Отчёт по лабораторной работе №1

Julia. Установка и настройка. Основные принципы

Косолапов Степан Эдуардович НПИбд-01-20

Содержание

# 1 Цель работы

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

# 2 Выполнение работы

1. Изучаем документацию по основным функциям Julia для чтения/записи/вывода информации на экран: read(), readline(), readlines(), readdlm(), print(), println(), show(), write():

read(): Эта функция используется для чтения из файла или потока данных, и возвращает результат в виде типа ByteArray. По умолчанию, она считывает весь файл или поток до его окончания.

io = IOBuffer("JuliaLang is a GitHub organization");  
read(io, String)

"JuliaLang is a GitHub organization"

io = IOBuffer("JuliaLang is a GitHub organization");  
read(io, Char)

'J': ASCII/Unicode U+004A (category Lu: Letter, uppercase)

readline(): Функция считывает линию из файла или потока данных, и возвращает её в виде строки. Это полезно при чтении текстовых документов, где данные организованы построчно.

input = readline()

stdin> Stepa  
  
"Stepa"

print(input)

Stepa

write(): Эта функция используется для записи исходных данных (например, строки или массива байтов) в файл или поток данных. Она позволяет вам контролировать, как именно данные будут записаны, и возвращает количество байтов, которые были успешно записаны.

write("my\_file.txt", "JuliaLang is a GitHub organization.\nIt has many members.\n");

readlines(): Эта функция аналогична readline(), но с той разницей, что она считывает все строки из файла или потока данных и возвращает их в виде массива строк.

lines = readlines("my\_file.txt")

2-element Vector{String}:  
 "JuliaLang is a GitHub organization."  
 "It has many members."

readdlm(): Это функция используется для чтения табличных данных, где значения разделены определенным символом (например, запятой или табуляцией). Она возвращает двумерный массив, где каждая строка представляет собой строку в исходном файле или потоке данных, а каждый столбец представляет собой значение, разделенное символом-разделителем.

using DelimitedFiles  
  
x = [1; 2; 3; 4];  
  
y = [5; 6; 7; 8];  
  
open("delim\_file.txt", "w") do io  
 writedlm(io, [x y])  
end  
  
readDLMLines = readdlm("delim\_file.txt", '\t', Int, '\n')

4×2 Matrix{Int64}:  
 1 5  
 2 6  
 3 7  
 4 8

Если бы в файле был разделитель - запятая(то есть файл формата csv), то можно было бы считать его вот так:

readCSVLines = readdlm("delim\_file.txt", ',', Int, '\n')

4×2 Matrix{Int64}:  
 1 5  
 2 6  
 3 7  
 4 8

print(): Эта функция используется для напечатания значения в файл или поток данных. Она не добавляет символ новой строки после значения, поэтому использование этой функции несколько раз подряд приведет к выводу всех значений на одной и той же строке.

print("hello "); print("world"); print("!")

hello world!

println(): Это функция аналогична print(), но с тем отличием, что она добавляет символ новой строки после значения. Это полезно, когда вам нужно напечатать несколько значений, каждое из которых должно быть на новой строке.

println("hello "); println("world"); println("!")

hello   
world  
!

show(): Эта функция используется для представления значения в читаемом виде. Она работает похожим образом как print(), но с той разницей, что она также может показывать внутреннюю структуру сложных объектов, таких как массивы или пользовательские типы данных.

struct Day  
 n::Int  
end  
  
Base.show(io::IO, ::MIME"text/plain", d::Day) = print(d.n);  
  
Day(1)

1

1. Изучаем документацию по функции parse(). Приведём свои примеры её использования

var = parse(Int, "1234")  
println(var)  
  
var = parse(Int, "1234", base = 5)  
println(var)  
  
var = parse(Int, "101001", base = 2)  
println(var)  
  
var = parse(Int, "afc", base = 16)  
println(var)  
  
var = parse(Float64, "1.2e-3")  
println(var)  
  
var = parse(Complex{Float64}, "3.2e-1 + 4.5im")  
println(var)  
  
println(parse(Bool, "0"))  
println(parse(Bool, "false"))  
println(parse(Bool, "1"))  
println(parse(Bool, "true"))

1234  
194  
41  
2812  
0.0012  
0.32 + 4.5im  
false  
false  
true  
true

Функция parse() в Julia используется для преобразования строки в заданный тип данных. В качестве первого аргумента, функция принимает тип данных, в который нужно преобразовать, а вторым аргументом является строка, которую нужно преобразовать.

