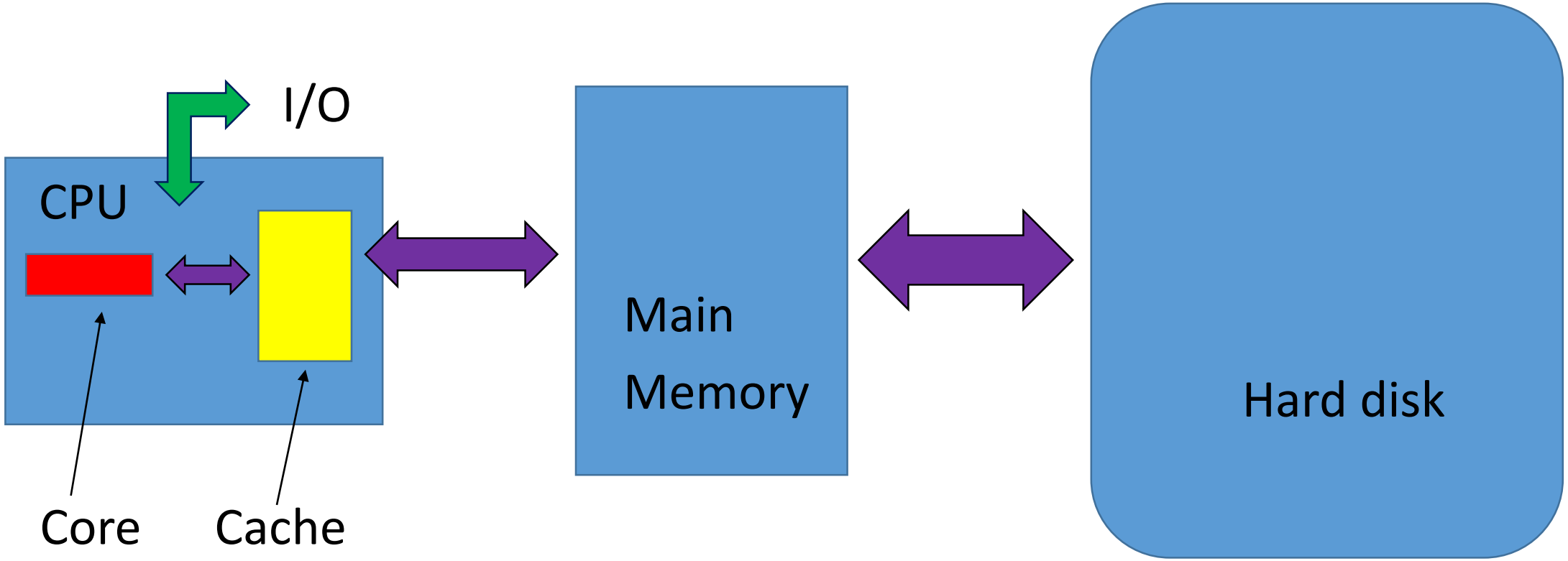
Shop Now ›

Computer key parts

- CPU. E.g. AMD A9-9410 2.9GHz
- Memory. 8GB is about 8,000,000,000 BYTES
- Hard disk: 1TB is about 1,000,000,000,000 BYTES

Accurately		
1KB	= 2^10 BYTES	= 1024 BYTES
1MB	= 2^10 KB	= 1024 KB
1GB	= 2^10 MB	= 1024 MB
1TB	= 2^10 GB	= 1024 GB
1PB	= 2^10 TB	= 1024 TB

A sketch of computer structure and data flow



CPU/Core can operate data only in cache at fixed frequency. Much additional time is spent in data transporting between the main memory and the cache. Fully making use of cache capacity reducing data movement between main memory and cache is critical for performance improvement.

Data and code

- are stored in hard disks as files in certain format.
- Files are placed in a hierarchy of directories.
- Although everything is bits/BYTEs, some files are stored in a way such that the BYTEs can be converted into characters, letter, numbers, and/or other symbols, then readable to people. Called text files.
- Other files are just bits, not indented to be converted. Called binary files. People can not read binary files. Meanwhile, computers can not run based on text files, but based on instructions in some binary files, called executable. Not all binary files are executable, e.g. movie data files.

Computer languages

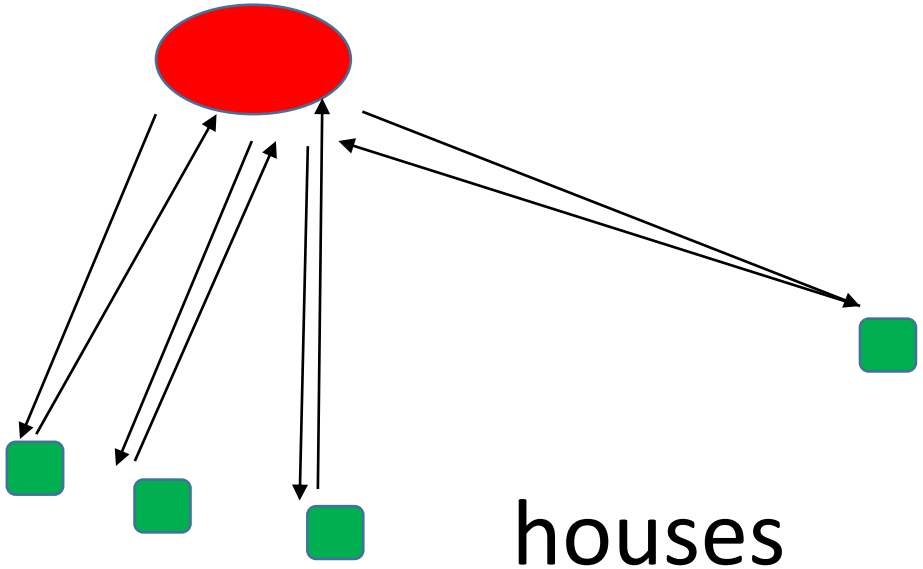
- Rules and facilities for writing (source) codes in text files, eventually converted into executable binaries to instruct computers what to do.
- Although they were created like natural languages as much as possible, their rules (syntax/grammar) are applied absolutely strictly. Any violation will be refused.
- There are a great number of computer languages.
- For application, especially computing/data science, high-level programming languages fall into two categories.
- Interpreted and compiled languages.

The difference

- In interpreted languages, like R, Python, and Matlab, one or more lines of source code is (are) converted into binary then executed. Then another section of source code. This procedure is repeated till end. Then computation is interrupted by conversions.
- In compiled languages, like C and FORTRAN, the whole source code is compiled into a big executable binary code. The compiled binary code can be run repeatedly later, forgetting the source code.

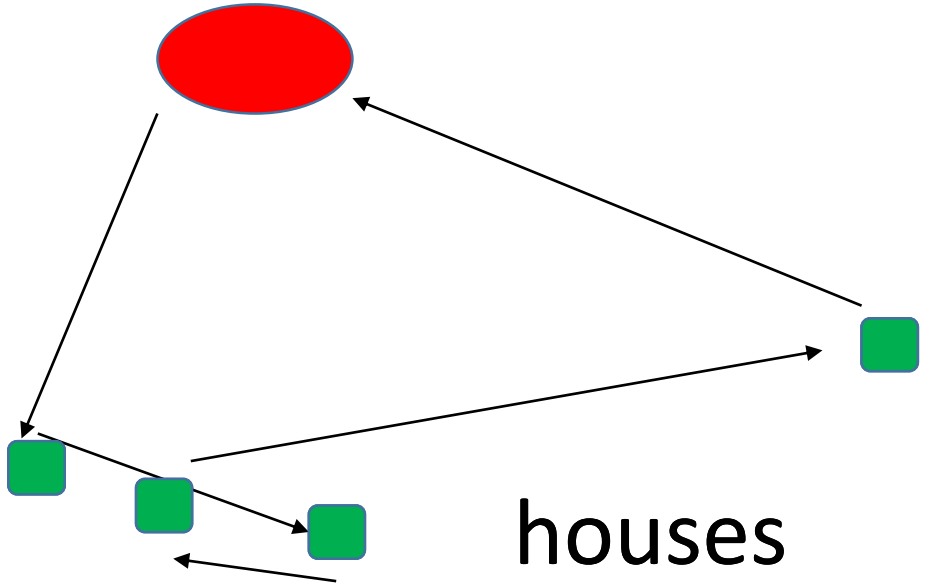
The difference

Post office



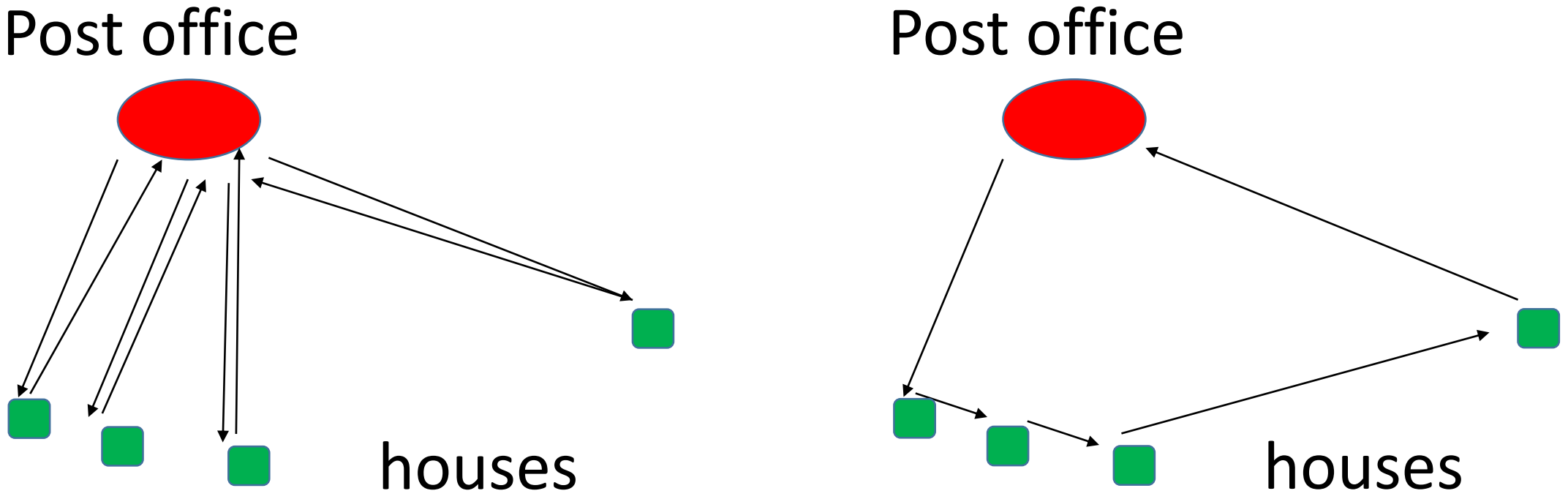
houses

Post office



houses

The difference



The interpreted, like a postman on the left, needs to go back and forth to deliver letters. The compiled (on the right) can deliver all together and further optimize since knows all tasks.

