Лабораторная Работа №1

Julia. Установка и настройка. Основные принципы.

Козлов В.П.

Российский университет дружбы народов им. Патриса Лумумбы, Москва, Россия

Докладчик

- Козлов Всеволод Павлович
- НФИбд-02-22
- Российский университет дружбы народов
- [1132226428@pfur.ru]

Выполнение лабораторной работы

Цель работы

Основная цель работы — подготовить рабочее пространство и инструментарий для работы с языком программирования Julia, на простейших примерах познакомиться с основами синтаксиса Julia.

Задание

1. Установите под свою операционную систему Julia, Jupyter.
2. Используя Jupyter Lab, повторите примеры.
3. Выполните задания для самостоятельной работы.

Установил Chocolatey

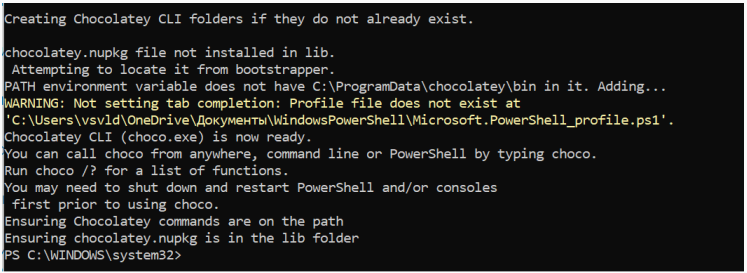


Figure 1: Установка Chocolatey

Установил Far

```
Installing Far...
Far has been installed.
  far may be able to be automatically uninstalled.
 The install of far was successful.
  Deployed to 'C:\Program Files\Far Manager\'

Chocolatey installed 1/1 packages.
 See the log for details (C:\ProgramData\chocolatey\logs\chocolatey.log).

Enjoy using Chocolatey? Explore more amazing features to take your
experience to the next level at
 https://chocolatey.org/compare
PS C:\WINDOWS\system32>
```

Figure 2: Установка Far

Установил Notepad++

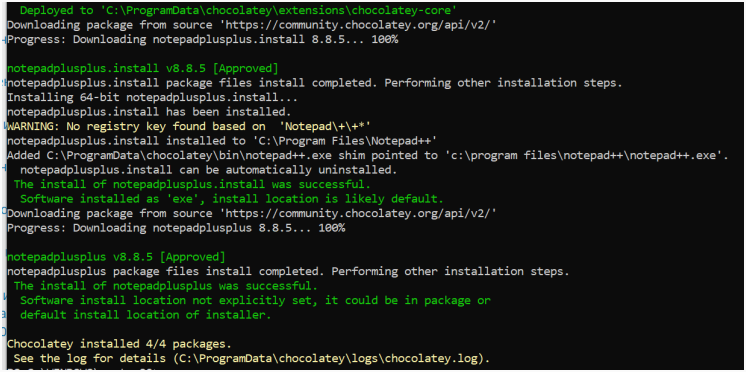


Figure 3: Установка Notepad++

Установил Julia

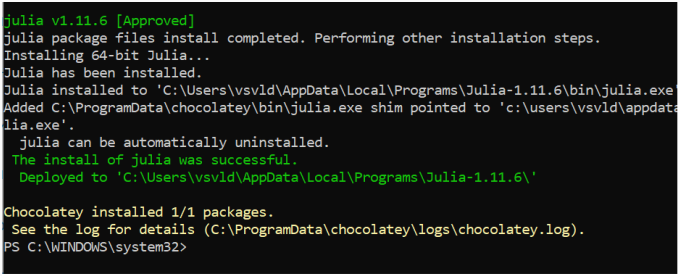


Figure 4: УСтановка Julia

Установил доп пакет IJulia

Figure 5: Установка IJulia

Установил Anaconda3

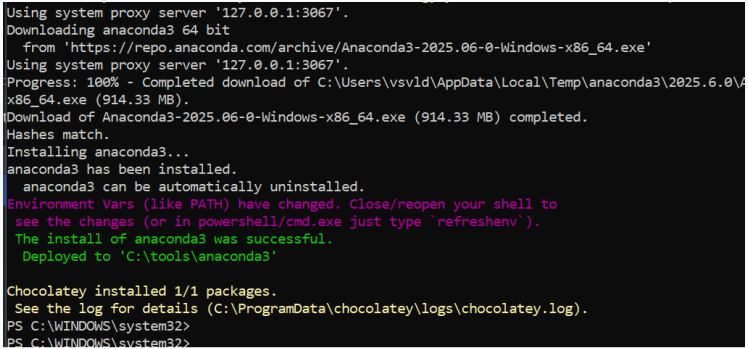


Figure 6: Установка Anaconda3

Документация по println и ее использование

```
[1]: ?println
```

```
search: println print sprint pointer printstyled
```

```
[1]:
```

```
println([io::IO], xs...)
```

Print (using `print`) `xs` to `io` followed by a newline. If `io` is not supplied, prints to the default output stream `stdout`.

