**1 Slide**

Tell about plan for today lecture:

1. Presentation
2. Practical part with overview of Julia

**2 Slide**

When a new programming language is created, this is done in order to save the advantages of the old languages and get rid of their limitations.

For example, from these considerations, Guido van Rossum created Python in the late 1980s to improve program language ABC.

**3 Slide. What is Julia?**

The same worked for creators of Julia. At the slide you can see Memorandum of Julia’s creators. So you can see the intentions of creators and how ambitious they are.

They want to keep the good from other languages and discard the bad, but instead of replacing one language, creators want to replace all of them.

**4 Slide. What is Julia?**

Julia is a high-level, dynamically typed, high-performance programming language for mathematical computing.

**5 Slide. Comparison with other languages**

Therefore, Julia is related to many programming languages. And let's look at it in comparison with this other languages

The syntax of the language is similar to the syntax of other mathematical languages (for example, MATLAB and Octave), but it has some significant differences.

Julia is written in C, C ++ and Scheme. JIT compiler included as standard

**6 Slide. Benchmark**

**7 Slide. Relationship with other languages**

But, Julia is friends with all other languages, and is not opposed to them.

Julia has foreign function interfaces for C, Fortran, C++, Python, R, Java, and many other languages. So, Julia can interface directly with external libraries written in C and Fortran.

Julia can also be embedded in other programs through its embedding API.

Specifically, Python programs can call Julia using PyJulia. and even share data between Python and Julia.

R programs can do the same with R's JuliaCall, which is demonstrated by calling MixedModels.jl from R.

**8 Slide. Julia and Python**

Julia is most often compared to Python.

There are more than 8 million Python developers in the world who regularly use this language for a variety of purposes. Due to its flexibility and easy scalability, Python has already become the language of choice for many developers, in many areas of use.

**Similarities:**

Julia has automatic memory management. Like Python, Julia doesn’t burden the user with the details of allocating and freeing memory

Julia has a straightforward syntax. Julia’s syntax is similar to Python’s—easy, but also expressive and powerful.

**Difference:**

Actually there are a plenty of differences, but one of the key difference between Julia and Python is how they approach the same task. While Julia is specifically designed to tackle high performance computing challenges, Python has come to this in its development. While Python has so far been able to meet the challenges of the industry, let's agree that it was not designed for the job. Developers and researchers are fortunate enough to let Python evolve and watch it evolve into a fast computing language. On the other hand, Julia is specially designed with high speed in mind.

**9 Slide.** **Python advantages**

1. Python has less startup overhead. Python programs may be slower than Julia programs, but the Python runtime itself is more lightweight, and it generally takes less time for Python programs to start and deliver first results. Also, while JIT compilation speeds up execution time for Julia programs, it comes at the cost of slower startup. Much work has been done to make Julia start faster, but Python still has the edge here.
2. Python is mature. Community bigger. More Libraries, so on
3. So, we can conclude, that the beauty of Python is not in Python itself, but in the great number of libraries. Some of which can even be called full-fledged tools. And until a critical mass of such libraries is available in a similar form in any other language, the majority will not switch to this language.

**10 Slide.** **Julia Features**

Before that, we said that Julia was originally made for tasks other than Python. Let's take a look at what it was made for

**Data Visualization and Plotting**

Plotting software makes trade-offs between features and simplicity, speed and beauty, and a static and dynamic interface.

Plots.jl is a visualization interface and toolset. It provides a common API across various backends, like GR.jl, PyPlot.jl, and PlotlyJS.jl. Makie.jl is a sophisticated package for complex graphics and animations

**11 Slide.** **Julia Features**

**Interact with Data**

The Julia data ecosystem provides DataFrames.jl to work with datasets, and perform common data manipulations. CSV.jl is a fast multi-threaded package to read CSV files and integration with the Arrow ecosystem is in the works with Arrow.jl. Online computations on streaming data can be performed with OnlineStats.jl. The Queryverse provides query, file IO and visualization functionality. In addition to working with tabular data, the JuliaGraphs packages make it easy to work with combinatorial data.

Julia can work with almost all databases using JDBC.jl and ODBC.jl drivers. In addition, it also integrates with the Spark ecosystem through Spark.jl.

**12 Slide.** **Julia Features**

**Scalable Machine Learning**

The MLJ.jl package provides a unified interface to common machine learning algorithms, which include generalized linear models, decision trees, and clustering. Flux.jl and Knet.jl are powerful packages for Deep Learning. Packages such as Metalhead, ObjectDetector, and TextAnalysis.jl provide ready to use pre-trained models for common tasks. AlphaZero.jl provides a high peformance implementation of the reinforcement learning algorithms from AlphaZero. Turing.jl is a best in class package for probabilistic programming.