So

The compiled ones usually run much fast than the interpreted ones.

Advantage

The interpreted is more automatic, easier to use.

The compiled need more details to be coded with more care.


julia

IS

the only combination of the interpreted and compiled languages.

It can be coded as easy as interpreted ones and run as fact as compiled ones.

pretty young

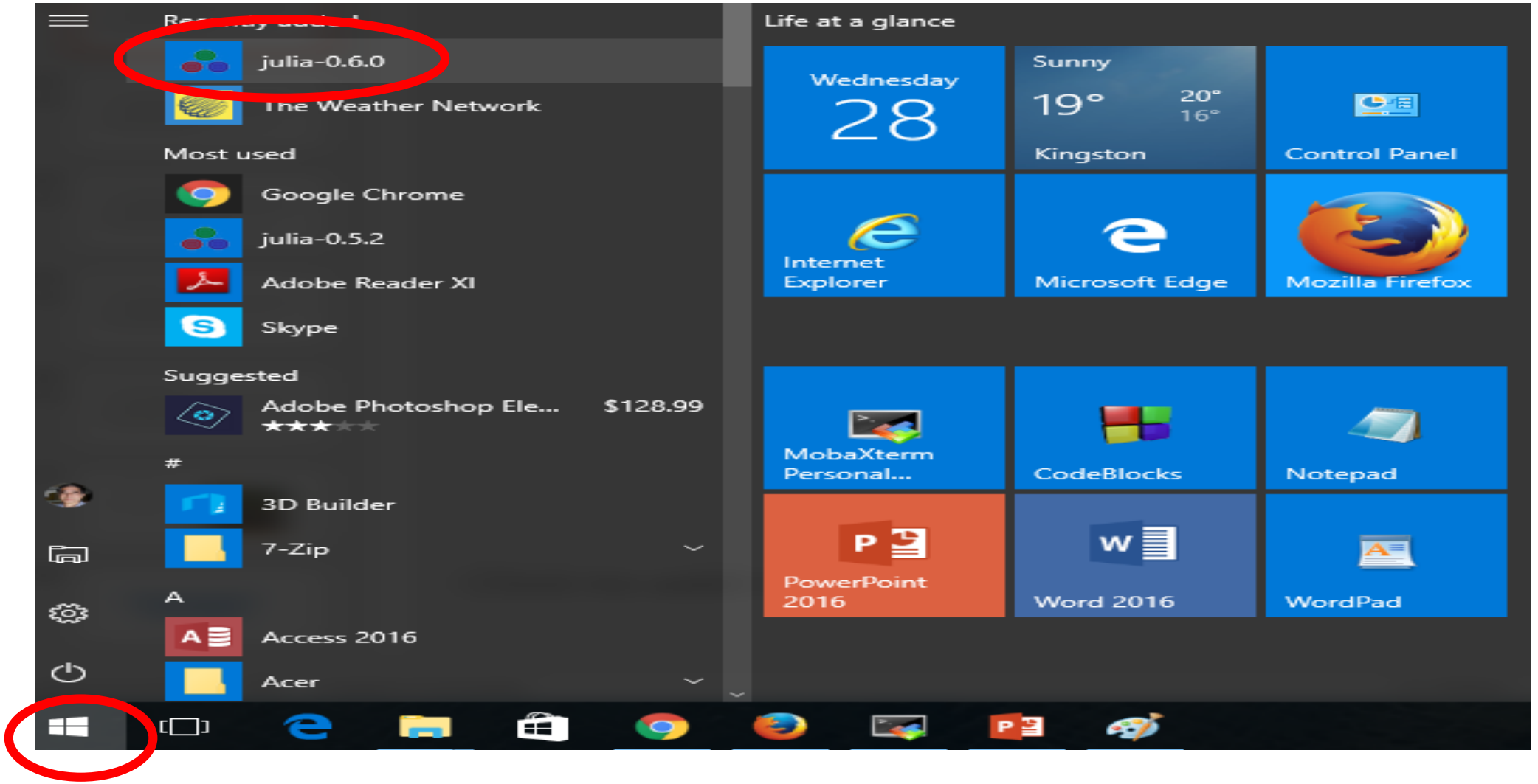
Developed by Jeff Bezanson, Stefan Karpinski, and Viral B. Shah, under supervision of Prof. Alan Edelman at MIT from 2009.

First presented publicly on Valentine's Day, 2012.

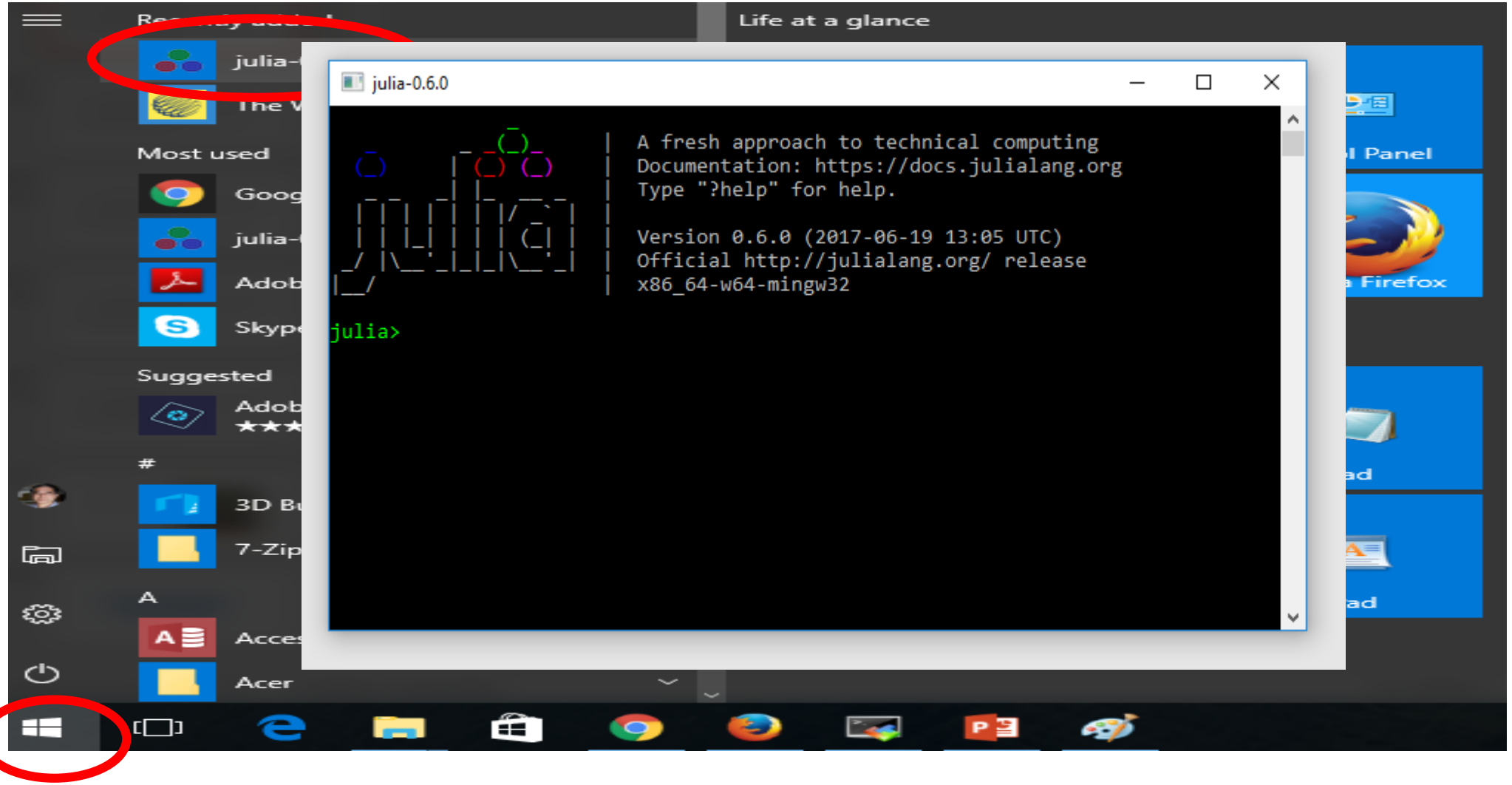
Free, open source with MIT license.

Starting with Julia's shell

Julia's shell



Julia's shell



Julia's shell

- The shell or REPL (Read-Evaluate-Print-Loop) or let us call it Command-line interface, is Julia's basic working environment, where you can interact with Julia's Just-in-time compiler to run your code piece by piece.
- You can type your code (expression) in "Julia>" prompt and press the ENTER key, then Julia should evaluate your expression and show.
- If you do not want your evaluated expression shown in the screen, type ";" , like 5+3;
- So many lines can be put into one line separated with ";", then the last result with be shown.
- You can exit it with (CTRL)+D keys or input quit().
- Test some: a=4 ans b=9 ans a+b ans c="No alcohol when driving" ans

Julia's shell

- Previous expressions can be retrieved with up and down array keys, then changed to get a new one.
- To clear or interrupt a current expression, press (CTRL)+C
- To clear the screen (but variables are kept in memory), press (CTRL)+L
- To reset the session so that variables are cleared, enter the command workspace()
- To search previous commands, press (CTRL)+R

Julia's shell

- Type ? to enter help mode help?> sort
- Julia> help(sort)
- propos("sort")
- Julia> so(TAB)(TAB)

Julia's shell

- ; to enter a system shell
- ;ls
- ;cd
- ;mkdir
- ;whoami
- ;pwd

Julia's shell

- To execute an existing Julia code
- Julie> include("d:/MyDirectory/myfolder/…/MyCode.jl")

Julia's comment lines

- To write something like notes, comments, for any purpose other than programming.
- After "#" in the line (anything before it is still code).
- # This is the best movie I have ever watched.
- price = 5000000 dollars # I am willing buy a ticket for my girl friend.

