How I use Julia

Andy Ferris

How I use Julia - Why?

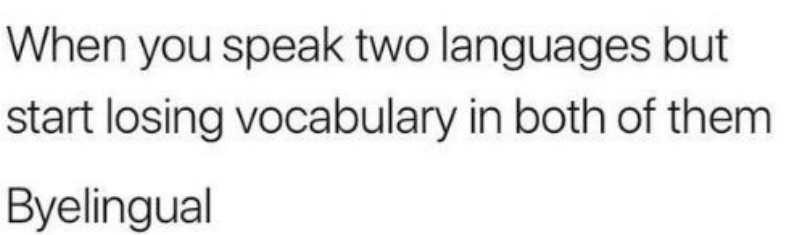
- See what Julia does for me

- Learn how to approach your problems:
  - What kind of problem do you have?
  - How do you script?
  - Or write libraries?
  - How about applications? Or a website?

About me (Andy Ferris)

- Programming computers as a child
- Undergraduate/PhD/Post-docs in quantum physics
  - Lot’s of numerical simulations, etc
- Geospatial (industry)
  - Lidar, imagery, feature extraction
  - Petabyte scale, AWS
- Business optimization (startup)
  - Data processing, integrations
  - Economic simulation and optimization

The “two-language problem”



Your **productivity is limited** by how fast you can write and iterate your ideas. Use:

- Python, MATLAB, R, etc (dynamically execute code, lots of functionality)

Your **productivity is limited** also by how fast your programs can produce output. Use:

- C++, Rust, Fortran, etc (strongly typed with optimizing compiler)

Finally - **your productivity is limited** by having to code the same idea twice!

Before Julia, the two-language problem was my *life*

For research, I would typically prototype algorithms in MATLAB and port them to C++ (and deploy to a cluster) to produce publication results.

In one case we even had a program (XMDS) which transformed XML into a C program as text, which was then compiled... to solve a variety of stochastic PDEs (“metaprogramming” - programs that write programs).

Also a common pattern in industry:

- data scientists create models and algorithms
- passed on to engineering team to deploy

A quick note about Python

Python is a great language.

Extensions like numpy can make it very fast. They are written in C, C++, etc.

However - if you write custom algorithms you either need to

- Use slower native python code
- Write extensions in C, C++, learn Cython/numba/Pypy/etc

Thus it suffers from its own form of the two-language problem.

In many cases it is a great choice - especially the well-trodden path.

Julia ...

- … can dynamically execute code, like an interpreted language
- … is very expressive and quick to write
- … has batteries included
- … has best-in-class ecosystem for mathematics-heavy problems
- … is a strongly-typed language
- … uses an optimizing compiler (LLVM) to produce very fast machine code

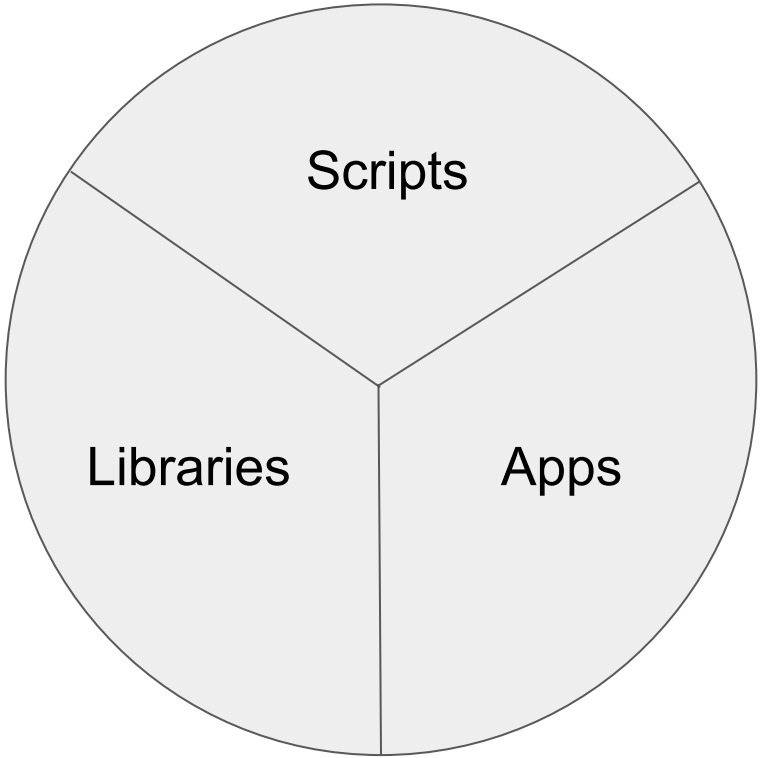
So it solves the “two language problem”.

But there’s more! Julia …

- … has out of this world support for reusing code and creating libraries
- … has built-in metaprogramming capabilities

INTERMISSION

The three types of Julia code



One-off scripts

Scripts are pieces of code that you put together to achieve some defined task.

Useful for both research and performing bespoke business/analysis/processing tasks.

- Analyse data
  - E.g. query a database for some data, perform some analysis, save results to disk.
- Run simulation or solve an optimization problem
  - E.g. run many simulations with different parameters, plot the results for a research paper.
- Ad-hoc data processing
  - e.g. download gigabytes of data and transform them to a particular format for a client.

Julia REPL

The “read evaluate print loop” (REPL) is a great way to explore.