See also `printstyled` to add colors etc.

Examples

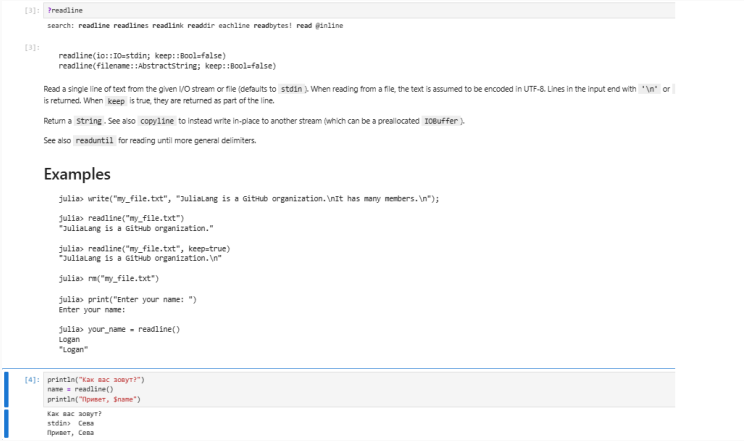
```
julia> println("Hello, world")
Hello, world

julia> io = IOBuffer();

julia> println(io, "Hello", ',', " world.")

julia> String(take!(io))
"Hello, world.\n"
```

```
[2]: println("Я буду учить Julia")
     print("Молодец")
```

```
Я буду учить Julia
Молодец
```

Figure 7: Документация по println и ее использование

Документация по readline и ее использование



```
[3]: ?readline

search: readline readlines readlink readdir eachline readbytes! read @inline

[3]:

    readline(io::IO=stdin; keep::Bool=false)
    readline(filename::AbstractString; keep::Bool=false)

Read a single line of text from the given I/O stream or file (defaults to stdin). When reading from a file, the text is assumed to be encoded in UTF-8. Lines in the input end with '\n' or
is returned. When keep is true, they are returned as part of the line.

Return a String. See also copyline to instead write in-place to another stream (which can be a preallocated IOBuffer).

See also readuntil for reading until more general delimiters.

Examples

    julia> write("my_file.txt", "JuliaLang is a GitHub organization.\nIt has many members.\n");

    julia> readline("my_file.txt")
    "JuliaLang is a GitHub organization."

    julia> readline("my_file.txt", keep=true)
    "JuliaLang is a GitHub organization.\n"

    julia> rm("my_file.txt")

    julia> print("Enter your name: ")
    Enter your name:

    julia> your_name = readline()
    Logan
    "Logan"

[4]: println("Как вас зовут?")
     name = readline()
     println("Привет, $name")

     Как вас зовут?
     stdin>  Сева
     Привет, Сева
```

Figure 8: Документация по readline и ее использование

Документация по readlines и ее использование

[5]: ?readlines

search: **readlines** **readline** **readlink** **readbytes!** **readdir** **eachline** leading_ones

[5]:

```
readlines(io::IO=stdin; keep::Bool=false)
readlines(filename::AbstractString; keep::Bool=false)
```

Read all lines of an I/O stream or a file as a vector of strings. Behavior is equivalent to saving the result of reading `readline` repeatedly with the same arguments and saving the resu
`eachline` to iterate over the lines without reading them all at once.

Examples

```
julia> write("my_file.txt", "JuliaLang is a GitHub organization.\nIt has many members.\n");

julia> readlines("my_file.txt")
2-element Vector{String}:
 "JuliaLang is a GitHub organization."
 "It has many members."

julia> readlines("my_file.txt", keep=true)
2-element Vector{String}:
 "JuliaLang is a GitHub organization.\n"
 "It has many members.\n"

julia> rm("my_file.txt")
```

[7]:

```
open("example.txt", "w") do f
    write(f, "Первая строка\nВторая строка")
end

lines = readlines("example.txt")

println(lines)
println(lines[2])
```

```
["Первая строка", "Вторая строка"]
Вторая строка
```

Figure 9: Документация по readlines и ее использование

Документация по readlm и ее использование

```
julia> using DelimitedFiles

julia> x = [1; 2; 3; 4];

julia> y = [5; 6; 7; 8];

julia> open("delim_file.txt", "w") do io
           writedlm(io, [x y])
       end

julia> readdlm("delim_file.txt", '\t', Int, '\n')
4×2 Matrix{Int64}:
 1  5
 2  6
 3  7
 4  8

julia> rm("delim_file.txt")
```

