Contents

1. Programming basics in Julia

This lesson is a overview of basic programming concepts using the Julia language and the REPL. The intended audience is people who have never programmed before, or who have only done a little programming in another language.

Goals

1. Download and install Julia and Visual Studio Code
2. Learn the basics of programming using Julia in the REPL (variables, types, basic operations, booleans, and arrays)
3. Find help in the Julia documentation and on the internet

Install Julia and Visual Studio Code

1. Download the latest version of Julia from the official website.
2. Follow the installation instructions for your operating system.
3. Download and install Visual Studio Code.
4. Install the Julia extension for Visual Studio Code by searching for "Julia" in the Extensions Marketplace.
5. Quit Visual Studio Code for now and open the terminal (command line interface) on your computer.

We will use the Julia REPL (Read-Eval-Print Loop) in the terminal for this lesson, not VS Code. VS Code will be used in the next lesson. Here we just install it.

Variables, basic operations, and types

Open the REPL (the command line interface for Julia) by typing `julia` in the terminal. REPL stands for Read-Eval-Print Loop.

Basic operations and variable assignment

Follow along with the code examples below in the REPL. Equals sign = is used for assignment, not equality. Feel free to adapt the code based on student feedback and progression. Remember to ask students to try things out themselves and play with the code. Use the question, "What do you think will happen?" to encourage students to think about the code before running it. Also a useful way of thinking about their optics setups later.

Start with basic operations on one variable.

```
julia> a = 2  # assignment
2

julia> a  # print the value of a
2

julia> a + 1  # addition
3

julia> a - 1  # subtraction
1

julia> a * 2  # multiplication
4

julia> a^2  # exponentiation
4
```

```
julia> a / 2  # division
1.0  # what's going on here?

julia> a = 5  # reassign a

julia> a
5

julia> a + 1
6

julia> a = a + 1  # reassign a using itself
6

julia> a = 5  # reassign a again
5

julia> a += 1  # increment a by 1. This is a shorthand for a = a + 1
6

julia> a
6

julia> a -= 1  # decrement a by 1
5

julia> a *= 2  # multiply a by 2
10

julia> a /= 2  # divide a by 2
5.0  # once again, the type has changed
```

Basic types

Now let's create some variables of different types and explore Julia's type system.

```
julia> a = 1
1

julia> b = 2.0
2.0

julia> c = 3.0 + 4.0im
3.0 + 4.0im

julia> d = "hello"
```

```
"hello"

julia> e = 'a'
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> typeof(a)
Int64

julia> typeof(b)
Float64

julia> typeof(c)
ComplexF64 (alias for Complex{Float64})

julia> typeof(d)
String
```

Questions

1. What happens when you type `a = hello` (no quotes)? What is the meaning of the single and double quotes?
2. What happens if you try to multiply a string by a number? Read the error message.
3. Can you generate other error messages?
4. How do you repeat a string 3 times?
5. Check what `typeof` does by typing `?typeof` in the REPL.

Playing with strings

```
julia> d = "hello"
"hello"

julia> d * d  # string concatenation
"hellohello"

julia> d^3
"hellohellohello"

julia> d = "hello "
"hello "

julia> d^3
"hello hello hello "
```

Logical operators

Logical operators are used to perform logical operations on boolean values. Booleans are a type that can be either `true` or `false`.

```
julia> t = true
true

julia> f = false
false

julia> t && f  # logical AND
false

julia> t || f  # logical OR
true

julia> !t  # logical NOT
false

julia> t == f  # equality
false

julia> t != f  # inequality
true

julia> t === f  # strict equality
false

julia> t !== f  # strict inequality
true
```

Try this by assigning numerical values to a and b (e.g. a = 1, b = 1.0).

Comparison operators

Comparison operators are used to compare values.

```
julia> a = 1
1

julia> b = 2
2

julia> a < b  # less than
true
```

```
julia> a <= b  # less than or equal to
true

julia> a > b  # greater than
false

julia> a >= b  # greater than or equal to
false

julia> a == b  # equal to
false

julia> a != b  # not equal to
true
```

Questions

1. Check how to use the funciton `isa` by typing `?isa` in the REPL.
2. Use `isa` to check the type of `a`, and `b`. What does it return?
3. What type `isa`? How can you check this?
4. What is the difference between `==` and `===`?