Например, если вам нужно преобразовать строку, содержащую число, в целочисленное значение, вы можете использовать parse(Int, “123”), и это вернет целое число 123. Аналогично, если вы хотите преобразовать строку в число с плавающей запятой, вы можете использовать parse(Float64, “123.45”), и это вернет число с плавающей запятой 123.45.

Особенностью функции parse() является ее способность обрабатывать и преобразовывать различные типы данных, включая пользовательские типы данных. Однако, нужно быть осторожным, так как если строка не может быть преобразована в желаемый тип данных, функция вызовет ошибку.

Таким образом, функция parse() является удобным инструментом для преобразования строковых значений во многие другие типы данных в Julia

1. Изучим синтаксис Julia для базовых математических операций с разным типом переменных: сложение, вычитание, умножение, деление, возведение в степень, извлечение корня, сравнение, логические операции.

В языке программирования Julia базовые математические операции работают так же, как и в большинстве других языков программирования. Они включают сложение (+), вычитание (-), умножение (\*), деление (/), целочисленное деление (÷), остаток от деления (%), возведение в степень (^) и извлечение квадратного корня (sqrt()).

Операции сравнения в Julia включают в себя равенство (==), неравенство (!=), меньше (<), меньше или равно (<=), больше (>), больше или равно (>=).

Логические операции включают логические И (&), ИЛИ (|), НЕ (!), исключающее ИЛИ (xor).

Важно отметить, что все эти операции могут быть использованы с переменными различного типа (например, Int, Float64, Complex, Bool и т.д.), но результат и поведение могут варьироваться в зависимости от типов данных.

Приведем примеры:

println(1+1)  
println(1-1)  
println(2\*2)  
println(2^2)  
println(sqrt(4))  
println(2 > 1)  
println(1<=3)  
println(2==2)  
println(2.3 + 1)  
println(sqrt(3.3) == 3.3^(1/2))  
println(1 | 1)  
println(parse(Int, "101010", base = 2) | parse(Int, "10101", base = 2) == parse(Int, "111111", base = 2))  
println(parse(Int, "101010", base = 2) & parse(Int, "10101", base = 2) == 0)  
println(parse(Int, "101011", base = 2) ⊻ parse(Int, "10101", base = 2) == 62)  
println(parse(Int, "101011", base = 2) >> 1 == parse(Int, "10101", base = 2))  
println(parse(Int, "101011", base = 2) << 1 == 86)  
println(parse(Int, "101011", base = 2) >> 2 == 10)

2  
0  
4  
4  
2.0  
true  
true  
true  
3.3  
true  
1  
true  
true  
true  
true  
true  
true

1. Приведите несколько своих примеров с пояснениями с операциями над матрицами и векторами: сложение, вычитание, скалярное произведение, транспонирование, умножение на скаляр.

Julia поддерживает множество математических операций для работы с матрицами и векторами.

Поэлементное умножение (и другие поэлементные операции) производятся с добавлением точки ‘.’ перед оператором.

Любые матричные операции требуют согласования размеров матриц и векторов. Если размеры не согласованы, Julia выдаст ошибку.

Сложение и вычитание матриц и векторов в Julia выполняются поэлементно (покомпонентно).

[1, 2, 3] + [1,2,3]

3-element Vector{Int64}:  
 2  
 4  
 6

[1, 2, 3] - [1,2,3]

3-element Vector{Int64}:  
 0  
 0  
 0

[1 2; 3 4] + [1 2; 3 4]

2×2 Matrix{Int64}:  
 2 4  
 6 8

[1 2; 3 4] - [1 2; 3 4]

2×2 Matrix{Int64}:  
 0 0  
 0 0

Скалярное произведение векторов/матриц вычисляется с использованием функции dot().

using LinearAlgebra  
dot([1,2,3], [1,2,3])

14

dot([1 2; 3 4], [1 2; 3 4])

30

Умножение матрицы или вектора на скаляр также работает обычным для математики образом

[1, 2, 3] \* 3

3-element Vector{Int64}:  
 3  
 6  
 9

[1 2; 3 4] \* 3

2×2 Matrix{Int64}:  
 3 6  
 9 12

Матричное умножение в Julia выполняется с помощью оператора \*.

[1 2; 3 4] \* [1 2; 3 4]

2×2 Matrix{Int64}:  
 7 10  
 15 22

Транспонирование матриц выполняется с помощью функции transpose() либо с помощью определённой в Julia операции.

transpose([1 2; 3 4])

2×2 transpose(::Matrix{Int64}) with eltype Int64:  
 1 3  
 2 4

# 3 Выводы

В данной работе мы подготовили рабочее пространство и инструментарий для работы с языком программирования Julia, на простейших примерах познакомились с основами синтаксиса Julia.