Basic Types, Variables, and Operations

Types

- 2
- 8.4,
- "Queen's University is in Kingston, a beautiful place. " # are some values.
- Each has a (data) type. "typeof()" function tells us the type
- typeof(1)
- typeof(1.0)
- typeof("1.0")
- typeof("Kingston is in Ontario. ")
- typeof(true)

Types

- Why are types?
- Each type was defined, so that a reasonable amount of bits or BYTEs allocated in memory to store such values with certain format.
- The most frequently used basic types in Julia are
- Int64
- Float64
- String
- Bool

Int64

- is for one integer.
- has 64 bits (8 BYTEs) in memory.
- one bit is used for sign, maybe 1 meaning +, 0 for -.
- the rest 63 bits are for the actual integer absolute value.
- the maximum positive value can be stored is 2^63 -1 = 9,223,3…,807
- can also be acquired by typemax(Int64) function call.
- If no negative allowed (the bit for sign also used for value), the corresponding type is called UInt64.

UInt64

- is for one non-negative integer.
- has 64 bits (8 BYTEs) in memory.
- the maximum value can be stored is 2^64 -1 = 18,446,…, 1,605
- can also be acquired by typemax(UInt64) function call.

Other built-in integer types

- Int8, Int16, Int32, Int128:
- Int128(3)
- typeof(ans)
- UInt8, UInt16, UInt32, UInt128:
- a=UInt128(3)
- typeof(ans)
- If all these types can not satisfy you, BigInt, which is supposed to hold unlimited big integer.
- BigInt(9)
- typeof(ans)

Float64

- is for one float number: 1.0, 2. 234323.9, 7.89234e72 .
- has 64 bits (8 BYTEs) in memory.
- one bit for the value sign.
- another bit for the sign of power.
- some bits for power.
- all the rest bits are for the absolute value.
- typemax(Float64)

Other built-in floating point types

- Float16, Float32
- No UFloat types
- BigFloat is defined, supposed for unlimited big floating point numbers, although actually maybe still limited.

Special values

- Inf (positive infinit)
- -Inf (negative infinit)
- NaN (not a number)
- a = 5.0^5
- a = a * a #repeat it

Variables

- Values of types are stored in memory when Julia is running.
- In order to retrieve them, variables are used.
- Variables can be regarded as a name or an ID for one or a group of values. Their types are defined as those of the values they represent.
- Variables are associated with values by assignment statement:
  a = Int32(4) b = Float32(2.0) a b a + b c = a + b
- Variables are case sensitive, typically lower case words separated by underscores.
- Variables must start with a letter and after that letters, digits, underscores, and exclamation points can follow in any combination:
  aaa, ddd, g, h, m8, spead_of_light, spead_of_light02, my_ spead_of_light
- Once defined, can be used as values.

Basic operations on numbers

- a = 3
- b = 4.8
- c = a + b
- c = a - b
- c = a * b
- c = a / b
- c = a ^ b
- d = sqrt(a)
- d = exp(b)
- d = sin(a)
- 2+3*5
- (2+3)*5

By default, operations on different types will automatically convert all values into a reasonable common type, then operate, since operations can only on the same types internally.

Reasonable precedence enabled (test to learn details). Always use parentheses ( ) to enforce precedence whenever not 100% sure.

Bool type

- true or false
- a = 3
- b = 4
- a < b
- a == b
- c = a == b
- c = (a == b)
- typeof(c)

Bool type

- Data comparison:
- 2==3
- 2 !=3
- 2 < 3
- 2 > 3
- 2 >= 3
- 2 <= 3
- Bool operations: &&, ||, !
- (5<3) && (4.0<8)
- (5<3) || (4.0<8)
- !true
- !false

String type

- str0 = "Volleyball is a sport and an entrainment."
- typeof(ans)
- Each letter can be retrieved with an index from 1 to end:
- str0[1]
- str0[2]
- str0[10]
- str0[end]
- str0[10]='g'
- Strings are immutable: any element can not be changed once created, although the variable can be assigned with anything else (then it has nothing to do with the old whole string).

String operations

- str0 = "Volleyball is a sport and an entrainment."
- str1 = search(str0, "sport")
- str1 = replace(str0, "Volleyball", "Hocky")
- str0
- str0 = replace(str0, "Volleyball", "Hocky")
- str0
- str1 = split(str0, ' ')
- str1 = str0* " Sorry, I missed the game yesterday, due to a flat tire."

Char type

- 'A' (only one letter with a single quote). Typeof(ans)
- If double quote like "A", it is a String type.
- Char can be converted into an integer (position in ASCII table) back and forth. Int('g') , Char(103)
- Any single letter can be defined and used as a Char or a single element String, which are different types.
- Char arrays can be converted into Strings with join() function call.

Complex type and Rational type

- are also defined as built-ins.

- (1+2im)*(3-4im)
- a = 8//3
- a * 5
- a/3
- numerator(a)
- denominator(a)

Since memory of computers

- is a great amount of bits/BYTEs but bits/BYTEs only,
- programming in any language must deal with types to save data there.
- In Julia, as in other modern interpreted languages, we can
  a = 0.5 , later a = "Newton likes apples, so do I."
- This means Julia manages types for us automatically, and even allows types change automatically.
- In C/FORTRAN, any given variable must be declared as a certain type before being used, and the type can no longer be changed.
- Later we will show that Julia further allows us to use types as something like assignable variables.

Julia

- is such a high level computer language, that programmers can completely forget types, letting Julia does everything on this respect;

- is such a high level computer language, that programmers can completely operate on types as variables.

Conditional branches

Conditional branches

- It allows us to code sections of code to be executed under conditions.
- Most cases, different conditions execute different part/path of code.
- The general form
- if condition1
- (do some things accordingly)
- elseif condition2
- (do some other things accordingly)
- elseif condition3
- (do some other things accordingly)
- elseif …
- (do some other things accordingly)
- else
- (do other things accordingly)
- end
- All elseif and else constructs are optional and no limit on number of elseif constructs.

Conditional branches

- An example
- var = 5
- if var > 10
- println("var has value $var and bigger than 10. ")
- elseif var < 10
- println("var has value $var and smaller than 10. ")
- else
- println("var has value $var and is 10. ")
- end

Conditional branches

- Another example
- a = 5; b = 9
- z = if a > b a
- else b
- end
- This short if structure can also be written as
- z = a > b ? a : b

Conditional branches

- if (condition) (statement) end
- can be written
- (condition) && (statement)
- For example if (a<0) println("Sorry, a is negative: $a") end
- (a<0) && println("Sorry, a is negative: $a")

- Similarly, if !(condition) (statement) end
- can be written
- (condition) || (statement)

Conditional branches

- can be unlimited nested.

- Example
- if my_dad_is_at_home
- (my dad will cook)
- else
- if my_mom_is_at_home
- (my mom will cook)
- else
- (I will cook)
- end
- end

Repeating constructs

If you have

- log(2.0) + log(3.0) + log(4.0) + log(5.0) + … + log(10000000000.0) to calculate,
- you would absolutely not like to write all such operations one by one.
- Loops are for such repeated works and can make life much easier.

The for loop

- takes a form of
- for i = beginning : ending
- (do something)
- end

- Example
- for a = 2001 : 2017
- println(a, " is a year of this centrury. ")
- end

The for loop

- takes a form of
- for i = beginning : ending
- (do something)
- end

- Example
- s = 0.0
- for i = 2 : 10000
- lll = log(i)
- s = s + lll
- end
- s

The for loop

- can take another form of
- for i = 1 : length(a_collection)
- (do something)
- end

- Example
- c = "What a beautiful day!"
- for ii = 1 : length(c)
- println("The $ii-th element of the collection is ", c[ii], " . ")
- end

The for loop

- can take another form of
- for e in (a_collection)
- (do something)
- end

- Example
- c = "What a beautiful day! "
- for ee2 in c
- println(ee2)
- end

The for loop

- can take another form of
- for (i,e) in enumerate(a_collection)
- (do something)
- end

- Example
- c = "What a beautiful day! "
- for (ii, ee2) in enumerate(c)
- println("The $ii-th element of the collection is $ee2 . ")
- end

The for loop

```
str0 = "Volleyball is a sport and also an entrainment. "
for elmt in split(str0, ' ')
    println(elmt)
end
```

Loops can always be unlimited nested

```
for  n1  =  1:5
    for n2 = 1:5
        println("The multiplication of $n1 and $n2 is $(n1*n2) . ")
    end
end
```

- Or (I do not like it)

```
for  n1  =  1:5,  n2 = 1:5
        println("The multiplication of $n1 and $n2 is $(n1*n2) . ")
end
```

The while loops

- while condition_is_true
- (do something)
- (usually the condition is updated, so that the condition false sometime later)
- end

- Example
- a = 10; b = 15
- while a < b
- println("$a, $b . ")
- a +=1 # a = a +1
- end

The while loops

- Please also try
- a = 15; b = 15
- while a < b
- println("$a, $b .")
- a +=1 # a = a +1
- end

A typical usage of the while loops

- (initialize some data including setting ccc = true)
- while ccc
- (do some things)
- (check if everything is good enough setting ccc = false)
- end

break statement

- A loop will be terminated when a break statement is met.

```
a = 15;  b = 1500
while a < b
    println("$a, $b .")
    a +=1
    if a > 20  break end
end
```

break statement

- A loop will be terminated when a break statement is met.

```
a = 15;  b = 1500
while true
    println("$a, $b .")
    a +=1
    if a > 20  break end
end
```

continue statement

- will skip all the rest statements in the current iteration.

```
for n in 1 : 20
    if n <= 15 && n >= 5
        continue
    end
    println("I like $n .")
end
```

continue statement

- Another example for even numbers
- for n = 1 : 20
- if isodd(n)
- continue
- end
- println("$n is even.")
- end

- The same effect
- for n = 2:2 : 20
- println("$n is even.")
- end

Range

- for n = 1 : 3 : 40 # start : step : end
- println(n)
- end

Functions

Nowadays, people do everything step by step

Programming is not an exception

- A big computational task is usually cut into many smaller ones.
- With input and output assumed to some degrees, what and how to complete the inside of each smaller task is usually independent on any other smaller tasks. In other words, the inside is encapsulated.
- Functions are used for such smaller tasks or steps.
- Normally when we code functions, we can focus on the specific small task, forgetting the whole big complicated task.
- Functions can be unlimitedly re-used and make code well structured.
- We already saw many built-in functions:
  typeof(a), Int64(2), sin(x), replace(aa, "…", "…"), println("…")
- Let us try to write our own functions.