[13]:
```julia
using DelimitedFiles

open("numbers.txt", "w") do f
    write(f, "1 2 3\n4 5 6\n7 8 9\n")
end

A = readdlm("numbers.txt")
println(A)
```

```
[1.0 2.0 3.0; 4.0 5.0 6.0; 7.0 8.0 9.0]
```

Figure 10: Документация по readlm и ее использование

Документация по show и ее использование

```
show(io::IO, mime, x)
```

The `display` functions ultimately call `show` in order to write an object `x` as a given `mime` type to a given I/O stream `io` (usually a memory buffer), if possible. In order to provide a rich multimedia representation of a user-c
type `T`, it is only necessary to define a new `show` method for `T`, via: `show(io, ::MIME"mime", x::T) = ...`, where `mime` is a MIME-type string and the function body calls `write` (or similar) to write that representation
to `io`. (Note that the `MIME""` notation only supports literal strings; to construct `MIME` types in a more flexible manner use `MIME{Symbol("")}`.)

For example, if you define a `MyImage` type and know how to write it to a PNG file, you could define a function `show(io, ::MIME"image/png", x::MyImage) = ...` to allow your images to be displayed on any PNG-capable
`AbstractDisplay` (such as Julia). As usual, be sure to `import Base.show` in order to add new methods to the built-in Julia function `show`.

Technically, the `MIME"mime"` macro defines a singleton type for the given `mime` string, which allows us to exploit Julia's dispatch mechanisms in determining how to display objects of any given type.

The default MIME type is `MIME"text/plain"`. There is a fallback definition for `text/plain` output that calls `show` with 2 arguments, so it is not always necessary to add a method for that case. If a type benefits from custom
human-readable output though `show(::IO, ::MIME"text/plain", ::T)` should be defined. For example, the `Day` type uses `1 day` as the output for the `text/plain` MIME type, and `Day(1)` as the output of 2-argum
`show`.

Examples

```
julia> struct Day
           n::Int
       end

julia> Base.show(io::IO, ::MIME"text/plain", d::Day) = print(io, d.n, " day")

julia> Day(1)
1 day
```

Container types generally implement 3-argument `show` by calling `show(io, MIME"text/plain"(), x)` for elements `x`, with `:compact => true` set in an `IOContext` passed as the first argument.

```
10]: x = 123.456
     show(x)
     println()
     show(stdout, "text/plain", x)

123.456
123.456
```

Would you like to get notified at
Jupyter news?

Open privacy

Figure 11: Документация по show и ее использование

Документация по write и ее использование

```
44

julia> String(take!(io))
"Sometimes those members write documentation."
```

User-defined plain-data types without `write` methods can be written when wrapped in a `Ref`:

```
julia> struct MyStruct; x::Float64; end

julia> io = IOBuffer()
IOBuffer(data=UInt8[...], readable=true, writable=true, seekable=true, append=false, si

julia> write(io, Ref(MyStruct(42.0)))
8

julia> seekstart(io); read!(io, Ref(MyStruct(NaN)))
Base.RefValue{MyStruct}(MyStruct(42.0))
```

```
write(filename::AbstractString, content)
```

Write the canonical binary representation of `content` to a file, which will be created if it does not exist yet or ov

Return the number of bytes written into the file.

[21]:
```
open("nums.txt", "w") do f
    write(f, "10\n20\n30\n")
end

lines = readlines("nums.txt")
nums = parse.(Int, lines)
println(nums)
```

```
[10, 20, 30]
```

Figure 12: Документация по write и ее использование

Документация по parse и ее использование

julia> parse(Complex{Float64}, "3.2e-1 + 4.5im")
0.32 + 4.5im

---

parse(::Type{Platform}, triplet::AbstractString)

Parses a string platform triplet back into a `Platform` object.

---

parse(::Type{SimpleColor}, rgb::String)

An analogue of `tryparse(SimpleColor, rgb::String)` (which see), that raises an error instead of returning

[23]:
```
# строка в целое число
x = parse(Int, "42")
println(x)
println(typeof(x))
```

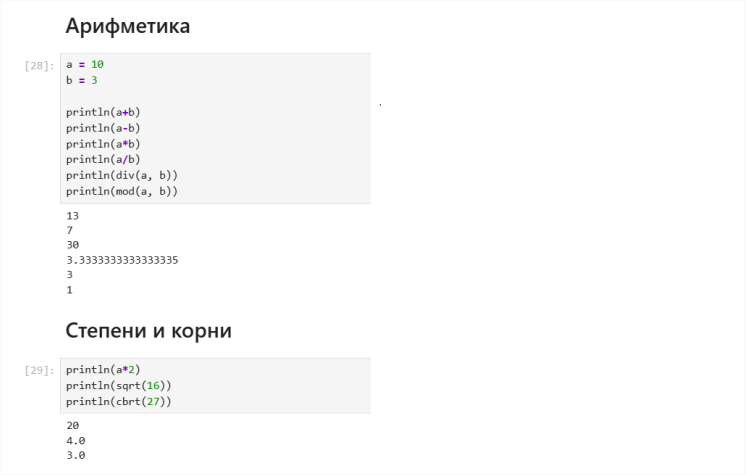
```
42
Int64
```

[25]:
```
# строка в плав точку
y = parse(Float64, "3.14")
println(y)
println(typeof(y))
```

```
3.14
Float64
```