You can:

- Load, save and plot data
- Experiment with functions and ideas
- Get help and read documentation
- Install and use packages
- Convenient: autocompletion, history, prompt-pasting, shell mode, etc.

DEMO - data analysis in the REPL

Julia script files

Julia scripts are text files ending in .jl.

DEMO - putting the analysis inside a script

Loads of awesome tools for scripting & literate programming

Jupyter notebooks

Pluto notebooks

Weave.jl

VS Code

Debugger.jl / Rebugger.jl

INTERMISSION

Libraries

A library is a place to put functionality that can be reused in other code (scripts, applications or even other libraries).

A library is more useful if it can be applied to more programs.

A library is more useful if it can be combined with other libraries.

Julia Packages

A Julia **package** is some code managed by Julia's package manager.

- It has some code in a */src* directory
- Importing the package means loading the file */src/PackageName.jl*
- That file is expected to contain a **module** (or namespace) called *PackageName*
- The */test/runtests.jl* file runs unit tests
- The Project.toml file has the package version and names the dependencies
- You can register a package on a registry, like Julia's public "general" registry, so other people can download and use it

DEMO - creating a simple *GroupCount* package

Advice on how to create useful packages

First, decide what your goal is. Examples from my geospatial past:

- A library for researchers to manipulate geometry and point clouds to help them build fast algorithms quickly and easily.
- A closed-source package for accessing our company's bespoke big data store and download 3D point clouds with a specific format.

The first one should be lightweight and support many applications by using standard Julia interfaces (AbstractArray, etc), work with Float64 or Float32, 2D and 3D, etc.

The second one has a fixed set of functionality, to achieve a specific task. There is no need to be generic whatsoever.

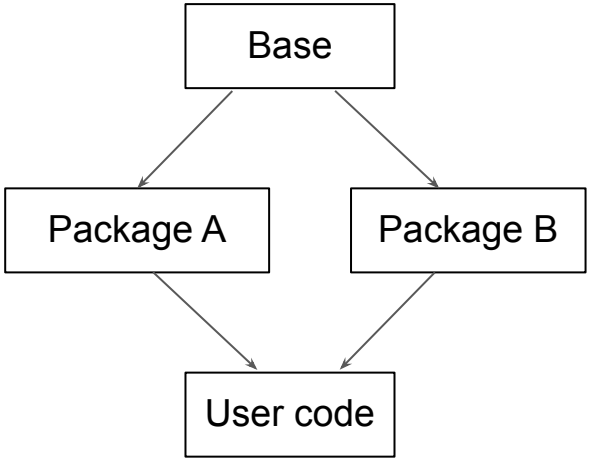
Composable and reusable code

Libraries export various datatypes and functions.

Ideally, you can use a function from Package A with and datatype from Package B without the authors of A and B being aware of each other's code.

You also want the resulting code to work fast as possible - without additional run-time indirection or conversions.

For composable packages, the resulting power is the **product** of the functionality, not the sum!



How to write generic code...

I could (and have) talked for hours on how to do generic programing in Julia.

That’s a nice topic for another day, but the tagline would be “**support blind composition at all costs**”.

INTERMISSION

Writing applications in Julia

Q: What the heck is an “application”, anyway?

A: An app

SOFTWARE TERMINOLOGY

Then	Now
application	app
program	app
operating system	app
script	app
shell	app
batch file	app
compiler	app
daemon	app
service	app
game	app
patch	app
software	app

What makes an app an app?

Research scripts should be repeatable - always calculate the same answer.

Library code - useful components are predictable - exports “pure” functions

An application **reads inputs** and **writes outputs**. It might stay **alive** a long time and maintain **state**.

Exactly what it does **isn't determined by just the code**. Its primary purpose is to cause **side effects**!

Typically, a non-programmer can use it to **do** stuff or to **interact** with it.

Step-by-step guide to creating an app

1. Create an environment of fixed packages for your app
    a. Project.toml (dependencies)
    b. Manifest.toml (locked dependency versions)

2. Create a Julia script that will act as your *main()* function

3. Package your app in some way
    a. A simple executable shell script that launches your script with your environment
    b. A docker container
    c. Create a binary (e.g. PackageCompiler.jl)

Command-line tools

Demonstrate creating a simple command line tool:

ls - unix command, short for “list” or “list directory” (DOS uses *dir*)

jlls - Julia version of *ls*

DEMO - jlls

Packaging your app with *PackageCompiler.jl*

Using a single binary is a lot easier to handle than:

- Installing Julia
- Downloading packages
- Running a script

The *PackageCompiler.jl* package can turn your script into a single executable file.

It can reduce load time by precompiling all the code.

The binary still includes a full copy of Julia, the compiler, package code, etc - so it isn’t minimal and small.

DEMO - *PackageCompiler.jl*

Server application

A server or daemon application stays alive and answers requests.

It generally doesn’t quit unless it is explicitly told to, instead waits for messages.

Webservers, databases, game servers, ssh daemons are all examples.

Libraries make it easy to create a web server - let’s do it!

DEMO - “hello world” webserver

Packaging your service with Docker

Docker let’s you create a reproducible environment for your service.

Basically a lightweight “virtual machine” sandboxed by the OS kernel.

You define a docker image with a Dockerfile (like a Makefile) via *docker build*.

You run a container based off that image via *docker run*.

DEMO - dockerized webserver

THANK YOU - ANY QUESTIONS?