The general form of functions

function function_name(argument_list)

- # function body (statements)
- return (values_or_variables)
- end

Function example

```
function myfunc_01(x, y)
    println("For $x and $y, the result is: ")
    return x + y^3
end

myfunc_01(3,2)
```

Function example

```
function myfunc_01(x, y)
    println("For $x and $y, the result is: ")
    return x + y^3
end

myfunc_01(3,2)
x
y  #arguments are “temporary variables”, disappeared when exited.
```

Function names can be assigned to others

```
function myfunc_01(x, y)
    println("For $x and $y, the result is: ")
    return x + y^3
end
```

- aaa = myfunc_01
- aaa(3,2)

Function example

```
function myfunc_02(x, y)
    println("For $x and $y, the result is: $(x+y^3). ")
    x,   y,   x + y^3
end

a,b,c = myfunc_02(3,2)
a
b
c
```

Function example

```
function myfunc_03(x, y)
    println("For $x and $y, the result is: $(x+y^3). ")
    if y == 0
        return x
    end
    x + y^3
end

myfunc_03(3,0)
```

Function with arbitrary number of arguments

```
function myfunc_04(x, y, argus ...)
        println("x: $x, y: $y, argus: $argus")
        for  a  in argus
                println(a)
        end
end
```

- a = 4; b = 7; c =12 ; myfunc_04(3, 0, a, b, c)

Function with arbitrary number of arguments

```
function myfunc_05(x, y, argus ...)
        println("x: $x, y: $y, argus: $argus")
        if length(argus) >0
                println(argus[1], " first and last ", argus[end])
        end
end
```

- a = 4; b = 7; c =12 ; myfunc_05(3, 0, a, b, c)

Compact form of function with one line in the body

```
function myfunc_06(x, y, z)
    x + 3y^3 + 6z^4
end
```

- can be written as

```
myfunc_06(x, y, z) =   x + 3y^3 + 6z^4
```

Anonymous function

```
function (x, y, z)
        x + 3y^3 + 6z^4
end
```

can be written as

```
(x, y, z)  ->   x + 3y^3 + 6z^4
ans(3, 4, 5)
```

If only one argument

```
x -> 3 + 4x + 82x^2 + 92x^3
ans(3)
```

If no argument

```
() -> println(5678)
ans()
```

Anonymous function can be named

```
(x, y, z)  ->   x + 3y^3 + 6z^4
ans(3, 4, 5)

myfunc_07 = (x, y, z)  ->   x + 3y^3 + 6z^4
```

Or

```
myfunc_07 = function (x, y, z)
                        x + 3y^3 + 6z^4
                    end
myfunc_07(3, 4, 5)
```

Optional arguments with default values

```
myfunc_08 = (x, y=2) -> x + y^3
```

- myfunc_08(3,0)
- myfunc_08(3)
- myfunc_08(3,2)
- myfunc_08(3,3)

Keyword arguments

Up to now, all arguments are defined by sequential positions. After them, a semicolon (;) can be placed, then optional keyword arguments can be defined with default values. Keyword arguments can be in any order when calling such a function.

```
myfunc_09 = (x; y=4 , z=2)  ->   x + 3y^3 + 6z^4
myfunc_09(1)
myfunc_09(1; y=4, z=2)
myfunc_09(1; z=2, y=4)
myfunc_09(1, z=2, y=4)
myfunc_09(1, z=2)
myfunc_09(z=2, 1)
myfunc_09(y=4, 1, z=2)
```

Functions can accept functions as arguments

```
function numerical_derivative(f, x, dx=0.01)
        derivative = (f(x+dx) - f(x-dx))/(2.0*dx)
        return derivative
end
```

- `h = x -> 2x^2 + 30x + 9`
- `numerical_derivative(h, 1,  0.001)`
- Or
- `numerical_derivative(x -> 2x^2 + 30x + 9, 1,  0.001)`

Functions can also contain and call function(s)

function layered1(x)

- h=x->5x
- return 2x+3+h(x)
- end

- layered1(2)

Functions can also contain and return function(s)

```
function  layered2(f)
    return    function(x)
                  return 2x+3+f(x)
    end
end
```

- h=x->5x
- layered2(h)
- layered2(h)(2)

Functions can also contain and return function(s)

```
function  layered3(f)
    return    function(x)
        return     function(y)
            return 2x+y+f(x)
        end
    end
end
```

- h=x->5x
- layered3(h)
- layered3(h)(2)
- layered3(h)(2)(3)

Interesting anonymous functions returned

```
function counter()
        n = 0
        () -> n += 1, () -> n =0, () -> n
end
(addOne, reset, theValue) = counter()
n
reset()
addOne()
addOne()
theValue()
addOne()
reset()
addOne()
```

Please note that n is only accessible to reset(), addOne(), and theValue(). Called closures.

More interesting:

```
(addOne2, reset2, theValue2) = counter()
addOne()
addOne2()
addOne()
addOne2()
```

Recursive functions

```
function factorial(n)
        if   n == 1
                return 1
        else
                n * factorial(n-1)
        end
end

factorial(5)
```

As you see

```
f3 = (x, y, z)  ->   x + 3y^3 + 6z^4
f3(3, 4, 5)
```

- the types of most variables in functions are not defined.
- When the function is called, the types of all argument are given, then types of all variables in the function are derived.
- In this procedure, Julia compiler will write a specific function code for the concrete types of the actual arguments passed in, based on the "general function" you wrote, then run it. Your function is actually a framework.
- This a great advantage of Julia over almost all other languages.
- However, this is slow. Better to specify types in your functions.

Type specifications in functions

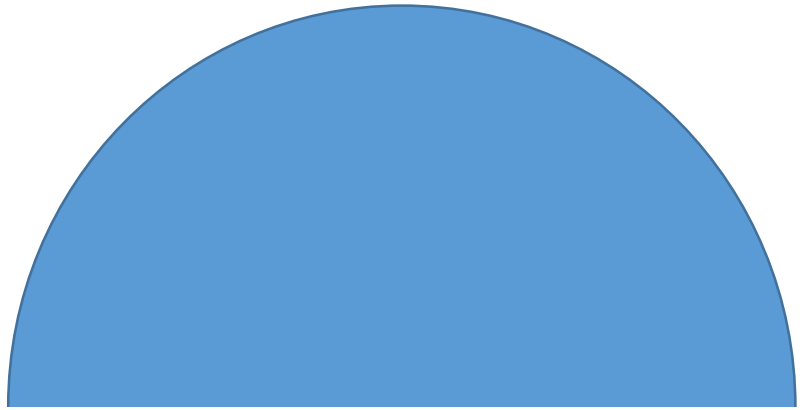
```
function factorial(n::Int64)  ::  Int64
        if   n == 1
              return 1
        else
              n * factorial(n-1)
        end
end
factorial(3)

f3 = (x::Int64,  y::Int64,  z::Int64)  ->   (x + 3y^3 + 6z^4) :: Int64
f3(3, 4, 5)
```

Application example:

π calculation of many digits

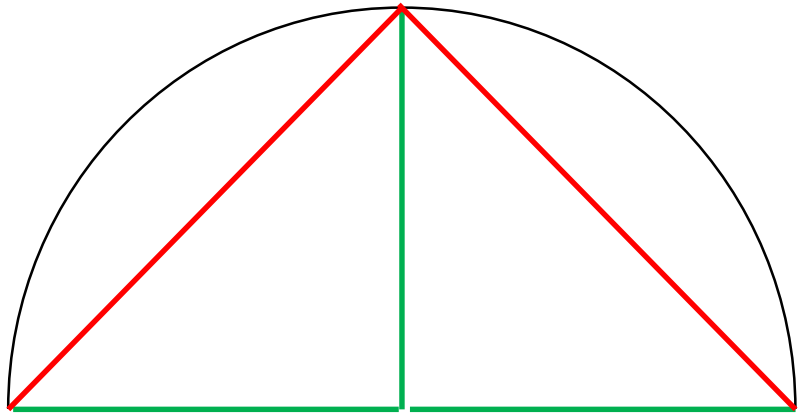
π calculation based on half circle

$$\pi = \frac{half\_circle}{r}$$



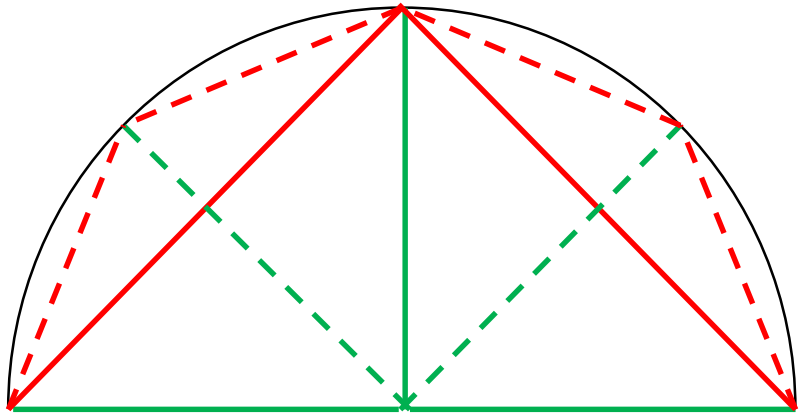
π calculation based on half circle

$$\pi \approx \frac{\sum chords}{r}$$



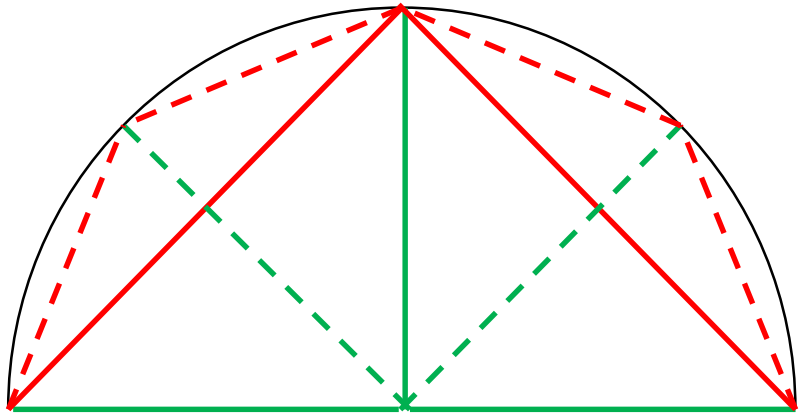
π calculation based on half circle

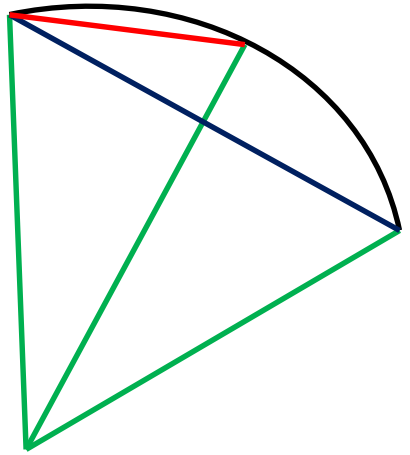
$$\pi \approx \frac{\sum chords}{r}$$



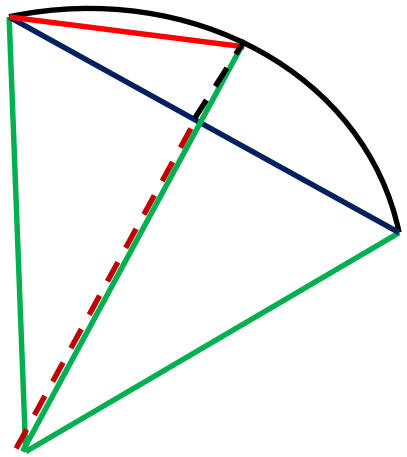
π calculation based on half circle

$$\pi \approx \frac{\sum chords}{r}$$





π calculation based on half circle



$$\pi \approx \frac{\sum chords}{r}$$

h: old chord in blue
$v1$: dark red dashed part
$v1 = \sqrt{r^2 - (h/2)^2}$
$v2$: black dashed part
$$v2 = r - v1$$

new chord in red

$$\sqrt{(h/2)^2 + (v2)^2}$$

π calculation based on half circle

```
function new_chord(old_chord)
        h = old_chord / 2
        hsq = h * h
        v1 = sqrt(1.0 - hsq)
        v2 = 1.0 - v1
        return(sqrt(hsq + v2*v2))
end
```

$$\pi = \frac{\sum chords}{r}$$

h: old chord in blue
$v1$: dark red dashed part
$v1 = \sqrt{r^2 - (h/2)^2}$
$v2$: black dashed part
$v2 = r - v1$

new chord in red

$$\sqrt{(h/2)^2 + (v2)^2}$$

π calculation based on half circle

accuracy = 1.0e-12

- number_of_chords = 2.0
- old_chord = sqrt(2.0)
- pi = old_chord * number_of_chords
- old_pi = pi
- while true
- number_of_chords = number_of_chords * 2.0
- old_chord = new_chord(old_chord)
- old_pi = pi
- pi = number_of_chords * old_chord
- relative_error = abs(pi-old_pi)/pi
- if relative_error < accuracy
- println("My PI: $pi to the accuracy: $accuracy.")
- break
- end
- end

$$\pi = \frac{\sum chords}{r} \qquad r=1.0$$

```
function new_chord(old_chord)
        h = old_chord / 2
        hsq = h * h
        v1 = sqrt(1.0 - hsq)
        v2 = 1.0 - v1
        return(sqrt(hsq + v2*v2))
end
```

π calculation for higher accuracy

accuracy = 1.0e-70

```
number_of_chords = 2.0
old_chord = sqrt(2.0)
pi = old_chord * number_of_chords
old_pi = pi
while true
        number_of_chords = number_of_chords * 2.0
        old_chord = new_chord(old_chord)
        old_pi = pi
        pi = number_of_chords * old_chord
        relative_error = abs(pi-old_pi)/pi
        if relative_error < accuracy
            println("My PI: $pi to the accuracy: $accuracy.")
            break
        end
    end
```

$$\pi = \frac{\sum chords}{r} \qquad r=1.0$$

```
function new_chord(old_chord)
        h = old_chord / 2
        hsq = h * h
        v1 = sqrt(1.0 - hsq)
        v2 = 1.0 - v1
        return(sqrt(hsq + v2*v2))
end
```

π calculation for higher accuracy

```
function new_chord(old_chord::BigFloat)
        h = old_chord / BigFloat(2.0)
        hsq = h * h
        v1 = sqrt( BigFloat(1.0) - hsq)
        v2 = BigFloat(1.0) - v1
        return(sqrt(hsq + v2*v2))
end
```

π calculation for higher accuracy

```
accuracy  =  1.0e-70
number_of_chords =  2.0
old_chord  =  sqrt(2.0)
pi = old_chord  * number_of_chords
old_pi =  pi
while true
        number_of_chords  = number_of_chords  * 2.0
        old_chord  = new_chord(old_chord)
        old_pi = pi
        pi = number_of_chords  * old_chord
        relative_error = abs(pi-old_pi)/pi
        if  relative_error < accuracy
            println("My PI: $pi to the accuracy: $accuracy.")
            break
        end
    end
```

π calculation for higher accuracy

accuracy = BigFloat(1.0e-70)

- number_of_chords = BigFloat(2.0)
- old_chord = sqrt(BigFloat(2.0))
- pi = old_chord * number_of_chords
- old_pi = pi

π calculation for higher accuracy

```
i = 1
while true
        number_of_chords  = number_of_chords  * BigFloat(2.0)
        old_chord  = new_chord(old_chord)
        old_pi = pi
        pi = number_of_chords  * old_chord
        println("$i :  $pi ")
        relative_error = abs(pi-old_pi)/pi
        if  relative_error < accuracy
            println("My PI: $pi to the accuracy: $accuracy.")
            break
        end
        i = i + 1
    end
```

π in wiki

- https://en.wikipedia.org/wiki/Pi

- The first 50 decimal digits are 3.14159265358979323846264338327950288419716939937510

Collection types and user defined types

Collection types and user defined types

- In real calculations, we will meet a great amount of data.
- To manage them reasonably is necessary for a successful work.
- Collection types and user defined types are for that purpose.

Vectors and matrixes

store a sequence of values of the same type (called elements), indexed by the sequential integer number starting from 1.

Vectors and matrixes

- a = [1, 2, 3]
- a[2]
- a2 = [1; 2; 3]
- a == a2
- b = [1 2 3]
- b[2]
- b == a2
- c = a * b
- c[2,3]
- d = b * a
- h = [1 2; 3 4]
- h[2, 3]
- g = h * h
- g[2, 3]

Vectors and matrixes

- To create a 3 by 5 matrix of random floating numbers between 0 and 1:
- a = rand(3, 5)
- Or
- b = rand(Float64, 3, 5)

- c = rand(Int64, 20)
- d = rand(Int64, 7, 10)
- d[1, 1]
- d[1, 2]
- d[1, 3]
- d[1, 10]

Vectors and matrixes

- a = Float64[]
- b = Array{Int64}(8)
- c = linspace(0, 10, 11)
- c[3]
- d = Float64[x^2 for x in 1:4]
- g = [x for x in 1:8]
- h = [x^3 for x in 1:5]
- table = [x*y for x in 1:10, y=1:10]
- stable = [sin(Float64(x+3y)) for x in 1:4, y=1:3]

Vectors and matrixes

- b = Array{Int64}(8)
- for j = 1: length(b)
- b[j] = 10+j
- end
- b
- b[3:7]
- b[3:7]=[45,46,47,48,49]
- b

  b[2:end]

Vectors and matrixes

- a = [x for x in 1:6]; a
- b = [x+12 for x in 1:3]; b
- pop!(a); a
- push!(a, 1080) ; a
- c = append!(a,b)
- sort(a)
- a
- sort!(a)
- a
- join(a, '-')

- shift!(a)
- unshift!(a, 36)
- splice!(a,3); a
- in(36, a)
- in(78, a)
- for elm in a
- println(" $elm ") end

Vectors and matrixes

- c = [1, 2, 4, 8, 16]
- d = c
- d[3] = -9
- d
- c

- c = [1, 2, 4, 8, 16]
- d = deepcopy(c)
- d[3] = -9
- d
- c

Vectors and matrixes

- a = rand(3, 5)
- ndims(a)
- size(a, 1)
- size(a, 2)
- a[:, 2]
- a[2, :]
- a[2:end, 2:end]

- b = eye(4)

Vectors and matrixes

- To make an array of arrays
- a = fill(Array(Int64, 1), 3)
- a[1] = [1,2]
- a[2] = [2,3,4]
- a[3] = [8,9,10,70]
- a

Vectors and matrixes

- To transpose an array
- a = [1 2; 3 4]
- b = a'
- Or
- c = transpose(a)
- c == b

Vectors and matrixes

- The inverse
- a = [1 2; 3 4]
- b = inv(a)
- b * a
- a * b

Vectors and matrixes

- Concatenations
- a = [1 2; 3 4]
- b = [5 6; 7 8]
- c = [a b]
- c = [a ; b]
- c = [a , b]
- hcat(a, b)
- vcat(a, b)

Vectors and matrixes

- Element-wise operations
- a = [1 2; 3 4]
- b = [2 2; 2 2]
- a * b
- a .* b
- b .* a
- a .+ b
- a .- b

Vectors and matrixes

- map(function, array) applies the function on elements
- a = [1 2 3; 4 5 6]
- map(x-> x^3, a)
- Other forms
- map(x-> begin
- if iseven(x) return 2
- else return 1
- end
- end , a)

- map(a) do x
- if iseven(x) return 2
- else return 1
- end
- end

Vectors and matrixes

- filter(function, array) only returns elements if the function evaluated true.
- a = [1 2 3; 4 5 6]
- filter(x-> x>4, a)

Tuples

- A group of values separated by commas, surrounded by parentheses ().
- The types can be the same or different.
- Any value of it can not be changed, as tuples are immutable.