Figure 13: Документация по parse и ее использование

Арифметика, степени и корни в Julia



Арифметика

[28]:
```
a = 10
b = 3

println(a+b)
println(a-b)
println(a*b)
println(a/b)
println(div(a, b))
println(mod(a, b))
```

```
13
7
30
3.3333333333333335
3
1
```

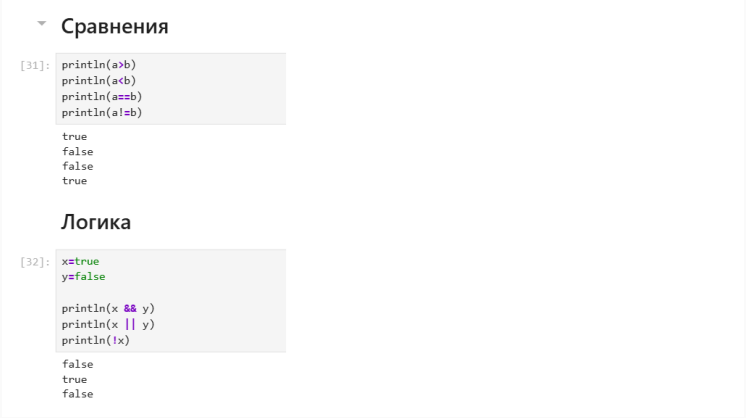
Степени и корни

[29]:
```
println(a^2)
println(sqrt(16))
println(cbrt(27))
```

```
20
4.0
3.0
```

Figure 14: Арифметика, степени и корни в Julia

Сравнения и логика в Julia



Сравнения

[31]:
```
println(a>b)
println(a<b)
println(a==b)
println(a!=b)
```

```
true
false
false
true
```

Логика

[32]:
```
x=true
y=false

println(x && y)
println(x || y)
println(!x)
```

```
false
true
false
```

Figure 15: Сравнения и логика в Julia

Операции с разными типами данных в Julia

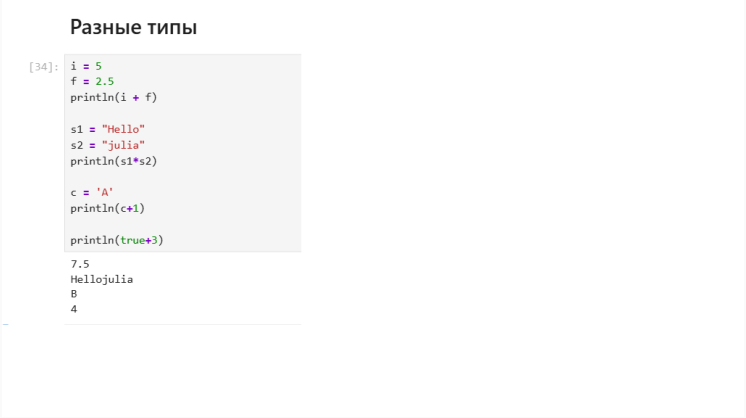


Figure 16: Операции с разными типами данных в Julia

Операции с матрицами в Julia

[36]:
```
using LinearAlgebra

A = [1 2; 3 4]
B = [5 6; 7 8]

println("A = ")
println(A)

println("A + B = ")
println(A+B)

println("A - B = ")
println(A-B)

# скал произв
v1 = [1, 2, 3]
v2 = [4, 5, 6]
println("dot(v1, v2) = ", dot(v1, v2))
```

```
A =
[1 2; 3 4]
A + B =
[6 8; 10 12]
A - B =
[-4 -4; -4 -4]
dot(v1, v2) = 32
```

Figure 17: Операции с матрицами в Julia

Операции с матрицами в Julia

[37]:
```
# транспонирование
println("A' (трансп матрица) = ")
println(A')

# умножение матрицы на скаляр
println("2 * A = ")
println(2*A)

# умножение матрицы на матрицу
println("A * B = ")
println(A*B)
```

```
A' (трансп матрица) =
[1 3; 2 4]
2 * A =
[2 4; 6 8]
A * B =
[19 22; 43 50]
```

[ ]:

Figure 18: Операции с матрицами в Julia

Выводы

Подготовил рабочее пространство и инструментарий для работы с языком программирования Julia, на простейших примерах познакомился с основами синтаксиса Julia.