Arrays

Arrays are a collection of elements, usually of the same type but not always.

```
julia> v = [1, 2, 3]  # array of integers
3-element Vector{Int64}:
 1
 2
 3

julia> w = [1.0, 2.0, 3.0]  # array of floats
3-element Vector{Float64}:
 1.0
 2.0
 3.0

julia> w = [1, 2.0, 3]  # the mixed array gets converted to Float64 because of the type prom
3-element Vector{Float64}:
 1.0
 2.0
 3.0

julia> a = ["hello", 2.0, 3]  # mixed array has type Any
3-element Vector{Any}:
```

```
 "hello"
 2.0
 3
```

Notice that these arrays are of type `Vector{Int64}`, `Vector{Float64}`, and `Vector{Any}`. The type `Vector` is a one-dimensional array in Julia. It is a subtype of `Array`.

```
julia> typeof(v)
Vector{Int64} (alias for Array{Int64, 1})

julia> typeof(w)
Vector{Float64} (alias for Array{Float64, 1})
```

Questions

1. Try comparing arrays `v` and `w` using the comparison operators.
2. What is the output of `v isa Array`?

Indexing arrays

You can access elements of an array using indexing. Julia uses 1-based indexing, which means that the first element of an array is at index 1, not 0.

```
julia> a[1]  # indexing starts at 1, not 0
"hello"

julia> a[2]
2.0

julia> a[3]
3

julia> a[1] = 1  # you can change the value of an element in the array
1

julia> a
3-element Vector{Any}:
 1
 2.0
 3
```

The length of a vector can be obtained using the `length` function.

```
julia> length(a)
3
```

Matrices

Matrices are two-dimensional arrays in Julia. You can create a matrix using the `reshape` function or by using semicolons to separate rows.

```julia
julia> A = [1 2 3; 4 5 6; 7 8 9]  # matrix with semicolons
3×3 Matrix{Int64}:
 1  2  3
 4  5  6
 7  8  9

julia> B = reshape(1:9, 3, 3)  # matrix with reshape
3×3 reshape(::UnitRange{Int64}, 3, 3) with eltype Int64:
 1  4  7
 2  5  8
 3  6  9

julia> A[1, 2]  # indexing a matrix
2

julia> A[2, 3]
6

julia> A[1, :]  # all elements in the first row
3-element Vector{Int64}:
 1
 2
 3

julia> A[:, 1]  # all elements in the first column
3-element Vector{Int64}:
 1
 4
 7
```

You can also use the `size` function to get the dimensions of a matrix.

```julia
julia> size(A)
(3, 3)
julia> size(B)
(3, 3)
```

Matrix and vector operations

You can perform various operations on matrices and vectors, such as addition, subtraction, multiplication, and division.

```julia
julia> A + B  # matrix addition
```

```
3×3 Matrix{Int64}:
  2   6  10
  6  10  14
 10  14  18

julia> A - B  # matrix subtraction
3×3 Matrix{Int64}:
 0  -2  -4
 2   0  -2
 4   2   0

julia> A * B  # matrix multiplication
3×3 Matrix{Int64}:
 14   32   50
 32   77  122
 50  122  194

julia> A / B  # matrix division
3×3 Matrix{Float64}:
 -0.333333    0.666667  -0.0
  2.26667    -7.53333    5.6
 -6.33333     6.66667   -0.0

julia> A'  # transpose of a matrix
3×3 adjoint(::Matrix{Int64}) with eltype Int64:
 1  4  7
 2  5  8
 3  6  9

julia> A * v  # matrix-vector multiplication. What do you think will happen?
3-element Vector{Int64}:
 14.0
 32.0
 50.0

julia> v * A  # what do you think will happen?
```

You can also perform element-wise operations using the .*, ./, and .^ operators.

```
julia> A .* B  # element-wise multiplication
3×3 Matrix{Int64}:
  1   8  21
  8  25  48
 21  48  81
```

There are more linear algebra operations available if you use the LinearAlgebra

package.

Resources

- [Official Julia Documentation](#)
- [Julia tutorials list](#)
- [The Julia WikiBook](#)
- [The Julia Express](#)

2. Functions and control flow

This lesson introduces students to Visual Studio Code. But first, the concept of files and folders are reviewed. Students create a project folder to store their tutorial files and future experiment code. This ensures consistent data storage and analysis practices in the lab. We also cover environments and how to create a new environment for each project. This is important for reproducibility and to avoid package conflicts.