- a, b, c, d = 2, 3.5, "good news", 'y'
- a
- b
- c
- d

Tuples

```
a = 2, 3.5, "good news", 'y'
for i = 1: length(a)
    println(a[i])
end
a[end]
```

```
a = (2, 3.5, "good news", 'y')
for i in a
    println(i)
end
a[end]
```

Dictionaries

For values indexed with unique keys.

- Collection of (key => value) pairs separated by commas.
- a = Dict([:A => 200, :B => 40])
- a[1]
- a[:A]
- a[:B]
- a[:B] = 80
- a[:C] = 90
- a # values can be changed.
- keys(a)
- values(a)

Dictionaries

- a = Dict(:A => 200, :B => 40, :C => 95)
- for (k,v) in a
- 	println(k, ' ', v)
- end

- for k in a
- 	println(k[1], ' ', k[2])
- end

Sets

- s1 = Set([2, 4, 8, 4, 5, 8, 8])
- s2 = Set([2, 14, 15, 4])
- union(s1, s2)
- intersect(s1, s2)
- setdiff(s1, s2)
- setdiff(s2, s1)
- issubset(s1, s2)
- issubset(intersect(s1,s2), s2)

User defined types

- type Point
- x::Float64
- y::Float64
- z::Float64
- end

- Point(2.0, 4.0, 5.0)
- a = Point(2.0, 4.0, 5.0)
- a.x
- a.y
- a.z
- a.y = 0.8
- a

User defined types

- immutable Point2
- x::Float64
- y::Float64
- z::Float64
- end

- Point2(2.0, 4.0, 5.0)
- a = Point2(2.0, 4.0, 5.0)
- a.x
- a.y
- a.z
- a.y = 0.8
- a

Input, output, and external files

Input, output, and external files

- a = readline()
- Hello!
- a
- a = readline(STDIN)
- Hello again!
- a

- println("A nice Summer! ")

Input, output, and external files

- Student , gender , age , weight , hight , grade
- First , boy , 17 , 63.4 , 178.4 , 11
- Second , girl , 16 , 50.3 , 165.2 , 10
- Third , girl , 15 , 47.6 , 155.3 , 8

```
myfile = open("c:/tmp/data.csv")
i = 0
while !eof(myfile)
        a = readline(myfile)
        i += 1
        println(" Line # $i is: $a ") end
close(myfile)
```

Input, output, and external files

- Student , gender , age , weight , hight , grade
- First , boy , 17 , 63.4 , 178.4 , 11
- Second , girl , 16 , 50.3 , 165.2 , 10
- Third , girl , 15 , 47.6 , 155.3 , 8

- myfile = open("c:/tmp/data.csv")
- myoutputfile = open("c:/tmp/out.csv", "w")
- while !eof(myfile)
- a = readline(myfile)
- println(myoutputfile, a)
- end
- close(myfile)
- close(myoutputfile)

Input, output, and external files

- Student , gender , age , weight , hight , grade
- First , boy , 17 , 63.4 , 178.4 , 11
- Second , girl , 16 , 50.3 , 165.2 , 10
- Third , girl , 15 , 47.6 , 155.3 , 8

- myfile = open("c:/tmp/data.csv")
- myoutputfile = open("c:/tmp/out.csv", "a")
- while !eof(myfile)
- a = readline(myfile)
- println(myoutputfile, a)
- end
- close(myfile)
- close(myoutputfile)

Reading CVS files

- CVS files are of lines with data separated by commas (or other fixed and unique delimiters like ";").

- Student , gender , age , weight , hight , grade
- First , boy , 17 , 63.4 , 178.4 , 11
- Second , girl , 16 , 50.3 , 165.2 , 10
- Third , girl , 15 , 47.6 , 155.3 , 8

- myfile = open("c:/tmp/data.csv")
- data = readdlm(myfile)
- close(myfile)

Reading CVS files

- myfile = open("c:/tmp/data.csv")
- # data = readdlm(myfile, ';' , Float64, '\n', header=true)
- data = readdlm(myfile, ',' , Any, '\n', header=true)
- close(myfile)

- data
- data[1]
- data[2]
- data[1][2,3]
- data[1][2,3] = 12.9
- data[1][2,3]
- data[2][3]
- dd = map(Float64, data[1][:, 3:end])
- dd

Reading and writing CVS files

- myfile = open("c:/tmp/data.csv")
- # data = readdlm(myfile, ';' , Float64, '\n', header=true)
- data = readdlm(myfile, ',' , Any, '\n', header=true)
- close(myfile)

- myoutfile = open("c:/tmp/out2.csv", "w")
- write(myoutfile, join(data[2], ';'), "\n")
- for i = 1: size(data[1],1)
- write(myoutfile, join(data[1][i,:], ';'), "\n")
- end
- close(myoutfile)

Reading and writing CVS files

- myfile = open("c:/tmp/out2.csv")
- data2 = readdlm(myfile, ';' , Any, '\n', header=true)
- close(myfile)

- data == data2

Reading and writing DataFrames

- The most natural representation of data. Here is a good example:
- Student , gender , age , weight , hight , grade
- First , boy , 17 , 63.4 , 178.4 , 11
- Second , girl , 16 , 50.3 , 165.2 , 10
- Third , girl , 15 , 47.6 , 155.3 , 8

- Pkg.add("DataFrames")
- using DataFrames
- myfile ="c:/tmp/data.csv"
- data = readtable(myfile, separator=',')
- writetable("c:/tmp/out3.csv", data)
- data3 = readtable("c:/tmp/out3.csv", separator=',')
- data == data3

Reading and writing DataFrames

- Student , gender , age , weight , hight , grade
- First , boy , 17 , 63.4 , 178.4 , 11
- Second , girl , 16 , 50.3 , 165.2 , 10
- Third , girl , 15 , 47.6 , 155.3 , 8

- using DataFrames
- data = readtable("c:/tmp/data.csv", separator=',')
- data[:hight]
- data[:hight][2]
- data[5][2]
- data[2, 5]
- data[:hight][2] = 2588.64
- data

Reading and writing DataFrames

- Student , gender , age , weight , hight , grade
- First , boy , 17 , 63.4 , 178.4 , 11
- Second , girl , 16 , 50.3 , 165.2 , 10
- Third , girl , 15 , 47.6 , 155.3 , 8

- using DataFrames
- data = readtable("c:/tmp/data.csv", separator=',')
- typeof(data)
- size(data)
- head(data)
- tail(data)
- names(data)
- eltypes(data)
  describe(data)

DataFrames constructor

- Student , gender , age , weight , hight , grade
- First , boy , 17 , 63.4 , 178.4 , 11
- Second , girl , 16 , 50.3 , 165.2 , 10
- Third , girl , 15 , 47.6 , 155.3 , 8

- using DataFrames
- dataf = DataFrame()
- dataf[:Student] = ["First", "Second", "Third"]
- dataf[:gender] = ["boy", "girl", "girl"]
- dataf[:age] = [17,16,15]
- dataf[:weight] = [63.4, 50.3, 47.6]
- dataf[:hight] = [178.4, 165.2, 155.3]
- dataf[:grade] = [11, 10, 8]
- dataf
- data == dataf