**13 Slide.** **Julia Features**

**Rich Ecosystem for Scientific Computing**

Julia is designed from the ground up to be very good at numerical and scientific computing. This can be seen by taking into account how many scientific tooling written in Julia, such as the state-of-the-art differential equations ecosystem (DifferentialEquations.jl), optimization tools (JuMP.jl and Optim.jl), iterative linear solvers (IterativeSolvers.jl), Fast Fourier transforms (AbstractFFTs.jl), and much more. General purpose simulation frameworks are available for Scientific Machine Learning, Quantum computing and much more.

Julia also offers a number of domain-specific ecosystems, such as in biology (BioJulia), operations research (JuMP Dev), image processing (JuliaImages), quantum physics (QuantumBFS), nonlinear dynamics (JuliaDynamics), quantitative economics (QuantEcon), astronomy (JuliaAstro) and ecology (EcoJulia). With a set of highly enthusiastic developers and maintainers, the scientific ecosystem in Julia continues to grow rapidly.

**14 Slide.** **Julia Features**

**Parallel and Heterogeneous Computing**

Julia is designed for parallelism, and provides built-in primitives for parallel computing at every level: instruction level parallelism, multi-threading, GPU computing, and distributed computing. The Celeste.jl project achieved 1.5 PetaFLOP/s on the Cori supercomputer at NERSC using 650,000 cores.

The Julia compiler can also generate native code for GPUs. Packages such as DistributedArrays.jl and Dagger.jl provide higher levels of abstraction for parallelism. Distributed Linear Algebra is provided by packages like Elemental.jl and TSVD.jl.

Our favorite MPI style parallelism is also available through MPI.jl.

**15 Slide.** **Julia Features. Other**

**Julia combines the benefits of dynamic typing and static typing.** You can specify types for variables, like “unsigned 32-bit integer.” But you can also create hierarchies of types to allow general cases for handling variables of specific types—for instance, to write a function that accepts integers without specifying the length or signing of the integer. You can even do without typing entirely if it isn’t needed in a particular context.

A major target audience for Julia is users of scientific computing languages and environments like Matlab, R, Mathematica, and Octave. Julia’s syntax for math operations looks more like the way math formulas are written outside of the computing world, making it easier for non-programmers to pick up on.

Unlike Python, Julia is not an object-oriented language, so some very neat Python-style approaches will have to be abandoned.

Julia has no classes, so you have to be content with structures just like MATLAB.

Another thing that needs attention in Julia is the completion of functions and loops with the **end** keyword. Julia's insensitivity to whitespace, so you will never encounter the error “incompatible use of tabs and spaces in indentation”.

**16 Slide. Speed!**

Julia is compiled, not interpreted. For faster runtime performance, Julia is just-in-time (JIT) compiled using the LLVM compiler framework. At its best, Julia can approach or match the speed of C.

In 2017, Julia even joined the Petaflop Club - a small club of languages that can reach speeds of one petaflop per second at maximum performance. Apart from Julia, the club now only has C, C ++ and Fortran.

**17-19 Slide. Time comparison**

Now let's take a closer look at comparing the execution times of different programs

**20 Slide. What will happen next? Should I teach?**

**Big Plans for Julia’s Future Applications**

Though data scientists and mathematicians favor the language, it has other applications in many industries including:

1. 3D printing
2. Augmented reality
3. Self-driving cars
4. Genomics
5. Risk management
6. Precision medicine
7. Machine learning

The language has a chameleon-like quality that allows coders to shape it to their needs.

Julia offers easy-to-learn syntax, incredible speed of code execution, built-in Python interpreter and many other potential improvements in the work of a specialist.

**Practice:**

Let's calculate the first 10,000 primes using simple functions

7.069463 seconds

Time it took: 136.28955698013306

**Debugging – looks scary imho**

**Julia supports three main categories of features for concurrent and parallel programming:**

**Asynchronous "tasks", or coroutines**

Julia Tasks allow suspending and resuming computations for I/O, event handling, producer-consumer processes, and similar patterns. Tasks can synchronize through operations like wait and fetch, and communicate via Channels.

**Multi-threading**

Multi-threading functionality builds on tasks by allowing them to run simultaneously on more than one thread or CPU core, sharing memory.

**Distributed computing**

Finally, distributed computing runs multiple processes with separate memory spaces, potentially on different machines. This functionality is provided by the Distributed standard library as well as external packages like MPI.jl and DistributedArrays.jl.