The main part of the lesson will cover some built-in Julia functions, user-defined functions, for-loops, and if-statements.

Files and folders on a computer

On a computer, files are stored in folders (directories), where a folder can contain both files and other folders. We want to store files and folders in a way that makes it easy to find and organize them. It is important in science to have well-organized data and code for reproducibility.

Follow these steps to create a project folder for your tutorials and experiments:

1. On Windows go to `C:\Users\<username>\Documents\` and create a new folder called `projects`. This is where you will store your projects while you are in the lab. On macOS, go to `~/Documents/` and create a new folder called `projects`.

2. Inside of `projects`, create a new folder called `tutorials`. This is where you will store code for these tutorials.

3. In `tutorials`, create a folder called `intro to programming` or something similar.

4. Open Visual Studio Code and open the `tutorials` folder that you just created.

5. Create a new file called `lesson2.jl` in the `intro to programming` folder. This is where you will write your code for this lesson.

Next we will create a new environment for this project.

Environments

In Julia, an environment is a collection of packages and their versions that are used for a specific project. This is important for reproducibility and to avoid package conflicts.

To create a new environment, follow these steps:

1. Open the Julia REPL in Visual Studio Code via the Command Palette (Ctrl+Shift+P) and type `Julia: Start REPL`.
2. In the REPL, type `]` to enter the package manager. (Hit backspace to exit the package manager and return to the Julia REPL.)
3. Type `activate .` to create a new environment in the current folder.

Notice that a `Project.toml` file and a `Manifest.toml` file were created in your project folder. These files contain information about the packages and their versions that are used in this environment.

When Makie has finished installing and compiling, let's go back to the REPL in VS Code and learn about functions.

Simple functions

Functions in programming operate similarly to mathematical functions. They take inputs (arguments) and produce outputs (return values). Julia has many built-in functions, and you can also define your own functions. Examples: `sin`, `cos`, `exp`, `log`, `sqrt`, `abs`, `round`, `floor`, `ceil`, `max`, `min`.

Play around with these functions in the REPL.
(To instructor: you don't have to do all of the below examples. They are just to show the variety of built-in functions available.)

```
julia> sin(0)
0.0

julia> cos(0)
1.0

julia> exp(0)
1.0

julia> log(1)
0.0

julia> sqrt(4)
2.0

julia> abs(-5)
5
```

```
julia> round(3.14159, digits=2)  # This function has two arguments (inputs).
3.14

julia> floor(3.14159)
3.0

julia> ceil(3.14159)
4.0

julia> max(3, 5)
5

julia> min(3, 5)
3
```

Let's define a simple function that takes one argument and returns the square of that argument.

```
julia> square(x) = x^2  # function definition
square (generic function with 1 method)

julia> square(3)
9

julia> square(4.0)
16.0

julia> square(4.0 + 2.0im)
12.0 + 16.0im
```

You can see that the function automatically works with different types of inputs. This is one of the powerful features of Julia: it can automatically determine the type of the input and return the appropriate type for the output. This is allowed because of Julia's "multiple dispatch" functionality and is a key feature of Julia's design.

Broadcasting

Broadcasting allows you to apply a function to each element of an array or collection. You can use the dot . operator to indicate that you want to apply a function element-wise.

```
julia> f(x) = 1 + x
f (generic function with 1 method)

julia> f(1)
```

```
2

julia> f(1.0)
2.0
```

What happens if we try to apply this function to a vector?

```
julia> v = [1, 2, 3]
3-element Vector{Int64}:
 1
 2
 3

julia> f(v)
ERROR: MethodError: no method matching +(::Int64, ::Vector{Float64})
For element-wise addition, use broadcasting with dot syntax: scalar .+ array
The function `+` exists, but no method is defined for this combination of argument types.

Closest candidates are:
  +(::Any, ::Any, ::Any, ::Any...)
   @ Base operators.jl:596
  +(::Real, ::Complex{Bool})
   @ Base complex.jl:322
  +(::Array, ::Array...)
   @ Base arraymath.jl:12
  ...

Stacktrace:
 [1] f(x::Vector{Float64})
   @ Main ./REPL[95]:1
 [2] top-level scope
   @ REPL[99]:1
```

The error message is telling us that the function + does not know how to add
a scalar to a vector. To fix this, we can use broadcasting with dot syntax.
This is also known as "element-wise" operations, "vectorized" operations, or
"broadcasting".

```
julia> f.(v)  # apply f to each element of v
3-element Vector{Int64}:
 2
 3
 4
```