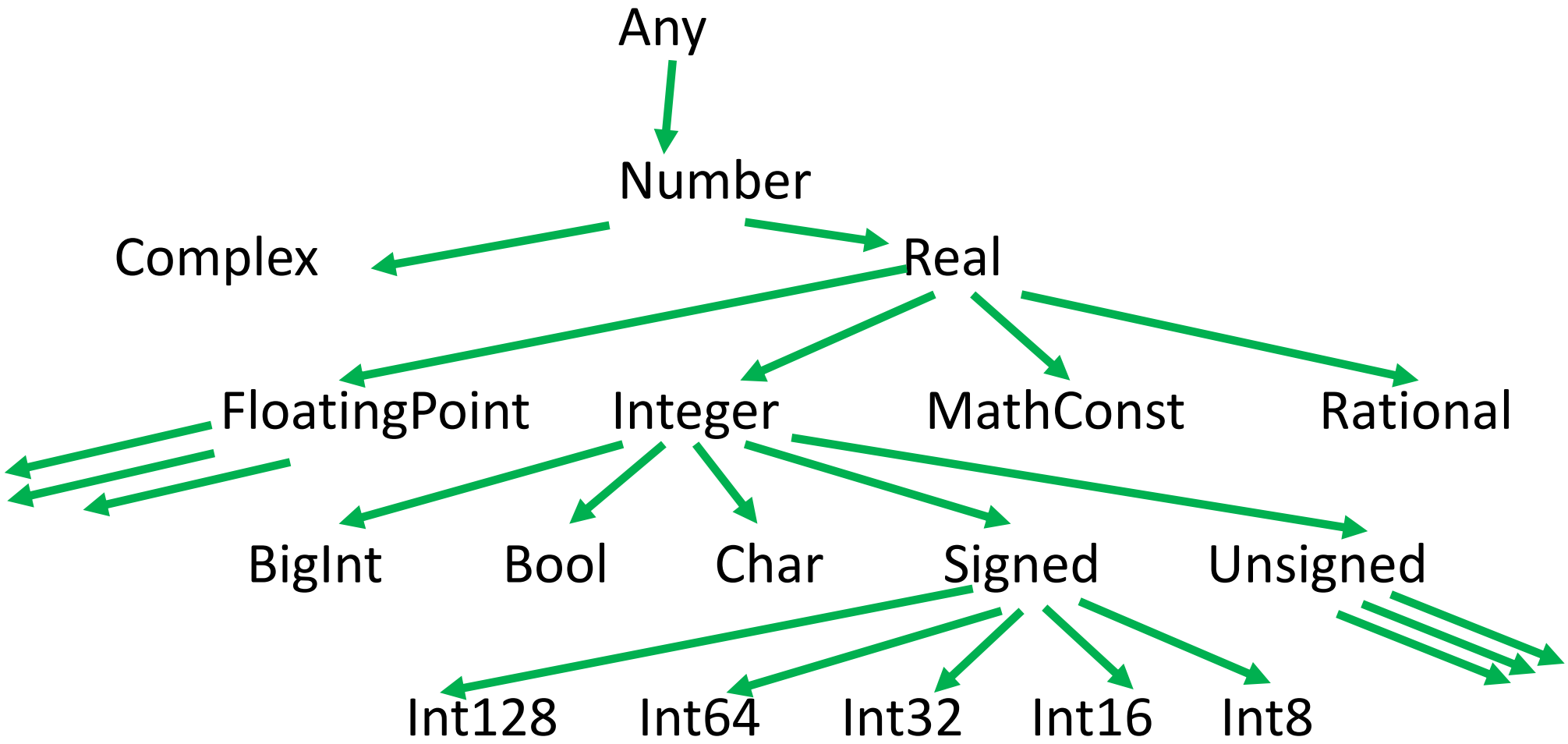
Other file formats

- JSON, by using the JSON package
- XML, the LightXML package
- YAML, the YAML package

Type hierarchy, immutability, and typed functions

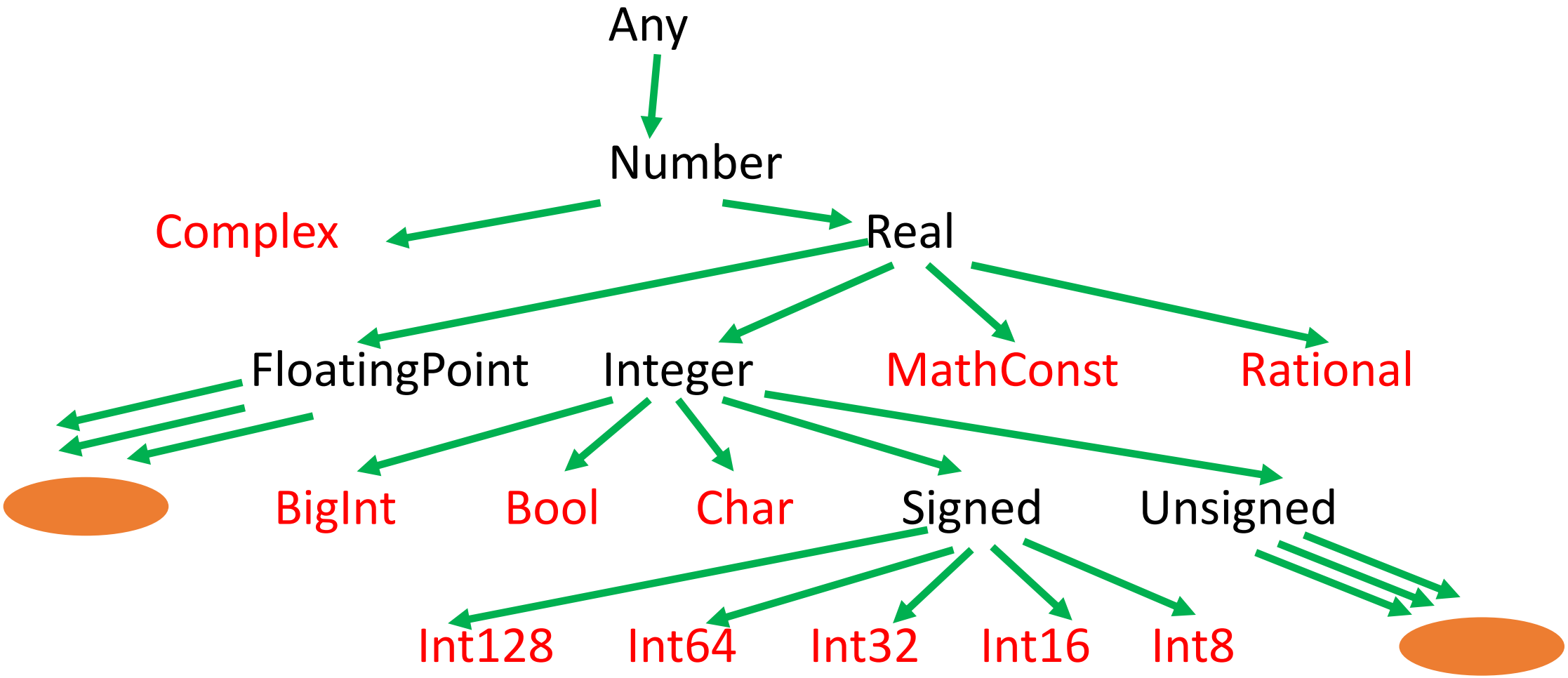
Type hierarchy

- Julia has many predefined types organized as a hierarchy



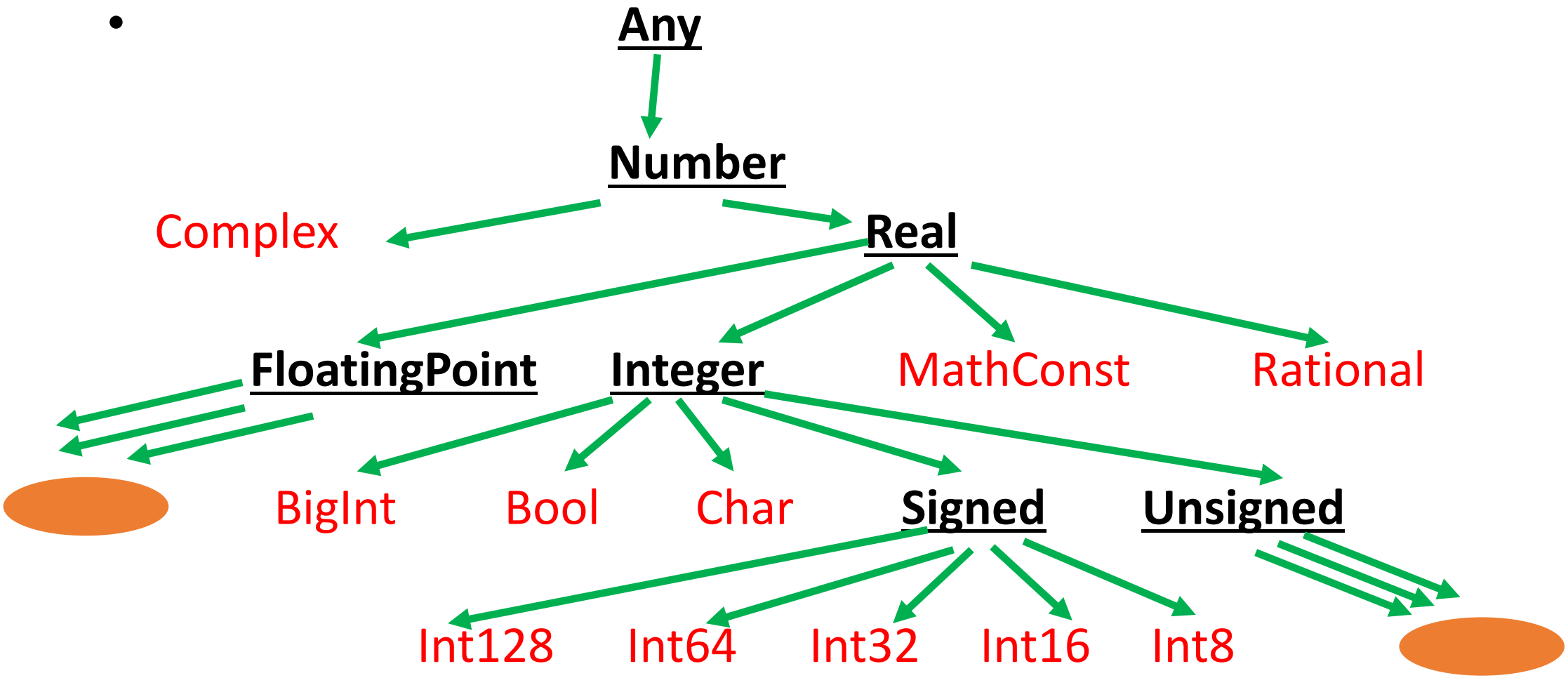
Type hierarchy

- Those have no their own off-springs are called concrete types.



Type hierarchy

- Those have their own off-springs are called **abstract types**
-



Type hierarchy

- Every data/value is a concrete data/value. typeof() returns its concreate type.
- Meanwhile it has also the type of all of its supertypes.

- typeof(3)
- isa(3, Number)
- isa(3, Real)
- isa(3, String)

User defined immutable types

- immutable Point2
- x::Float64
- y::Float64
- z::Float64
- end

- Point2(2.0, 4.0, 5.0)
- a = Point2(2.0, 4.0, 5.0)
- a
- a.y = 0.8
- a
- a= Point2(-2.0, 40.0, 0.05)
- a

Strings are immutable

- str0 = "Volleyball is a sport and an entrainment."
- typeof(ans)
- str0[10]='g'
- str0 = "The blue danube is a great music."