You can also use broadcasting with other functions.

```
julia> sin.(v)  # apply sin to each element of v
3-element Vector{Float64}:
 0.8414709848078965
```

```
 0.9092974268256817
 0.1411200080598672
```

It is also possible to define a function that takes a vector as input and returns a vector as output.

```
julia> f(x) = x .+ 1  # broadcasting with dot operator
f (generic function with 1 method)
julia> f(v)
3-element Vector{Int64}:
 2
 3
 4
```

or using the @. macro

```
julia> @. f(x) = x + 1  # broadcasting with @. macro
f (generic function with 1 method)
julia> f(v)
3-element Vector{Int64}:
 2
 3
 4
```

Control flow

Control flow statements allow you to control the flow of execution in your code. This includes `if` statements, `for` loops, and `while` loops.

If statements

If statements allow you to execute code conditionally based on a boolean expression.

```
x = 5
if x > 0
    println("x is positive")
elseif x < 0
    println("x is negative")
else
    println("x is zero")
end

# output: x is positive
```

Now change the value of `x` to a negative number and run the code again.

```
x = -5  # output: x is negative
```

What happens if you don't include the `end` statement?

Loops

Now let's make a loop that prints the numbers from 1 to 10.

```
for i in 1:10
    println(i)
end
```

There are also `while` loops, which execute code while a condition is true.

```
i = 1
while i <= 10
    println(i)
    i += 1
end
```

Functions with multiple arguments

It is more efficient in Julia to put code inside of functions. This allows you to reuse code and avoid writing the same code over and over. You can define functions with multiple arguments by separating the arguments with commas.

```
function add(x, y)
        return x + y
    end
```

```
add(1, 2)  # 3
```

Let's package an if statement and a for loop into a function.

```
function print_numbers(n)
    for i in 1:n
        if i % 2 == 0
            println("$i is even")
        else
            println("$i is odd")
        end
    end
end
```

```
print_numbers(10)
```

This would have been tedious to write in the REPL.

3. Data files and Fitting

Resources

If you need some help remembering least squares fitting, here are some resources:

- Least Squares Fitting on Wolfram MathWorld
- Basic tutorials on residuals and least squares regression
- Ledvij, M. "Curve Fitting Made Easy." Industrial Physicist 9, 24-27, Apr./May 2003.

4. Optimizing a function

In the final experiment tutorial, you measured the spectrum for a polariton system. If the two polariton peaks have the sample transmission amplitude, then the Rabi splitting is the frequency (energy) difference between the two peaks. Another way to extract the Rabi splitting is to measure the angle-resolved spectrum, plot the energy versus incidence angle, and fit the data to a simple coupled harmonic oscillator model. We did not measure the angle dependence, so this lesson provides some fake data to work with.

Since the data is more complicated than a simple curve, we will write our own loss function and minimize it using the `Optim.jl` package. This is often required if the data do not fit to a simple model, or you wish to use a specific algorithm. (Julia has a rich ecosystem of machine learning and optimization packages that you may wish to take advantage of during your studies).

As a reminder, we want to minimize the square of the errors between the model and the raw data:

$$\text{min}\sum_i \left(y_\text{model}(x_i) - y_\text{i, data} \right)^2.$$

In this case, we will use generated data representing the polariton dispersion curve from an angle-resolved measurement.

To fit this data, we use a simple coupled-oscillator Hamiltonian:

$$H_\text{total} = H_0 + H'$$

$$H_\text{total} = \begin{pmatrix} E_\text{cavity} & 0 \\ 0 & E_\text{vibration} \end{pmatrix}$$

- $$\begin{pmatrix} 0 & \Omega_R / 2 \\ \Omega_R / 2 & 0 \end{pmatrix} = \begin{pmatrix} E_\text{cavity} & \Omega_R / 2 \\ \Omega_R / 2 & E_\text{vibration} \end{pmatrix}$$

The cavity mode energy is E_cavity, which depends on the angle θ of the beam with respect to the normal of the cavity surface. The molecular vibrational transition is $E_\text{vibration}$, and the Rabi splitting is

Ω_R. Diagonalize the Hamiltonian to find the upper and lower polariton energies.

Finally, write an error function to minimize the lower and upper polariton energy equations.