Constants are immutable

- const GC = 6.67e-11 # gravitational constant in m^3/(kg*s^2)

- GC = 7.34

Parametric types

- type Point
- x::Int64
- y::Int64
- end

- type MyPoint{T<:Real}
- x::T
- y::T
- end

- MyPoint(2, 4)
- MyPoint(2.0, 4.0)
- MyPoint(2, 4.0)

Parametric types

- workspace()
- type MyPoint{T1<:Real, T2<:Real}
- x::T1
- y::T2
- end

- MyPoint(2, 4)
- MyPoint(2.0, 4.0)
- MyPoint(2, 4.0)

Parametric types in functions

```
function myfunc_01{T<:Real}(x::T,  y::T)
        return x + y^3
end
```

- myfunc_01(3,2)
- myfunc_01(3.4,2.0)
- myfunc_01(3,2.0)

Parametric types in functions

```
Workspace()
function myfunc_01{T1<:Real, T2<:Real}(x::T1,  y::T2)
        return x + y^3
end
```

- myfunc_01(3,2.0)

Methods

For the same functionality, there could be many specific functions with specific types. Then functions are also called methods.

methods(+)

methods(replace)

methods(myfunc_01)

Julia

- is such a high level computer language, that programmers can completely forget types, letting Julia does everything on this respect;

- is such a high level computer language, that programmers can completely operate types as variables.

Variable scopes, modules, and exception handling

Variable scopes

- The region where a variable is accessible is called the scope of that variable.
- Variables defined/introduced in the top-level areas are accessible anywhere by default, called global variables
- Variables defined in a local scope are only accessible within that scope by default, called local variables.
- Typical local scopes are inside functions and loops.
- In a local scope, the same name local variable hides/blocks the corresponding global one by default.
- Local scope variables can be declared with “global” as global.
- Local variables run faster.
- In the case of many levels of local scope, “local” may be needed to make variables sure.

Variable scopes

- a = 4
- function ff04(x)
- 	return a+x
- end
-
- ff04(3)

Variable scopes

- a = 4
- function ff04(x)
- 	a = 2
- 	return a+x
- end
-
- a
- ff04(3)
- a

Variable scopes

- a = 4
- function ff04(x)
- 	global a = 2
- 	return a+x
- end
-
- a
- ff04(3)
- a

Variable scopes

- a = 4
- function ff04(x)
- global a = 2
- b = 9
- return a+x
- end
-
- b
- ff04(3)
- b

Variable scopes

- a = 4
- function ff04(x)
- global a = 2
- global b = 9
- return a+x
- end
-
- b
- ff04(3)
- b

Variable scopes

```
a = 4
function ff04(x)
        a = 2
        function ff0402()
                a = 1
        end
        ff0402()
        return a+x
end

ff04(3)
```

Variable scopes

```
a = 4
function ff04(x)
        a = 2
        function ff0402()
                local a = 1
        end
        ff0402()
        return a+x
end

ff04(3)
```

Modules

- Julia packages/libraries are coded as modules.
- Packages supply a great amount of useful functions and other tools.
-
```
module MyModule
        (module code)
end
```
- After installed, Modules can be used by “using” or “import”.
- Each package has its own naming space, independent on each other.
- To access anything in a module, the module name should be used, then same name items in different modules are differentiated.
- Our simple normal programming is in the unique Main module.
- Variables of imported modules are read-only.

Package management

- ?Pkg
- Pkg.status()
- Pkg.update()
- Pkg.add("some_new_package")

Modules

- import Winston
- import Gadfly
- Winston.plot(rand(4))
- Gadfly.plot(x=[1:10], y=rand(10))

- Same function name “plot”, but no conflicts, as in different module and the way to access.

Modules

- Module has a type of module: typeof(Main)
- Some modules are loaded as default: Main, Base, Core …

- a = 8
- d = 0.4
- g = "Good morning! "
- names(Main)
- names(Core)
- names(Base)
- whos()
- whos(Base)

Ways to load a module

- https://docs.julialang.org/en/stable/manual/modules/

Summary of module usage

To load a module, two main keywords can be used: `using` and `import`. To understand their differences, consider the following example:

```
module MyModule

export x, y

x() = "x"
y() = "y"
p() = "p"

end
```

Ways to load a module

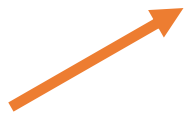
- https:

Import Command	What is brought into scope	Available for method extension
`using MyModule`	All `exported` names (`x` and `y`), `MyModule.x`, `MyModule.y` and `MyModule.p`	`MyModule.x`, `MyModule.y` and `MyModule.p`
`using MyModule.x, MyModule.p`	`x` and `p`	
`using MyModule: x, p`	`x` and `p`	
`import MyModule`	`MyModule.x`, `MyModule.y` and `MyModule.p`	`MyModule.x`, `MyModule.y` and `MyModule.p`
`import MyModule.x, MyModule.p`	`x` and `p`	`x` and `p`
`import MyModule: x, p`	`x` and `p`	`x` and `p`
`importall MyModule`	All `exported` names (`x` and `y`)	`x` and `y`

Ways to load a module

- https:/

Import Command	What is brought into scope	Available for method extension
`using MyModule`	All `exported` names (`x` and `y`), `MyModule.x`, `MyModule.y` and `MyModule.p`	`MyModule.x`, `MyModule.y` and `MyModule.p`
`using MyModule.x, MyModule.p`	`x` and `p`	
`using MyModule: x, p`	`x` and `p`	
`import MyModule`	`MyModule.x`, `MyModule.y` and `MyModule.p`	`MyModule.x`, `MyModule.y` and `MyModule.p`
`import MyModule.x, MyModule.p`	`x` and `p`	`x` and `p`
`import MyModule: x, p`	`x` and `p`	`x` and `p`
`importall MyModule`	All `exported` names (`x` and `y`)	`x` and `y`



suggested

A module example

- module MyModule
- 	my_a = 1
- 	function update()
- 		global my_a += 2
- 	end
- end
-
- import MyModule
- MyModule.my_a
- MyModule.update()
- MyModule.my_a
- MyModule.update()
- MyModule.my_a
- MyModule.my_a =30

Path operations

- The variable LOAD_PATH contains a list of directories where Julia looks for module files when needed.

- LOAD_PATH
- push!(LOAD_PATH, " c:/tmp ")
- LOAD_PATH

Exception handling

```
a = [3, 4, 5]
a[10]


try
    a[10]  #dangerous code
catch  the_expt  # the exception, error,  problem
    println(typeof(the_expt))
    showerror(STDOUT, the_expt)
finally
    println(" End of try structure. ")  #whatever happens, do these.
end
```

More features

Metagramming

- Code a certain frame (macro structure) for a section of code, allowing other lines of code inserted in by “calling” it like a function.

```
macro macro_name(a)
        (more lines including $a)
end
```

Metagramming

```
macro mymacro(a)
        quote
            println(" Beginning in mymacro ... ")
            $a
            println(" end of mymacro. ")
        end
end

@mymacro global a1=1; global a2=2; global a3=a1+a2
(a1, a2, a3)
```

Metagramming

```
macro mymacro(a)
        quote
             println(" Beginning in mymacro … ")
            $a
            println(" end of mymacro. ")
        end
end

@mymacro begin
                a1 = 10;       a2 = 100
                a3 = 1000;  a4 = 10000
                global a5 = a1+a2+a3+a4
              end
a5
```

Built-in macros

- @assert, @test, @test_approx_eq, @which, @show
- @time, @timed, @elapsed
- @async

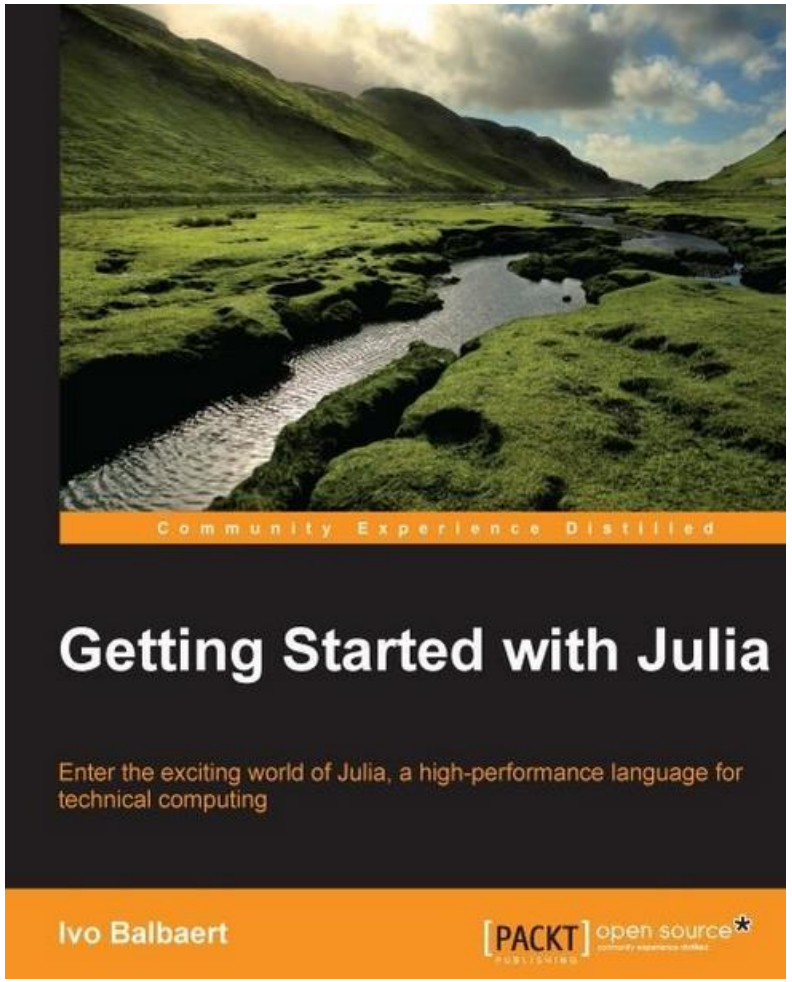
- @elapsed global tt = [x^2 for x = 1:1000]

Julia also supports

- Networking (data transfer over TCP/IP protocol).
- Open Database Connectivity (ODBC package for databases and/or datasources online connections).
- Parallel computing in MPI style.
- Call FORTRAN, C, and Python libraries.

References

Julialang.org



https://github.com/JuliaLang/julia.git

Thank you very much
for your attention!
Have a nice day!